Unidad 5: Modularidad

¡Hoy vamos a abordar el fascinante mundo de la **modularidad** en programación. Este concepto no solo mejora la organización de nuestro código, sino que también nos permite construir programas más claros, reutilizables y fáciles de mantener. En esta clase, exploraremos tres aspectos fundamentales: declaración y uso de módulos, paso de parámetros, e implementación. Además, comprenderemos cómo y por qué usar funciones, junto con detalles sobre el paso de parámetros por valor y por referencia, incluyendo variables, arreglos, estructuras y arreglos de estructuras como argumentos.

¿Por qué usar funciones?

Las funciones son bloques de código reutilizables que nos permiten resolver problemas específicos de manera organizada y eficiente. Aquí algunas razones clave para usarlas:

1. **Reutilización del código**: Escribimos una función una vez y la usamos múltiples veces.
2. **Facilidad de depuración**: Si algo falla, solo necesitamos revisar el código de la función específica.
3. **Claridad y mantenimiento**: Dividir el programa en partes más pequeñas lo hace más comprensible y manejable.
4. **Colaboración**: Los equipos pueden trabajar en diferentes funciones de manera simultánea.

5.1 Declaración y uso de módulos

Definición de funciones en C++

Una función debe ser **declarada** antes de ser utilizada. Una declaración o prototipo le indica al compilador el nombre de la función, los parámetros que espera y el tipo de valor que regresa.

Ejemplo de declaración:

```
int suma(int a, int b);
```

Uso de funciones

Las funciones deben ser **definidas** (se escribe el cuerpo con las instrucciones) y luego **llamadas** desde el programa principal o desde otras funciones.

Ejemplo:

```
#include <iostream>
using namespace std;
// Declaración de la función (prototipo)
int suma(int a, int b);
// Función principal
int main() {
        int x = 5, y = 10;
        cout << "La suma es: " << suma(x, y) << endl; // Llamada
        return 0;
}
// Definición
int suma(int a, int b) {
        return a + b;
 }
```

5.2 Paso de parámetros o argumentos

Cuando una función recibe valores, estos se denominan **parámetros** o **argumentos**. En C++, existen dos formas principales de pasarlos: **por valor** y **por referencia**.

Paso por valor

En esta modalidad, se copia el valor del argumento en una nueva variable dentro de la función. Esto significa que los cambios realizados en la función no afectan la variable original.

Ejemplo:

```
void incrementar(int n) {
        n++;
}
int main() {
        int numero = 5;
        incrementar(numero);
        cout << "Número después de incrementar: " << numero << endl; // Sigue siendo 5
        return 0;
}
```

Paso por referencia

En este caso, se pasa una referencia a la variable original, lo que permite modificar su contenido desde la función.

Ejemplo:

```
void incrementar(int &n) {
n++; }
int main() {
        int numero = 5;
        incrementar(numero);
        cout << "Número después de incrementar: " << numero << endl; // Ahora es 6
        return 0;
}
```

5.3 Implementación de funciones con diferentes tipos de parámetros

Las funciones pueden recibir diferentes tipos de parámetros, como **variables**, **arreglos**, **estructuras** y **arreglos de estructuras**.

Variables como parámetros

Son los ejemplos más comunes, como ya vimos en el caso de paso por valor y por referencia.

Arreglos como parámetros

Cuando pasamos un arreglo, realmente estamos pasando una referencia a su dirección de memoria. Esto significa que las funciones pueden modificar directamente su contenido.

Ejemplo:

```
void imprimirArreglo(int arr[], int tam) {
        for (int i = 0; i < tam; i++) {
                cout << arr[i] << " ";
        }
        cout << endl;
}
int main() {
        int numeros[] = {1, 2, 3, 4, 5};
        imprimirArreglo(numeros, 5);
        return 0;
}
```

Estructuras como parámetros

Las estructuras permiten agrupar varios datos en una sola entidad. Podemos pasarlas por valor o por referencia.

Ejemplo:

```
struct Persona {
    string nombre;
    int edad;
};
void mostrarPersona(Persona p) {
    cout << "Nombre: " << p.nombre << ", Edad: " << p.edad << endl;
}
int main() {
    Persona p1 = {"Juan", 25};
    mostrarPersona(p1);
    return 0;
}
```

Arreglos de estructuras como parámetros

Podemos pasar arreglos de estructuras para trabajar con conjuntos de datos más complejos.

Ejemplo:

```
void mostrarPersonas(Persona personas[], int tam) {
    for (int i = 0; i < tam; i++) {
        cout << "Nombre: " << personas[i].nombre << ", Edad: " << personas[i].edad << endl;
    }
}
int main() {
    Persona grupo[] = {{"Ana", 30}, {"Luis", 28}, {"María", 22}};
    mostrarPersonas(grupo, 3);
    return 0;
}
```

Conclusión

El uso de funciones nos permite estructurar programas complejos en bloques manejables. Entender cómo declarar, implementar y utilizar funciones con diferentes tipos de parámetros es fundamental para desarrollar aplicaciones eficientes. ¡Practiquemos con ejemplos para dominar este tema!

En algunos ejemplos, a veces se pone un prototipo de una función antes de la función main y después de la función main se coloca la implementación o definición de la función y, en otras ocasiones, se pone la definición de la función antes de la función main, sin prototipo.

¿Cuál es la forma correcta o qué diferencias, ventajas y desventajas hay de cada forma de hacerlo?

En C++, tanto declarar un prototipo antes del main y definir la función después, como definir la función directamente antes del main son prácticas válidas. Sin embargo, cada enfoque tiene sus ventajas y desventajas dependiendo del contexto. Vamos a analizar cada caso:

1. Prototipo antes del main **y definición después**

En este método, se declara el **prototipo** de la función antes del main para que el compilador la reconozca, y la implementación o definición completa de la función se coloca después del main.

Ejemplo:

```
#include <iostream>
using namespace std;
// Prototipo de la función
int suma(int a, int b);
int main() {
        int x = 5, y = 10;
        cout << "La suma es: " << suma(x, y) << endl;
        return 0;
}
 // Definición de la función
int suma(int a, int b) {
        return a + b;
}
```

Ventajas:

1. **Claridad del flujo principal:** El main aparece al principio del programa, lo que facilita identificar el flujo lógico principal.
2. **Estructuración para programas grandes:** En proyectos grandes, las definiciones de funciones suelen ir en archivos separados. Los prototipos en un encabezado (.h) facilitan su inclusión y reutilización.
3. **Facilidad de lectura:** Al listar todos los prototipos antes del main, se puede entender rápidamente qué funciones estarán disponibles.

Desventajas:

1. **Complejidad en programas pequeños:** Para programas simples, tener prototipos y definiciones puede parecer redundante.

2. **Mayor fragmentación:** Las definiciones quedan dispersas, lo que podría dificultar el seguimiento de las funciones en programas pequeños.

---

2. Definición completa antes del main

En este método, se escribe toda la definición de la función antes del main, eliminando la necesidad de un prototipo.

Ejemplo:

```
#include <iostream>
using namespace std;
// Definición de la función
 int suma(int a, int b) {
        return a + b;
}
int main() {
        int x = 5, y = 10;
        cout << "La suma es: " << suma(x, y) << endl;
        return 0;
}
```

Ventajas:

1. **Simplicidad en programas pequeños:** En aplicaciones sencillas, evita la redundancia de tener prototipos.
2. **Flujo lineal:** El programa fluye de forma natural desde las definiciones hasta el main.

Desventajas:

1. **Desorden en programas grandes:** Si hay muchas funciones, el main puede quedar desplazado, haciendo que el flujo principal sea difícil de encontrar.
2. **Menor compatibilidad con proyectos grandes:** En programas modulares, donde las definiciones suelen ir en archivos separados, este enfoque no es práctico.

¿Cuál es la forma correcta?

No existe una única "forma correcta". La elección depende del tamaño del programa y de las buenas prácticas que sigas:

1. **Para programas pequeños o de aprendizaje:** Puedes definir las funciones directamente antes del main sin usar prototipos. Esto simplifica el código y evita confusiones.

2. **Para programas medianos o grandes:** Es mejor usar prototipos antes del main y mover las definiciones después. Este enfoque facilita la lectura del flujo principal y permite dividir el programa en módulos o archivos separados.

Recomendaciones prácticas

- **Proyectos educativos y básicos:** Define las funciones antes del main para mantener la simplicidad.
- **Proyectos reales o grandes:** Usa prototipos antes del main y organiza tus definiciones en archivos separados (.cpp para las definiciones y .h para los prototipos). Esto es estándar en la mayoría de los proyectos C++.

Finalmente, cualquiera que sea el enfoque que elijas, asegúrate de que tu código sea legible y esté bien documentado para facilitar el trabajo en equipo y la revisión futura.

Creación de librerías:

Crear tus propias librerías en C++ es un excelente paso para organizar tu código y reutilizar funciones en varios programas. Aquí te explicaré el proceso completo:

---

1. Entendiendo las librerías en C++

Una librería en C++ consiste en:

- **Un archivo de encabezado (.h):** Contiene las declaraciones de funciones (prototipos) y, opcionalmente, definiciones de constantes, macros y clases.
- **Un archivo de implementación (.cpp):** Contiene las definiciones completas de las funciones declaradas en el archivo de encabezado.

---

2. Estructura básica de una librería

Supongamos que queremos crear una librería llamada mimath con funciones matemáticas personalizadas.

Archivo de encabezado (mimath.h**):**

El encabezado contiene los **prototipos** de las funciones y una **directiva de inclusión única** para evitar múltiples inclusiones accidentales.

```
#ifndef MIMATH_H // Protección contra inclusiones múltiples
#define MIMATH_H
// Declaraciones de funciones
int suma(int a, int b);
int resta(int a, int b);
int factorial(int n);
#endif // MIMATH_H
```

Archivo de implementación (mimath.cpp**):**

Aquí se escriben las **definiciones** de las funciones declaradas en el encabezado.

```
#include "mimath.h" // Incluye el archivo de encabezado
int suma(int a, int b) {
	return a + b;
}
int resta(int a, int b) {
	return a - b;
}
int factorial(int n) {
	if (n <= 1)
		return 1;
	return n * factorial(n - 1);
}
```

---

3. Usando la librería en tu programa

Para usar tu librería, solo necesitas incluir el archivo de encabezado en tu programa principal y asegurarte de compilar y enlazar el archivo de implementación.

Ejemplo (main.cpp**):**

```
#include <iostream>
#include "mimath.h" // Incluimos la librería personalizada
using namespace std;
int main() {
	int x = 5, y = 3;
	cout << "Suma: " << suma(x, y) << endl;
	cout << "Resta: " << resta(x, y) << endl;
	cout << "Factorial de 5: " << factorial(5) << endl;
	return 0;
}
```

---

4. Compilación y enlace

Cuando compiles el programa, necesitas incluir tanto el archivo principal (main.cpp) como el archivo de implementación de la librería (mimath.cpp).

En Dev-C++ (manual):

1. **Abrir un nuevo proyecto de tipo consola.**
2. Añadir los tres archivos (mimath.h, mimath.cpp, main.cpp) al proyecto.
3. Compilar y ejecutar el proyecto.

Desde la terminal:

Si usas una terminal, puedes compilar con el siguiente comando:
g++ main.cpp mimath.cpp -o programa
Esto genera un ejecutable llamado programa.

---

5. Hacer que funcione como una librería global

Para poder incluirla con #include <mimath> sin usar comillas, debes colocar el archivo mimath.h en una ubicación reconocida por el compilador como un directorio de inclusión global.

En Windows (Dev-C++ o MinGW):

1. Copia mimath.h a la carpeta include dentro del directorio donde está instalado Dev-C++ o MinGW (por ejemplo, C:\Dev-Cpp\include).
2. Si deseas compilarlo como una librería estática (.a) o dinámica (.dll), sigue los pasos del próximo punto.

---

6. Opcional: Crear una librería estática o dinámica

Librería estática (.a):

1. Compila solo el archivo mimath.cpp en formato de objeto:

g++ -c mimath.cpp -o mimath.o

2. Crea la librería estática:

ar rcs libmimath.a mimath.o

3. Usa la librería al compilar tu programa:

g++ main.cpp -L. -lmimath -o programa

Aquí, -L. indica que la librería está en el directorio actual y -lmimath enlaza con libmimath.a.

Librería dinámica (.dll o .so):

1. Genera un archivo de objeto compartido:

g++ -shared -o libmimath.dll mimath.cpp

2. Asegúrate de que el programa pueda encontrar la librería al ejecutarse.

Con estos pasos, puedes crear, organizar y reutilizar tus propias librerías en C++. ¡Es una excelente forma de llevar tus habilidades al siguiente nivel! 😊

Si prefieres crear y usar tus propias librerías estáticas o dinámicas directamente desde **Embarcadero Dev-C++** sin necesidad de usar comandos manuales en la terminal, aquí te explico cómo hacerlo paso a paso.

Crear una Librería Estática (.a)

Las librerías estáticas contienen todas las definiciones necesarias para ser enlazadas con tus programas.

Pasos en Dev-C++:

1. **Crear un proyecto de tipo "Static Library":**

a) Abre Dev-C++ y selecciona **File > New > Project**.
b) Elige la opción **Static Library** y asigna un nombre al proyecto, por ejemplo, mimath.
c) Guarda el proyecto en una carpeta específica (por ejemplo, C:\MisLibrerias).

2. **Añadir las funciones al proyecto:**

a) Crea un nuevo archivo llamado mimath.cpp y añade tus definiciones de funciones.
b) Asegúrate de crear también un archivo de encabezado (mimath.h) con los prototipos de las funciones.

Archivo mimath.h**:**

```
#ifndef MIMATH_H
#define MIMATH_H
int suma(int a, int b);
int resta(int a, int b);
int factorial(int n);
#endif
```

Archivo mimath.cpp**:**

```
#include "mimath.h"
int suma(int a, int b) {
        return a + b;
}
int resta(int a, int b) {
        return a - b;
}
int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}
```

3. **Compilar la librería:**

- Haz clic en **Execute > Compile**. Dev-C++ generará un archivo con extensión .a, por ejemplo, libmimath.a, que es la librería estática.
- Este archivo estará en la carpeta de salida del proyecto (normalmente C:\MisLibrerias).

4. **Usar la librería en otro proyecto:**

- Crea un nuevo proyecto (por ejemplo, main.cpp) y asegúrate de incluir el encabezado mimath.h.
- Ve a **Project > Project Options > Parameters > Linker** y haz clic en **Add Object/Library** para agregar el archivo libmimath.a.
- Incluye también la ruta donde se encuentra el archivo de encabezado (mimath.h) en **Project Options > Parameters > C Includes**.

Archivo main.cpp**:**

```
#include <iostream>
#include "mimath.h"
using namespace std;
int main() {
	cout << "Suma: " << suma(3, 7) << endl;
	cout << "Factorial: " << factorial(5) << endl;
	return 0;
}
```

5. **Compila y ejecuta:** Ahora tu programa usará las funciones de la librería estática.

Crear una Librería Dinámica (.dll)

Las librerías dinámicas son compartidas y cargadas en tiempo de ejecución.

Pasos en Dev-C++:

1. **Crear un proyecto de tipo "DLL":**

- Abre **File > New > Project**.
- Selecciona **DLL** y asigna un nombre al proyecto (por ejemplo, mimath_dll).

2. **Configurar el archivo fuente:**

- Escribe las definiciones de las funciones y asegúrate de usar las directivas de exportación para las funciones.

Archivo mimath.h**:**

```
#ifndef MIMATH_H
#define MIMATH_H
#ifdef _WIN32
#define DLL_EXPORT __declspec(dllexport)
#else
#define DLL_EXPORT
#endif
DLL_EXPORT int suma(int a, int b);
DLL_EXPORT int resta(int a, int b);
DLL_EXPORT int factorial(int n);
```

```
#endif
```

Archivo mimath.cpp**:**

```
#include "mimath.h"
int suma(int a, int b) {
	return a + b;
}
int resta(int a, int b) {
	return a - b;
}
int factorial(int n) {
	if (n <= 1) return 1;
	return n * factorial(n - 1);
}
```

3. **Compilar la DLL:**

- Haz clic en **Execute > Compile**. Dev-C++ generará un archivo .dll (por ejemplo, mimath.dll) junto con un archivo .a necesario para enlazar.

4. **Usar la librería en otro proyecto:**

- Crea un nuevo proyecto con un archivo fuente (por ejemplo, main.cpp) que incluya el encabezado mimath.h.
- Ve a **Project > Project Options > Parameters > Linker** y agrega el archivo .a generado con la DLL.
- Asegúrate de que el archivo .dll esté en el mismo directorio que el ejecutable.

Archivo main.cpp**:**

```
#include <iostream>
#include "mimath.h"
using namespace std;
int main() {
	cout << "Suma: " << suma(3, 7) << endl;
	cout << "Factorial: " << factorial(5) << endl;
	return 0;
}
```

5. **Compila y ejecuta:** El programa enlazará dinámicamente con la DLL.

Conclusión

- Usa **librerías estáticas** para proyectos pequeños o cuando necesites simplicidad.
- Usa **librerías dinámicas** para ahorrar memoria en programas grandes o cuando desees compartir el código entre múltiples programas.

Con Dev-C++, todo esto se puede hacer de manera gráfica sin necesidad de comandos manuales.

El archivo principal, que contiene la función main, puede tener cualquier nombre válido para un archivo fuente en C++ (por ejemplo, programa.cpp, inicio.cpp o miPrograma.cpp). Sin embargo, este archivo **debe cumplir con ciertas responsabilidades clave** para que el programa funcione correctamente:

Requisitos del archivo principal:

1. **Debe contener la función** main**:**

- La función main es el punto de entrada del programa. Sin esta función, el compilador no sabrá dónde comenzar la ejecución.

Ejemplo:

```
int main() {
        // Código principal del programa
        return 0;
}
```

2. **Debe incluir los encabezados necesarios (**#include**):**

- El archivo principal necesita incluir tanto las librerías estándar como tus propios archivos de encabezado, para que las funciones y recursos estén disponibles.

Ejemplo:

```
#include <iostream> // Librería estándar para entrada/salida
#include "mimath.h" // Tu propia librería
```

3. **Es donde se invocan las funciones que deseas usar:**

- Las funciones definidas en tus librerías (o en otros archivos del proyecto) se llaman desde la función main o desde otras funciones en este archivo.

Ejemplo:

```
int main() {
        int resultado = suma(5, 3); // Invoca la función suma
        std::cout << "Resultado: " << resultado << std::endl;
        return 0;
}
```

Distribución del código en el proyecto:

Aunque el archivo principal **debe contener la función** main, el resto de tu código (como funciones auxiliares, clases, etc.) puede estar distribuido en otros archivos fuente (.cpp) o encabezados (.h) para mantener una estructura organizada.

Ventajas de esta estructura:

1. **Flexibilidad en el nombre del archivo principal:** Puedes llamarlo como prefieras, lo importante es que el compilador pueda encontrar la función main.
2. **Organización del código:** Mantener funciones, clases y datos en archivos separados mejora la claridad y la reutilización.
3. **Compatibilidad con proyectos grandes:** Este enfoque es fundamental para proyectos más complejos con múltiples módulos.

Por lo tanto, mientras incluyas las librerías y llames a tus funciones desde el archivo principal, puedes nombrarlo como desees.