PEL

Actividad Colaborativa 2

Yago Iglesias Díaz

Iván Alexander Morán Neira

Javier Comas González

Marcos Eladio Somoza Corral

Alejandro García García

20/11/2021

ÍNDICE

1. Documentación del ejercicio 1

```
struct Pila {
    int pNum;
    Pila *siguiente;
};
```

Se crea una estructura de nombre **Pila**, donde almacenaremos los valores que se piden al usuario y un puntero con el valor de la posición que sigue al dato. Por medio de varios métodos interaccionamos con la Pila tal y como se pide en el ejercicio.

```
void addPila(Pila *&pila, int num) {
    Pila *nuevaPila = new Pila();
    nuevaPila->pNum = num;
    nuevaPila->siguiente = pila;
    pila = nuevaPila;
}
```

Mediante el método **addPila**, al que le pasamos la Pila (como puntero por referencia) y el valor int, crearemos una nueva Pila, reservaremos la memoria por parte del 'new'. Por medio del operador '->' se asigna el número y la siguiente pila a la que estamos creando. Almacenamos por último la pila generada en la variable 'pila'.

Ya en el método main, inicializamos lo primero el puntero pila como *NULL*, creamos una variable num para almacenar los valores solicitados y poder compartirlos con los métodos como parámetros y, por medio de un bucle while, mostramos las posibles opciones y pedimos una selección por medio de un número introducido por teclado.

Las posibles selecciones serían:

1 - Almacenar nuevo valor

```
case 1:
        cout << endl;
        cout << "Introduce un numero:";
        cin >> num;
        addPila(pila, num);
            break;
```

Donde solicitamos un valor numérico en terminal por medio de *'cin'* que se pasará como parámetro al método **addPila** anteriormente visto. Al ser un primer valor, la variable pila será null, hasta que se le adjudique el espacio en memoria.

2 - Ver valores almacenados

```
case 2:
        cout << endl;
        cout << "Elementos de la pila:" << endl;
        while (pila != NULL)
        {
            getPila(pila, num);
            cout << num << endl;

        }

            break;
```

Este caso cuenta con un *while* que recorrerá, con ayuda del método **getPila** y los parámetros de la anterior pila introducida, los valores que se encuentren almacenados. Parará cuando se encuentre con un valor nulo de pila (cuando se encuentre con el final de ésta).

```
void getPila(Pila *&pila, int &num) {
    Pila *x = pila;
    num = x ->pNum;
    pila = x->siguiente;
}
```

El método **getPila** recibe un puntero referenciado de pila y un valor, que serán los últimos que se hayan almacenado. Por medio de la variable auxiliar 'x' encontramos el valor y la pila siguiente a obtener, que nos ayudará a recorrer las posiciones de una forma similar a la que usamos para guardar los valores.

3 - Eliminar valores

```
case 3:
        while (pila != NULL)
        {
            deletePila(pila, num);
        }
        cout << "Pila borrada" << endl;

            break;
```

A la hora de eliminar los valores, contamos con otro *while* que nos indicará, al igual que en el método donde podemos obtenerlos, el final de la pila. Llamamos al método **deletePila** que irá borrando uno a uno los valores que se le pasen como parámetros.

```
void deletePila(Pila *&pila, int &num) {
    Pila *x = pila;
    num = x ->pNum;
    pila = x->siguiente;
    delete x;
}
```

Funciona también de forma muy similar al método **getPila**, obtenemos la estructura de datos exacta que queremos eliminar y la borramos por medio de la palabra ‘delete’.

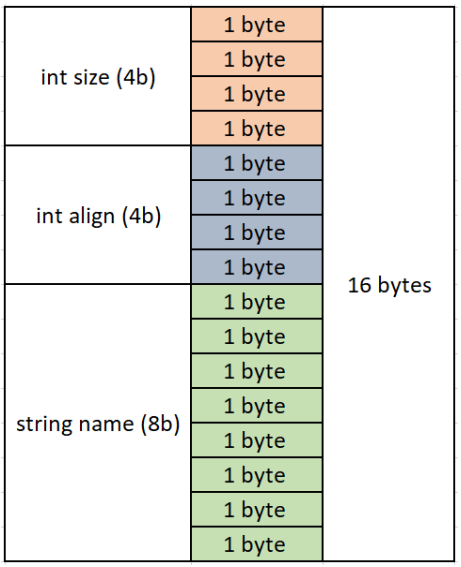
S - Salir

```
default:
            c = false;
            break;
```

Para salir del programa, finalizar su ejecución, se utiliza el caso *‘default’* del switch donde tenemos las opciones que cambiará la variable del bucle *while* que engloba el switch a *‘false’* y la ejecución llegará al final del main. Realmente podríamos escribir cualquier carácter que fuese distinto de 1, 2 y 3 para que entrase en el *‘default’*, para facilitar la salida del programa.

2. Documentación del ejercicio 2

Para implementar este ejercicio, se ha creado una clase **BagParent** que contendrá la información de cada objeto que se quiera "meter" en la mochila. Dicha información consta del tamaño, la alineación y el nombre (el tipo) además de un método para imprimir la información. Cabe destacar que, para optimizar dicha clase en memoria, se ha alineado con alignas(8) y ordenar las variables para optimizar su alineación tal que primero aloja dos int (4 bytes) y luego un string de 32 bytes de tamaño pero 8 bytes de alineación.




```
class alignas(8) BagParent
{
public:
    int size;
    int align;
    string name;
    void PrintInfo() { cout << "variable type: " << name << " and size of:
" << size << ", and align of: " << align << endl; };
};
```

Por otro lado, como se quiere "almacenar" cualquier tipo de objeto, se ha creado una clase **BagClass** (que hereda de **BagParent**) a modo de template, que almacenará el tipo deseado así como asignar su alineación, tamaño y nombre mediante el constructor. Cabe destacar que esta vez se ha especificado la alineación del tipo de objeto que se pase con **alignas(T)**.

```
template <class T> class alignas(T) BagClass : public BagParent
{
public:
    T m_data;
    BagClass()
    {
        this->align = alignof(T);
        this->size = sizeof(T);
        this->name = typeid(m_data).name();
    };
};
```

Por otro lado, se ha creado una clase **Menu** para facilitar el manejo de menús del programa, esta clase contiene un atributo int selección que guarda la última selección del usuario. También los métodos **PrintBaseMenu()**, encargado de mostrar el menú inicial, **PrintMenu()**, encargado

de mostrar el menú de añadido de tipos así como el método **PrintRemove()** que lee la selección del usuario.

```
class Menu
{
private:
    int selection;
public:
    int PrintBaseMenu()
    {
        cout << "Que desea hacer: " << endl;
        cout << "1. Anadir" << endl;
        cout << "2. Eliminar" << endl;
        cout << "3. Ver los elementos" << endl;
        cout << "4. Salir" << endl;
        cin >> selection;
        return selection;
    }
    int PrintRemove()
    {
        cin >> selection;
        return 0;
    };
    int PrintMenu()
    {
        cout << "Selecciona un tipo a guardar [1-8]: " << endl;
        cout << "1. short" << endl;
        cout << "2. char" << endl;
        cout << "3. int" << endl;
        cout << "4. long int" << endl;
        cout << "5. float" << endl;
        cout << "6. double" << endl;
        cout << "7. puntero" << endl;
        cout << "8. long double" << endl;
        cin >> selection;
        return selection;
    };
};
```

Finalmente, se ha creado la clase **Bag** que almacenará los diferentes objetos, esta clase tiene como atributo un **std::vector** de tipo **BagParent**. También cuenta con los métodos **Show()**, que recorre el vector para llamar al **PrintInfo()** de cada elemento, **Add(Menu menu)**, que añade un nuevo elemento al vector del tipo correspondiente al elegido en **menu.PrintMenu()**, y el método

Remove(), que mostrará todos los elementos actuales del vector y eliminará del vector el elemento elegido por el usuario.

```
class Bag
{
public:
    vector<BagParent> bag;
    void Show()
    {
        for (BagParent i : bag)
        {
            i.PrintInfo();
        }
    };
    void Add(Menu menu)
    {
        switch (menu.PrintMenu())
        {
        case 1: bag.push_back(BagClass<short>()); break;
        case 2: bag.push_back(BagClass<char>()); break;
        case 3: bag.push_back(BagClass<int>()); break;
        case 4: bag.push_back(BagClass<long int>()); break;
        case 5: bag.push_back(BagClass<float>()); break;
        case 6: bag.push_back(BagClass<double>()); break;
        case 7: bag.push_back(BagClass<int *>()); break;
        case 8: bag.push_back(BagClass<long double>()); break;
        default:
            break;
        }
        cout << "\033c";
    };
    void Remove()
    {
        for (size_t i = 0; i < bag.size(); i++)
        {
            cout << i + 1 << ".";
            bag[i].PrintInfo();
        }
        int selected = 0;
        cin >> selected;
        bag.erase(bag.begin() + (selected - 1));
    };
};
```

Finalmente, se ha creado una función **Init(Bag bag, Menu menu)** que iniciará el programa, llamando a **menu.PrintBase()** y en función de la elección del usuario, a **bag.Add(menu)**, **bag.Remove()** o **bag.Show()**. También se encuentra el main donde se llama a Init() pasando como argumentos **Bag()** y **Menu()**.

```
void Init(Bag bag, Menu menu)
{
    int num = menu.PrintBaseMenu();
    bool ingame = true;
    while (num != 5 && ingame)
    {
        if (num == 1)
            bag.Add(menu);
        else if (num == 2)
            bag.Remove();
        else if (num == 3)
            bag.Show();
        else
            ingame = false;
        num = menu.PrintBaseMenu();
    }
}
int main(int, char **)
{
    Init(Bag(), Menu());
}
```

Cabe destacar que se están alineando los objetos contenidos en el vector, pero no el vector en sí mismo. Para alinear realmente un vector en c++ se requiere del uso de funciones propias de la asignación dinámica de memoria (malloc allocator) para escribir un allocator propio que permita su alineación.

3. Documentación del ejercicio 3

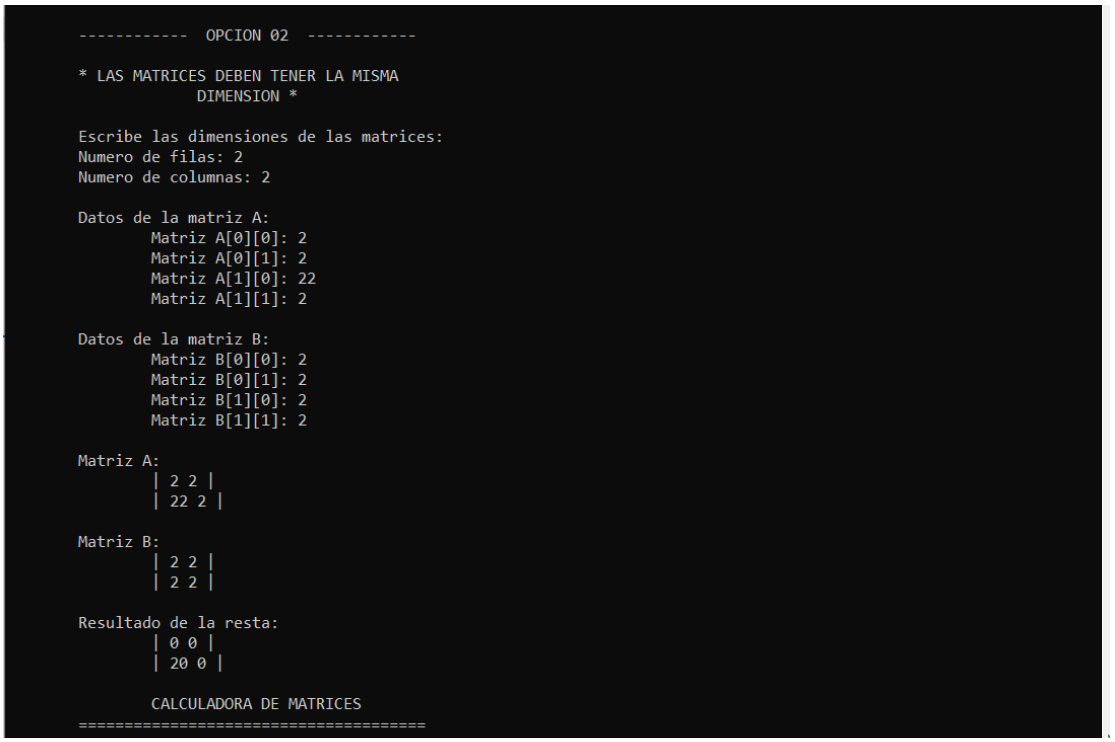
El ejercicio 3 consistía en crear una calculadora de matrices
Empezaremos explicando los detalles del programa a continuación:
Hemos creado un menú en el que solicitamos que el usuario escriba por pantalla la opción que quiere llevar a cabo. Es un menú que sólo acepta números enteros como valores correctos

```
void menu()
{
    cout << "" <<endl;
    cout << "" <<endl;
    cout << "\t\tCALCULADORA DE MATRICES" <<endl;
    cout << "\t====================================" <<endl;
    cout << "\t Elige una opcion:" <<endl;
    cout << "\t\t1. Suma de matrices" <<endl;
    cout << "\t\t2. Resta de matrices" <<endl;
    cout << "\t\t3. Multiplicacion de matrices" <<endl;
    cout << "\t\t4. Matriz traspuesta" <<endl;
    cout << "\t\t5. Determinante de una matriz" <<endl;
    cout << "\t\t6. Salir" <<endl;
    cout << "\t====================================" <<endl;
    cout << "" <<endl;
    cout << "\tOPCION:  ";
}
```

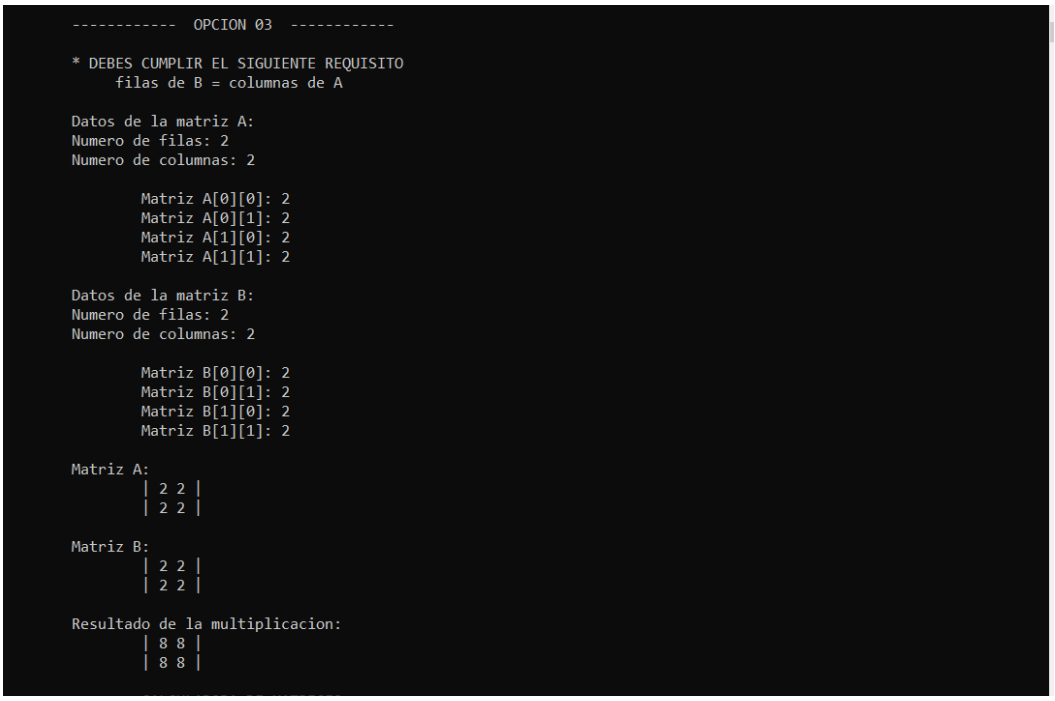
La primera opción es la suma de matrices. Tenemos que tener en cuenta que las matrices deben coincidir en dimensiones, por lo que pedimos al usuario que especifique el número de filas y columnas de ambas matrices en una sola opción, así evitamos problemas.

```
* LAS MATRICES DEBEN TENER LA MISMA
            DIMENSION *

Escribe las dimensiones de las matrices:
Numero de filas: 3
Numero de columnas: 3

Datos de la matriz A:
        Matriz A[0][0]: 1
        Matriz A[0][1]: 1
        Matriz A[0][2]: 1
        Matriz A[1][0]: 1
        Matriz A[1][1]: 1
        Matriz A[1][2]: 1
        Matriz A[2][0]: 1
        Matriz A[2][1]: 1
        Matriz A[2][2]: 1

Datos de la matriz B:
        Matriz B[0][0]: 1
        Matriz B[0][1]: 1
        Matriz B[0][2]: 1
        Matriz B[1][0]: 1
        Matriz B[1][1]: 11
        Matriz B[1][2]: 1
        Matriz B[2][0]: 1
        Matriz B[2][1]: 11
        Matriz B[2][2]: 1

Matriz A:
        | 1 1 1 |
        | 1 1 1 |
        | 1 1 1 |

Matriz B:
        | 1 1 1 |
        | 1 11 1 |
        | 1 11 1 |

Resultado de la suma:
        | 2 2 2 |
        | 2 12 2 |
        | 2 12 2 |
```

La opción de la resta es similar a la de la suma, lo único en lo que cambian es en el signo.



La multiplicación de matrices especifica que el número de columnas de la primera matriz debe coincidir con el número de filas de la segunda matriz. A partir de ahí, el usuario puede asignar por separado el número de filas y columnas de ambas matrices, pero si no coinciden con la condición antes explicada, arrojará un error, que explicará al usuario que no ha cumplido con la condición.



La matriz traspuesta es una matriz que intercambia filas, columnas y viceversa, por lo que pedimos al usuario que escriba por pantalla su matriz y nosotros imprimimos la traspuesta.

```
OPCION:  4

------------  OPCION 04  ------------

Escribe las dimensiones de la matriz:
Numero de filas: 3
Numero de columnas: 4

Datos de la matriz:
        Matriz[0][0]: 1
        Matriz[0][1]: 2
        Matriz[0][2]: 3
        Matriz[0][3]: 4
        Matriz[1][0]: 5
        Matriz[1][1]: 6
        Matriz[1][2]: 7
        Matriz[1][3]: 8
        Matriz[2][0]: 9
        Matriz[2][1]: 10
        Matriz[2][2]: 11
        Matriz[2][3]: 12

Matriz:
        | 1 2 3 4 |
        | 5 6 7 8 |
        | 9 10 11 12 |

Matriz traspuesta:
        | 1 5 9 |
        | 2 6 10 |
        | 3 7 11 |
        | 4 8 12 |
```

En cuanto al determinante, es imprescindible que la matriz sea cuadrática, por lo que pedimos al usuario que escriba el número de filas y columnas a la vez, después de eso, nosotros imprimimos el valor del determinante.

```
------------  OPCION 05  ------------

Escribe las dimensiones de la matriz:
Numero de filas y columnas: 3
Datos de la matriz:
        Matriz[0][0]: 1
        Matriz[0][1]: 2
        Matriz[0][2]: 3
        Matriz[1][0]: 4
        Matriz[1][1]: 5
        Matriz[1][2]: 6
        Matriz[2][0]: 7
        Matriz[2][1]: 8
        Matriz[2][2]: 9

Matriz:
        | 1 2 3 |
        | 4 5 6 |
        | 7 8 9 |

DETERMINANTE = -0
```

4. Documentación del ejercicio 4

Respecto al ejercicio 4 se ha realizado un programa el cual se encarga de altas y bajas de usuarios con unos atributos específicos, los cuales son los siguientes; nombre, apellidos y número de usuario (generado automáticamente).

Para la realización de esta aplicación en c++ se ha desarrollado una clase Usuario la cual tiene los atributos mencionados anteriormente, también cuenta con dos constructores, uno lleno y el otro vacío, un destructor.

```
// Constructor de usuario
Usuario::Usuario(string nom, string ape, int numUser)
{
    nombre = nom;
    apellidos = ape;
    numUsuario = numUser;
}
// Destructor de Usuario
Usuario::~Usuario() {}
```

Y por último en la clase Usuario, los siguientes tres métodos:

- insertarDatos(), el cual se encarga de introducir los atributos del usuario en el objeto.

```
void Usuario::insertarDatos(string n, string a, int id)
{
    this->nombre = n;
    this->apellidos = a;
    this->numUsuario = id;
}
```

- mostrarDatos(), el cual se encarga de exponer todos los usuarios y sus atributos.

```
void Usuario::mostrarDatos()
{
    cout << "Nombre: "<< nombre << endl;
    cout << "Apellidos: "<< apellidos << endl;
    cout << "Numero de usuario: "<< numUsuario << endl;
    cout << endl;
}
```

- getNombre(), que retorna el nombre del objeto correspondiente.

Mas tarde se crea una función de mostrar el menú, donde el usuario va a poder interactuar con él mediante el siguiente código de opciones, retornando la elección en una variable:

```
cout << "[MENU] Selecciona una de las opciones: " << endl;
    cout << "  [1] - Introducir un usuario" << endl;
    cout << "  [2] - Mostrar usuarios" << endl;
    cout << "  [3] - Eliminar un usuario" << endl;
    cout << "  [4] - Salir" << endl;
```

Este menú está incluido en un do while el cual hasta que no se proporcione una respuesta válida no se podrá seguir adelante y cuando se pulse la opción 4 , el programa finalizará.

En cuanto a la opción uno del menú, “Introducir un usuario”, lo que se hace primeramente es crear un objeto vacío llamado user, y luego se piden el nombre (irrepetible) y apellidos de este para meterlos mediante el método “insertarDatos” (el id de usuario se genera automáticamente dependiendo de cuantos objetos halla creados), y por último se introduce el objeto en un vector anteriormente creado.

Más adelante, la opción dos del menú, “Mostrar los usuarios”, lo que se realiza es un recorrido del vector donde se encuentran los objetos usuarios para poder mostrarlos uno a uno como se ve en el siguiente código:

```
for (int i = 0; i < usuarios.size(); i++)
 {
                // Mediante el método "mostrarDatos", se muestran por
pantalla todos los usuarios dados de alta
                usuarios[i].mostrarDatos();
 }
```

Y por último, la opción tres, “Destruir un usuario”, se encarga de pedir un nombre de usuario, el cual mediante un destructor de objetos, el usuario con dicho nombre desaparecerá como se muestra a continuación:

```
// Se recorre el vector "usuarios" através de un for
            for (int i = 0; i < usuarios.size(); i++)
            {
                // Si el nombre introducido coincide con uno de los
objetos creados, se elimina dicho objeto
                if (usuarios[i].getNombre() == nombre)
                {
                    // Eliminamos el elemento por posición de la lista
                    usuarios.erase(usuarios.begin() + i);

                    // Eliminamos el objeto por medio del DESTRUCTOR
                    usuarios[i].~Usuario();

                    cout << "El objeto usuario de nombre "<< nombre << "
ha sido destruido\n" << endl;
                    break;
                }
            }
```

5. Documentación del ejercicio 5

```
void adivina(){
    bool numc=false; //boleano para salir del bucle
    int cont=0; //contador de intentos
    int num=1+rand() % (101-1); //num aleatorios entre 1 y 100
    double x;
    cout<<"Adivina el numero generado aleatoriamente";
    cout<<"\nEscribe un numero entre el 1 y 100: ";
```

En la primera parte del método **adivina()** declaramos las variables:

numc: De tipo booleano inicializado a false, cuando se den las circunstancias deseadas lo usaremos para salir del bucle.

cont: De tipo integer para contar el número de intentos que ha hecho el jugador.

num: De tipo integer, en esta variable se guardará el número aleatorio generado.

x: De tipo double, para que si el jugador decide usar un número muy grande podamos recogerlo en una excepción.

Con estas variables declaradas empezamos el bucle:

```
while(numc==false){
        cont++; //intentos
        cin>>x;
        if (x==false){
            cout<<"\nSolo puedes introducir numeros\n";
            break;
        }else{
            if(x>100 || x<1){
                cout<<"\nRecuerda que el numero esta entre 1 y 100!, intentalo otra vez: ";
            }
            else if(x>num){
                cout<<"\nIncorrecto, el numero es menor, escribe otro numero: ";
            }
```

```
        else if(x<num){
            cout<<"\nIncorrecto, el numero es mayor, escribe otro numero: ";
        }
        else
            numc=true;
        }
    }
```

Mientras el booleano sea ‘false’ se seguirá repitiendo el bucle, usamos la variable cont para llevar la cuenta de los intentos del jugador.

Al principio del bucle se esperará que el jugador introduzca un número, si el jugador decide introducir algo que no sea un número el programa le informará de que sólo puede introducir números y se cerrará, si el número introducido es un número el programa continuará.

Si el número introducido es mayor o menor al número que el jugador debe adivinar se le informará para ayudarle, también se le recordará que el número que debe adivinar está entre 1 y 100, en el caso de que introduzca uno fuera de ese rango.

Cuando el número introducido coincida con el generado aleatoriamente la variable numc cambiará a true, de ese modo saldremos del bucle while.

```
if(numc==true){
        cout<<"\nCorrecto!";
        cout<<"\nLo has adivinado en " <<cont<< " intentos\n";
    }
```

Al salir del bucle se le informará al jugador de que ha acertado el número y se le mostrará el número de intentos en el que lo ha realizado.