PROYECTO #1

Videojuego: Space Invaders

Resumen

Nuestro primer Proyecto será el clásico videojuego “Space Invaders” (“Invasores Espaciales”, en español). Vamos a estar trabajando con la biblioteca “`raylib`”; la cual no es una biblioteca estándar de C/C++.

Concepto-1.1. Biblioteca

Una aplicación en C/C++ es un conjunto de atributos y funciones que juntos hacen cosas; por ejemplo, una función `main` corre las funciones necesarias para hacer “funcionar” un programa. Una biblioteca, también llamada librería, es un conjunto de código pre-escrito que también contiene atributos y funciones, pero estos no pueden ejecutarse, y solo tienen el objetivo de ser utilizados por otros programas —incluso por otras bibliotecas.

Fundamento-1.1. ¿Cómo funcionan las bibliotecas en C/C++?

Cree un archivo `main.cpp` y copie el siguiente código:

```
#include <iostream>


int sumar(int a, int b);


int main() {
	int primer_numero, segundo_numero, resultado;


	std::cout << "Ingresa un numero: ";
	std::cin >> primer_numero;
	std::cout << "Ingresa otro numero: ";
```

```
	std::cin >> segundo_numero;

	resultado = sumar(primer_numero, segundo_numero);

	std::cout << std::endl << "El resultado es: " << resultado << std::endl;

	return 0;
}

int sumar(int a, int b) {
	int respuesta;

	respuesta = a + b;

	return respuesta;
}
```

El `#` en `#include` es una directiva de preprocesador; es decir, `#` le va a decir a tu compilador: "copia todo el código que esté en la librería `iostream` justamente aquí arriba del resto del código. Siendo así, tu archivo después de la compilación quedaría así:

```
//toda la librería iostream
int sumar(int a, int b);

int main() {
//resto del código
```

Obvio es, que esto es un proceso interno y no se ve así reflejado en la pantalla del desarrollador.

`iostream` es una librería del estándar (ISO) de C/C++ que se utiliza para ingresar e imprimir datos. `i` de input (ingresar, en español) + `o` de output (sacar, en español) + `stream`, flujo en español = `iostream`.

`int sumar(int a, int b);` es una función **solamente declarada** (i.e. sin inicializar, sin llevar a cabo una operación para que funcione) de tipo entero `int`, llamada `sumar`; y, que recibe como parámetros un entero `a` y un entero `b`. Como toda sentencia termina en punto y coma `;`. Como no es una función vacía `void`; sino entera, sí o sí tendrá que retornar un valor una vez sea inicializada.

Ahora,

```
int main() {
	int primer_numero, segundo_numero, resultado;

	std::cout << "Ingresa un numero: ";
	std::cin >> primer_numero;
	std::cout << "Ingresa otro numero: ";
	std::cin >> segundo_numero;

	resultado = sumar(primer_numero, segundo_numero);

	std::cout << std::endl << "El resultado es: " << resultado << std::endl;

	return 0;
}
```

se trata de una función entera (i.e. va a retornar un valor numérico entero) llamada `main`, sin parámetros, y que ya se encuentra inicializada.

Hay tres variables enteras declaradas `int primer_numero, segundo_numero, resultado;`.

En `std::cout << "Ingresa un numero: ";`, `std` es la abreviatura de "standard" (estándar) y se refiere al espacio de nombres estándar, que contiene la biblioteca estándar del lenguaje C/C++. Este espacio de nombres agrupa tipos de datos, funciones, clases y otros símbolos predefinidos para tareas comunes, como el uso de la entrada/salida (`std::cin`, `std::cout`), contenedores (`std::vector`), algoritmos (`std::sort`), etc. Y `cout` es para mostrar un mensaje por consola. `cin` es para ingresar un dato por consola.

`resultado = sumar(primer_numero, segundo_numero);` aquí a la variable resultado se le asigna (con el símbolo `=`) la función declarada previamente `sumar` que contiene los parámetros `primer_numero` y `segundo_numero` con los que el usuario interactúa por consola. `primer_numero` y `segundo_numero` van a remplazar a los valores `a` y `b` que fueron declarados inicialmente en la función `sumar(int a, int b`, porque ahora si se le está dando uso a la función.

Ahora, ¿cómo el `main()` esta utilizando la función sin siquiera saber cómo "funciona" la función? La respuesta es simple, en

```
int sumar(int a, int b) {
```

```
	int respuesta;

	respuesta = a + b;

	return respuesta;
}
```

por fin se ha inicializado la función definiendo su comportamiento. Se creó una variable entera `respuesta;` que almacenaba la operación binaria suma de `a` y `b`, cuales fueron posteriormente remplazados por los parámetros `primer_numero` y `segundo_numero` en la función `main()`. Queda sobreentendido que al ser una función no vacía iba a retornar un número —`return respuesta;` en este caso.

Ejecuté su programa e interactué con él. Puede tratar de escribir su propia función para realizar una búsqueda, aplicar un método numérico… lo que deseé. Tal vez una buena opción sería aprender a codificar y decodificar creando su propio estándar de codificación (NIVEL ING. DE SOFTWARE DE SILICON VALLEY —no es bait).

¡Felicidades!, con el código anterior acaba de aprender lo más básico de cualquier lenguaje de programación. Ahora sí, avancemos unos "pasitos".

Un comportamiento que todo ingeniero de software debería tener, es el uso de una buena ARQUITECTURA DE SOFTWARE, ¿por qué?, antes de entender las razones juguemos con el código anterior añadiéndole tres nuevas funciones `restar`, `multiplicar` y `dividir`. El código resultante es:

```
#include <iostream>

int sumar(int a, int b);
int restar(int a, int b);
int multiplicar(int a, int b);
int dividir(int a, int b);

int main() {
	int primer_numero, segundo_numero, resultado, resultadoResta, resultadoMultiplicacion, resultadoDivision;

	std::cout << "Ingresa un numero: ";
	std::cin >> primer_numero;
	std::cout << "Ingresa otro numero: ";
```

```
	std::cin >> segundo_numero;


	resultado = sumar(primer_numero, segundo_numero);
	resultadoResta = restar(primer_numero, segundo_numero);
	resultadoMultiplicacion = multiplicar(primer_numero, segundo_numero);
	resultadoDivision = dividir(primer_numero, segundo_numero);


	std::cout << std::endl << "El resultado de la suma es: " << resultado << std::endl;
	std::cout << std::endl << "El resultado de la suma es: " << resultadoResta << std::endl;
	std::cout << std::endl << "El resultado de la suma es: " << resultadoMultiplicacion << std::endl;
	std::cout << std::endl << "El resultado de la suma es: " << resultadoDivision << std::endl;


	return 0;
}


int sumar(int a, int b) {
	int respuesta;


	respuesta = a + b;


	return respuesta;
}


int restar(int a, int b) {
	int respuesta;


	respuesta = a - b;
```

```
	return respuesta;
}

int multiplicar(int a, int b) {
	int respuesta;

	respuesta = a * b;

	return respuesta;
}

int dividir(int a, int b) {
	int respuesta;

	respuesta = a / b;

	return respuesta;
}
```

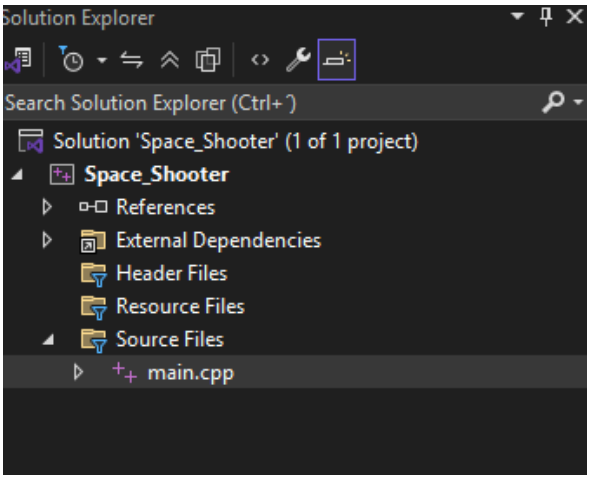
Aquí hay una imagen más apreciable de cómo debe ir yendo su trabajo:

main.cpp*

Space_Shooter (Global Scope)

```cpp
#include <iostream>

int sumar(int a, int b);
int restar(int a, int b);
int multiplicar(int a, int b);
int dividir(int a, int b);

int main() {
    int primer_numero, segundo_numero, resultado, resultadoResta, resultadoMultiplicacion, resultadoDivision;

    std::cout << "Ingresa un numero: ";
    std::cin >> primer_numero;
    std::cout << "Ingresa otro numero: ";
    std::cin >> segundo_numero;

    resultado = sumar(primer_numero, segundo_numero);
    resultadoResta = restar(primer_numero, segundo_numero);
    resultadoMultiplicacion = multiplicar(primer_numero, segundo_numero);
    resultadoDivision = dividir(primer_numero, segundo_numero);

    std::cout << std::endl << "El resultado de la suma es: " << resultado << std::endl;
    std::cout << std::endl << "El resultado de la suma es: " << resultadoResta << std::endl;
    std::cout << std::endl << "El resultado de la suma es: " << resultadoMultiplicacion << std::endl;
    std::cout << std::endl << "El resultado de la suma es: " << resultadoDivision << std::endl;

    return 0;
}

int sumar(int a, int b) {
    int respuesta;

    respuesta = a + b;

    return respuesta;
}

int restar(int a, int b) {
    int respuesta;

    respuesta = a - b;

    return respuesta;
}

int multiplicar(int a, int b) {
    int respuesta;

    respuesta = a * b;

    return respuesta;
}

int dividir(int a, int b) {
    int respuesta;

    respuesta = a / b;

    return respuesta;
}
```

Observe bien que en los `std::cout << std::endl`… por copiar se dejo el mismo mensaje de suma, cambíelo por resta, multiplicación y división (i.e. `std::cout << std::endl << "El resultado de la resta`…).



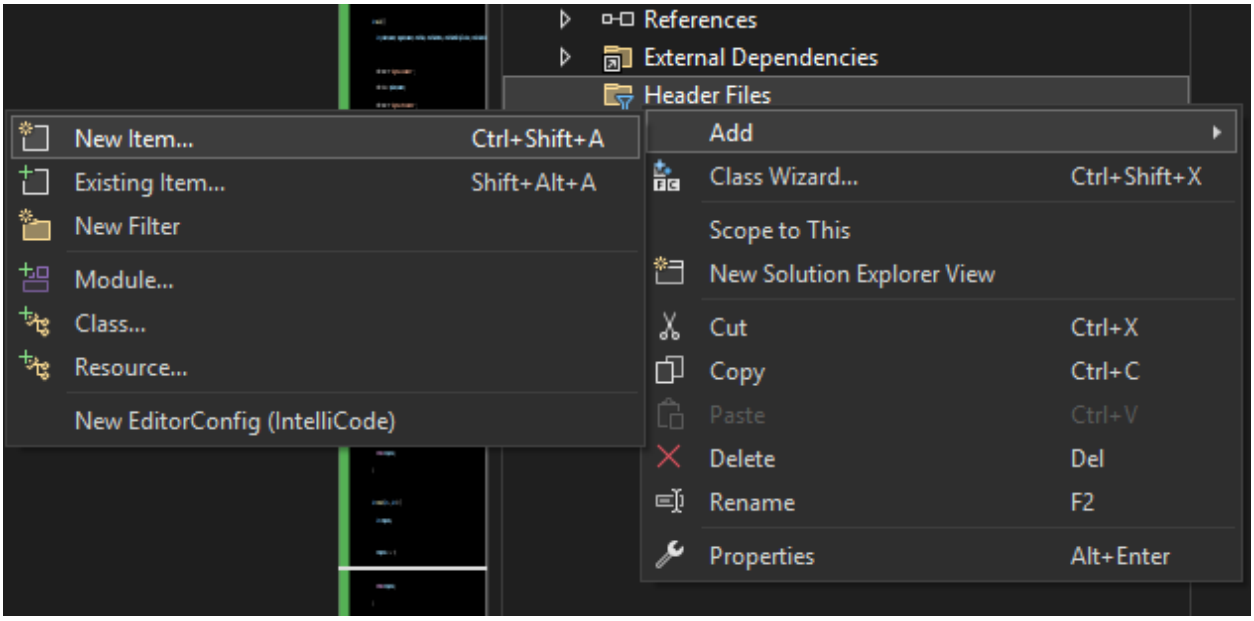
Como es de apreciar, su código se ha vuelto una monstruosidad. ¿Qué puede hacer para aliviar tanta carga en un solo archivo? Trabajar por **MÓDULOS**.

Concepto-1.2. Módulos

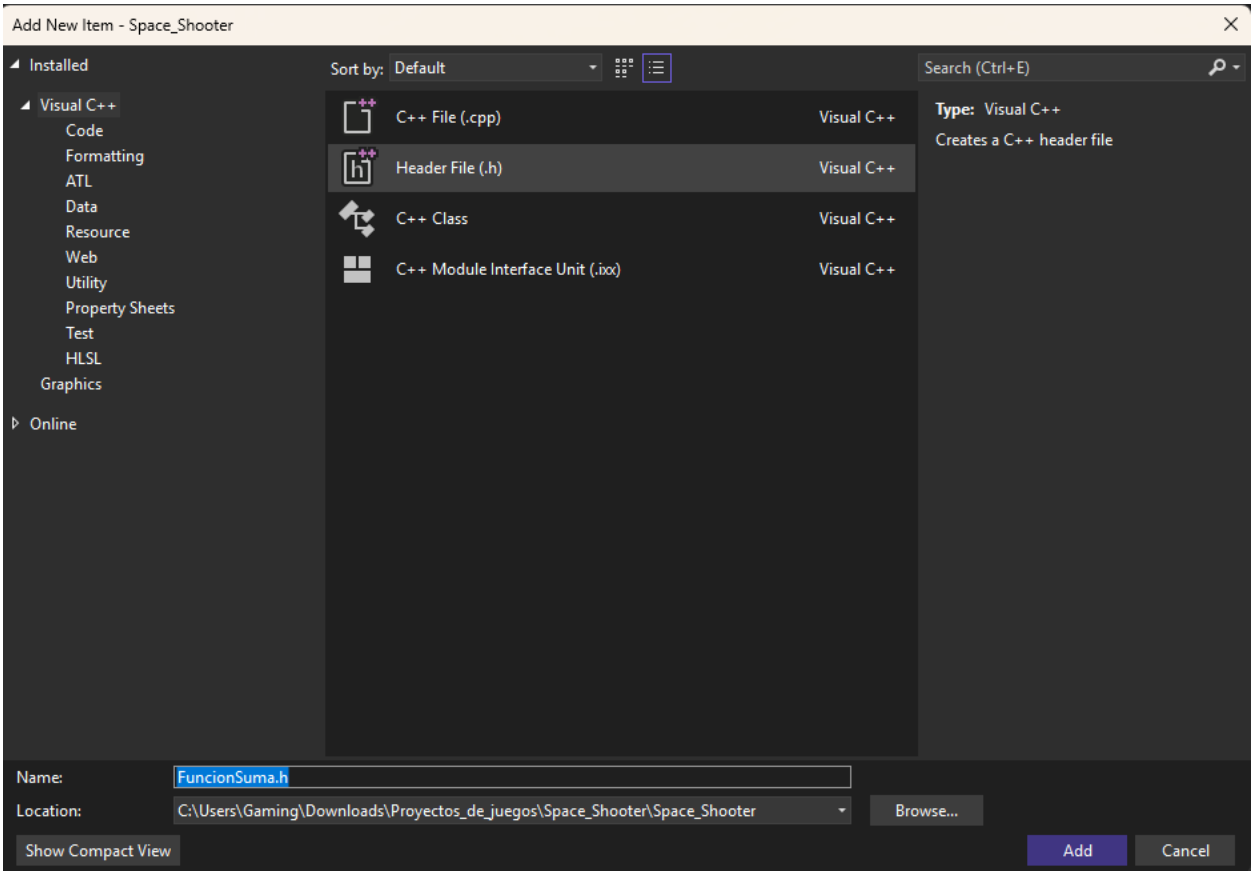
Un módulo es un conjunto de código organizado para la reutilización y la compilación. Es una forma más eficiente de organizar el código.

Toda la monstruosidad se va a separar por módulos de la siguiente manera:

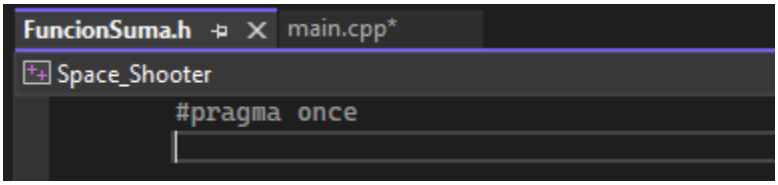
1) Dar clic derecho en “Header Files”, posicionarse sobre “add” y luego añadir un “New item…” (nuevo ítem, en español)

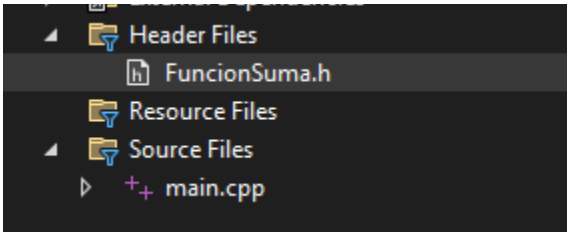


2) Crea su archivo `FuncionSuma.h`. Y le da clic a “Add” para añadirlo a su proyecto.

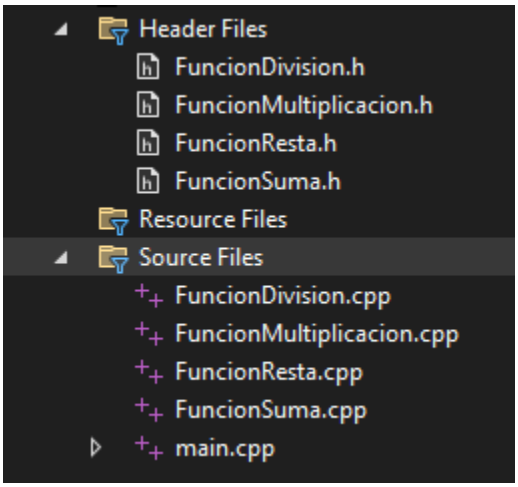


3) Ahora va a tener un archivo sobre el filtro “Header Files”.





4) Antes de la explicación, cree otros archivos de extensión .h: `FuncionResta.h`, `FuncionMultiplicacion.h`, `FuncionDivision.h`. También cree archivos de extensión .cpp con el mismo nombre para cada función: `FuncionSuma.cpp`, `FuncionResta.cpp`, `FuncionMultiplicacion.cpp`, `FuncionDivision.cpp`, sobre el filtro “Source Files”. La estructura de sus archivos debe quedar algo así:



Los archivos del lenguaje tienen una extensión .C; mientras que, lo archivos C++ tienen una extensión .Cpp. Para el lenguaje C, existen archivos headers (cabeceras, en español) de extensión .h; mientas que, para C++ tenemos archivos cabeceras de extensión .hpp.

Concepto-1.2. Archivo cabecera

Un archivo cabecera (.h para C) o (.hpp para C++) contiene declaraciones que otros archivos del programa necesitan para poder usar funciones, tipos, constantes (conocidas también como macros) o clases que están implementadas en otro archivo.

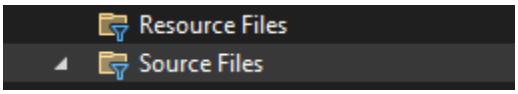
Es una regla simple: Un archivo .h dice “esto existe”. El archivo .c o .cpp dice “así funciona”. Declarar e inicializar. Es posible que algún guía académico le haya hecho hincapié en esa diferencia.

Desde el punto de vista de un ingeniero de software, los archivos cabecera representan:

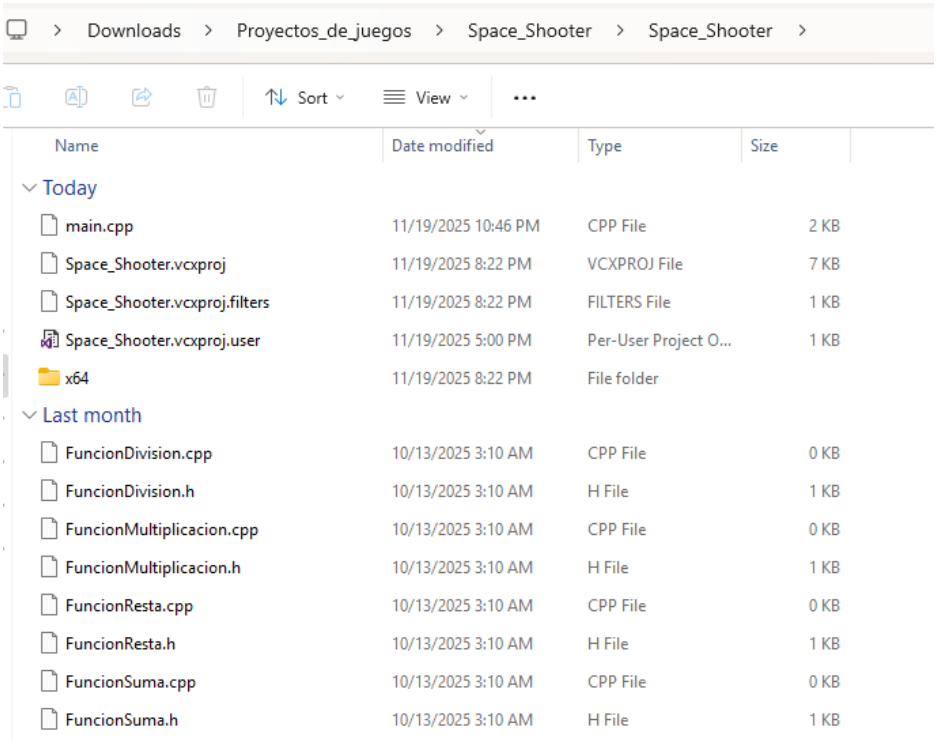
- Separación INTERFAZ vs IMPLEMENTACIÓN: Esto es un principio clásico de la arquitectura modular.
    - Header (.h) = INTERFAZ: Lo que se ofrece al usuario del módulo (funciones, tipos, clases).
    - Source (.c/.cpp) = IMPLEMENTACIÓN: Cómo funciona internamente.
- Reutilización de código: Otros programadores pueden usar las funciones sin conocer la implementación.
- Compilación modular: Los compiladores pueden recompilar solo los archivos modificados, acelerando el build (construcción, en español).
- Encapsulamiento: En C++ especialmente, los .h permiten esconder detalles internos.

Sí se puede usar .h con .cpp sin problema, pero no es buena idea (ni común) usar .hpp con .c, porque C no entiende varios elementos exclusivos de C++. Además, estos no son los únicos archivos cabeceras que existen, también están los .imp, .tcc, .tpp y demás.

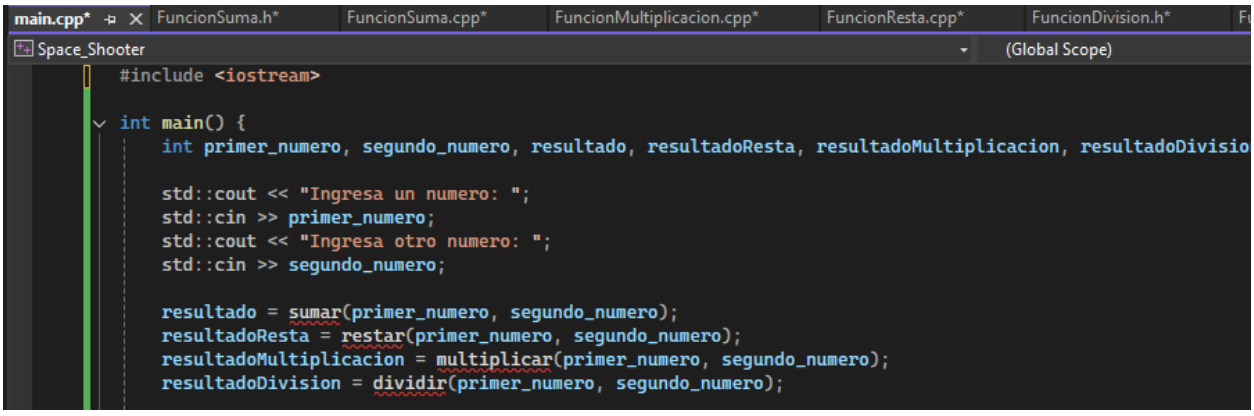
Sobre los filtros,

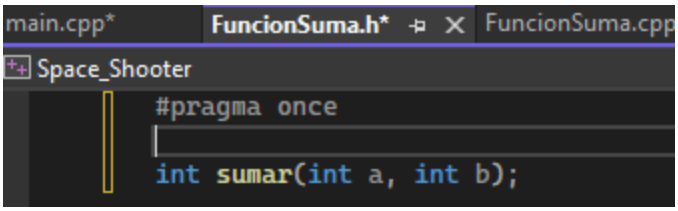


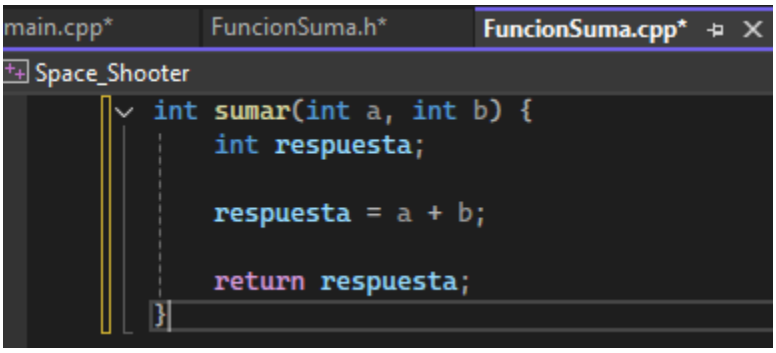
tiene que saber que no se trata de las carpetas reales de sus proyectos, son solo carpetas imaginarias que proporciona el entorno de desarrollo integrado para una mejor visualización de la estructura del código. Puede comprobarlo yendo a la carpeta original en su explorador de archivos.



Para aplicar la modularidad, va a cortar la declaración de cada función del `main.cpp` y la va a pegar sobre su respectivo .h. Al igual que va a hacer lo mismo con cada función inicializada sobre su respectivo .cpp. Quedando así:







La directiva `#pragma once` es una directiva del preprocesador utilizada en archivos de encabezado de C y C++. Su propósito es asegurar que el archivo de encabezado actual se incluya solo una vez durante una única compilación. Esto previene problemas como los errores de redefinición que pueden ocurrir cuando el mismo archivo de encabezado se incluye múltiples veces a través de diferentes rutas dentro de un proyecto.

A veces es mejor utilizar los "include guards" (guardianes de inclusión, en español) envés de la directiva `#pragma once`.

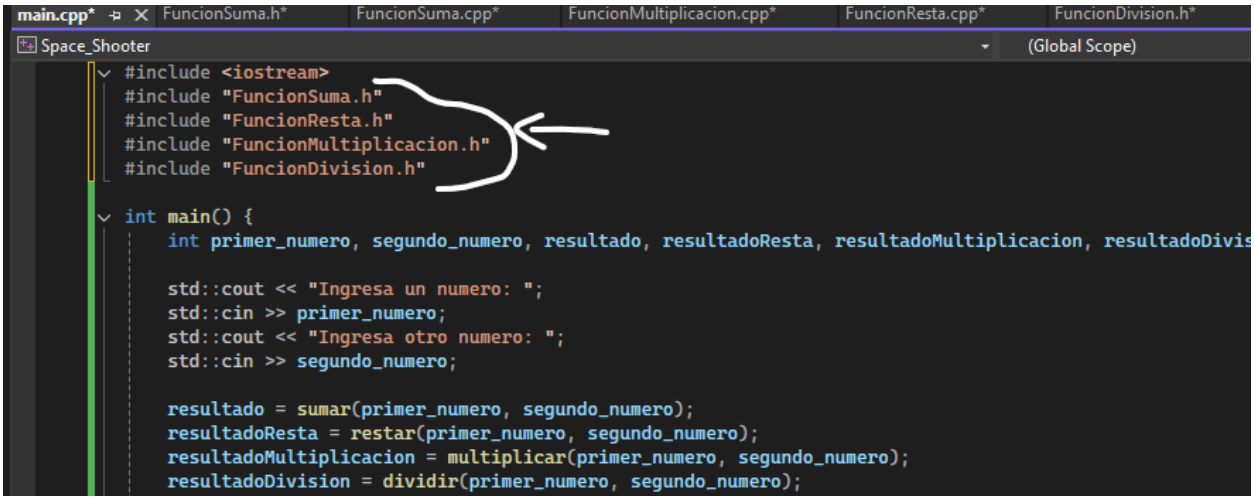
Aspecto	Include Guards	`#pragma once`
¿Qué son?	Técnica tradicional usando macros del preprocesador para evitar incluir un header más de una vez.	Directiva específica del compilador que indica incluir el archivo una sola vez sin usar macros.
¿Para qué sirven?	Evitar redefiniciones múltiples causada por inclusiones repetidas del mismo archivo .h.	Hacer lo mismo, pero de manera más simple, rápida y menos propensa a errores.
¿Cuándo se utilizan?	Cuando necesitas **compatibilidad total**, incluyendo compiladores muy antiguos o de C puro.	Cuando usas compiladores modernos (GCC, Clang, MSVC). Prácticamente estándar de facto hoy.
Ventajas principales	100% portable. Funciona en cualquier compilador C/C++.	Sintaxis simple, más limpio y más rápido (el compilador no reabre el archivo múltiples veces).
Desventajas principales	Verboso, propenso a errores si el macro no coincide o se duplica.	No está en el estándar C/C++; depende del compilador (pero todos los modernos lo soportan).
Sintaxis	Usa `#ifndef`, `#define`, `#endif`.	Una sola línea: `#pragma once`.
Ejemplo de uso	Ver abajo.	Ver abajo.

Aquí un ejemplo (solo para demostración) del uso de "include guards":

```
// archivo: MiClase.h

#ifndef MICLASE_H    // Se evalúa si el macro NO está definido
#define MICLASE_H    // Define el macro

class MiClase {
public:
    void saludar();
};
#endif // MICLASE_H
```

Continuando con el código actual, una vez trate de compilar y correr el main, se va a percatar que ya no se ejecutará. Nos falta enlazar/relacionar toda la estructura.

Lo primero que va a hacer es a invocar las bibliotecas que ha creado (headers .h) de la siguiente manera:



Ahora nuevamente podrá ejecutar su código sin problema alguno.

El código actual en el main,

```
#include <iostream>
#include "FuncionSuma.h"
#include "FuncionResta.h"
#include "FuncionMultiplicacion.h"
#include "FuncionDivision.h"
```

```
int main() {

	int primer_numero, segundo_numero, resultado, resultadoResta, resultadoMultiplicacion, resultadoDivision;


	std::cout << "Ingresa un numero: ";

	std::cin >> primer_numero;

	std::cout << "Ingresa otro numero: ";

	std::cin >> segundo_numero;


	resultado = sumar(primer_numero, segundo_numero);

	resultadoResta = restar(primer_numero, segundo_numero);

	resultadoMultiplicacion = multiplicar(primer_numero, segundo_numero);

	resultadoDivision = dividir(primer_numero, segundo_numero);


	std::cout << std::endl << "El resultado de la suma es: " << resultado << std::endl;

	std::cout << std::endl << "El resultado de la suma es: " << resultadoResta << std::endl;

	std::cout << std::endl << "El resultado de la suma es: " << resultadoMultiplicacion << std::endl;

	std::cout << std::endl << "El resultado de la suma es: " << resultadoDivision << std::endl;


	return 0;

}
```

aún puede optimizarme mediante la implementación de un **PATRÓN "DELEGATION"** ("delegación", en español), ¿cómo?, delegando a una nueva función la funcionalidad de pedir datos de entrada por consola al usuario envés de hacerlo directamente en el main. Para esto podemos utilizar referenciación —símil a punteros de C.

```cpp
#pragma once

int sumar(int a, int b);
void pedirNumero(int& numero);
```

```cpp
#include <iostream>

int sumar(int a, int b) {
    int respuesta;

    respuesta = a + b;

    return respuesta;
}

void pedirNumero(int& numero) {
    std::cout << "Ingresa un numero: ";
    std::cin >> numero;
}
```

```cpp
#include <iostream>
#include "FuncionSuma.h"
#include "FuncionResta.h"
#include "FuncionMultiplicacion.h"
#include "FuncionDivision.h"

int main() {
    int primer_numero, segundo_numero, resultado, resultadoResta, resultadoMultiplicacion, resultadoDivision;

    pedirNumero(primer_numero);
    pedirNumero(segundo_numero);

    resultado = sumar(primer_numero, segundo_numero);
    resultadoResta = restar(primer_numero, segundo_numero);
    resultadoMultiplicacion = multiplicar(primer_numero, segundo_numero);
    resultadoDivision = dividir(primer_numero, segundo_numero);
```

Lo va a aplicar solo con **UN** .h con su respectivo .cpp, porque si no estaría referenciando todas las bibliotecas/librerías y no estaría aplicando correctamente la referenciación. Ni siquiera se ejecutaría.

Ya puede nuevamente compilar y ejecutar su programa tranquilamente habiendo aplicado arquitectura de software.

¡Felicitaciones si ha llegado hasta aquí! Ahora, conociendo todos estos conocimientos básicos podemos generar nuestro primer proyecto. A medida de su realización también podrá profundizar en otros conceptos como las clases, referenciación, entre otros.

Space Invaders

g