Guía Completa de C++ para Robótica y Visión

Concepto 1: ¿Qué es C++ y por qué es relevante en robótica y visión? 🤖👁️

C++ es un lenguaje de programación potente y de **propósito general**. Es una extensión del lenguaje C, lo que significa que hereda su eficiencia y control de bajo nivel sobre el hardware. Sin embargo, C++ añade características clave de **Programación Orientada a Objetos (POO)**, lo que lo hace mucho más versátil y estructurado para proyectos complejos.
¿Por qué es tan usado en robótica y visión por computadora?

1. **Rendimiento:** C++ es extremadamente rápido. En campos como la robótica y la visión, donde se procesan grandes volúmenes de datos (imágenes, datos de sensores) en tiempo real, la velocidad de ejecución es crucial. Por ejemplo, ORB-SLAM3 necesita procesar fotogramas de cámara a alta velocidad para construir mapas y localizar un dron.
2. **Control de Hardware:** Permite un control muy fino sobre la memoria y el hardware. Esto es vital para sistemas embebidos, como los drones, y para optimizar el uso de recursos limitados.
3. **Librerías Potentes:** Muchas de las librerías más importantes en robótica y visión están escritas en C++ o tienen sus interfaces principales en este lenguaje. Ejemplos incluyen **OpenCV** (para visión artificial), **Eigen** (para álgebra lineal de alto rendimiento) y gran parte de la infraestructura de **ROS 2**. Esto facilita la integración y el uso de estas herramientas.
4. **POO (Programación Orientada a Objetos):** Permite organizar el código de manera modular, escalable y reutilizable. Esto es fundamental para gestionar la complejidad de proyectos como ORB-SLAM3, donde diferentes componentes (fotogramas clave, puntos del mapa, cámaras) se modelan como objetos interconectados.

Concepto 2: Estructura Básica de un Programa C++ 📝

Todo programa en C++ sigue una estructura fundamental.

- **Comentarios (// y /* ... */):** Se utilizan para documentar el código. El compilador los ignora. Son esenciales para entender y mantener código, especialmente en proyectos complejos.
    - // Esto es un comentario de una sola línea.
    - /* Esto es un comentario de múltiples líneas. */
- **Directivas de Preprocesador (#include):** Indican al compilador que incluya el contenido de otras librerías o archivos de cabecera. Por ejemplo, #include <iostream>

es necesario para usar funcionalidades de entrada y salida de datos (como imprimir en pantalla).

- **Función main():** Es el punto de entrada de todo programa C++. Cuando ejecutas un programa C++, el sistema operativo comienza la ejecución en esta función.
    - int: Indica que la función main devolverá un valor entero (0 si la ejecución fue exitosa, diferente de 0 si hubo un error).
    - (): Los paréntesis indican que es una función y pueden contener argumentos (aunque en el ejemplo básico están vacíos).
    - {}: Las llaves delimitan el cuerpo de la función, conteniendo el código que se ejecutará.
- **std::cout y std::endl:**
    - **std::cout**: Es un objeto de la librería iostream que representa la **salida estándar** (normalmente la consola). Se utiliza junto con el operador de inserción << para imprimir datos.
    - **std::endl**: Es un manipulador que inserta un carácter de nueva línea y vacía el buffer de salida, asegurando que el texto se imprima inmediatamente.
    - **std::**: Este prefijo indica que cout y endl pertenecen al **espacio de nombres std (Standard)**. Los espacios de nombres son una forma de organizar el código en C++ para evitar conflictos de nombres entre diferentes librerías o partes de un programa.
- **return 0;:** Al final de main, indica que el programa se ejecutó correctamente.

Concepto 3: Variables y Tipos de Datos Fundamentales 💾

Las **variables** son espacios con nombre en la memoria de la computadora que se utilizan para almacenar valores. Cada variable debe tener un **nombre** (para identificarla) y un **tipo de dato** (para especificar qué clase de información guardará).

Tipos de Datos Fundamentales en C++

Tipo de Dato	Descripción	Ejemplos de Valores	Aplicación en Robótica/Visión
int	Números enteros (sin decimales).	5, -100, 0	Contadores, índices de arrays, IDs de objetos, tamaños de estructuras, número de puntos clave.
float	Números de punto flotante (con decimales), precisión simple.	3.14f, -0.5f, 1.23e-5f	Coordenadas 2D (x, y), mediciones de sensores donde la alta precisión no es crítica. Se usa poco en cálculo

			numérico preciso.
double	Números de punto flotante (con decimales), **precisión doble**.	3.1415926535, -1.23e-5	**Muy usado.** Coordenadas 3D (x, y, z), orientaciones (cuaterniones), cálculos de transformaciones, mediciones de sensores de alta precisión. Es el tipo preferido para cálculos geométricos y numéricos en robótica.
char	Un solo carácter.	'a', 'Z', '7', '\n' (salto de línea)	Representar caracteres individuales, parte de cadenas de texto.
bool	Valor booleano: verdadero o falso.	true, false	Banderas de estado (ej. esta_mapeando = true), resultados de comparaciones lógicas.

Nota: La f al final de un número decimal (15.75f) es necesaria para que C++ lo interprete como float; de lo contrario, por defecto lo trata como double.

Declaración e Inicialización

- **Declaración:** tipo_dato nombre_variable;
- **Inicialización:** nombre_variable = valor;
- **Declaración e Inicialización combinadas (buena práctica):** tipo_dato nombre_variable = valor_inicial;

Concepto 4: Operadores en C++ ➕➖✖️➗

Los **operadores** son símbolos que indican al compilador que realice una operación sobre uno o más operandos (valores o variables).

1. Operadores Aritméticos

Para cálculos matemáticos:

Operador	**Descripción**
+	Suma
-	Resta

*	Multiplicación
/	División (¡cuidado con la división entera si ambos operandos son enteros!)
%	Módulo (resto de la división entera)

2. Operadores de Asignación

Para asignar valores a variables. El operador básico es =. Los operadores compuestos combinan una operación aritmética con la asignación:

Operador	Equivalente a...	Descripción
=	x = valor;	Asigna el valor.
+=	x = x + valor;	Suma y asigna.
-=	x = x - valor;	Resta y asigna.
*=	x = x * valor;	Multiplica y asigna.
/=	x = x / valor;	Divide y asigna.
%=	x = x % valor;	Módulo y asigna.

3. Operadores de Incremento y Decremento

Para aumentar o disminuir un valor en 1.

Operador	Descripción
++	Incrementa el valor en 1 (x = x + 1;)
--	Decrementa el valor en 1 (x = x - 1;)

Post-incremento (x++) vs. Pre-incremento (++x):

- **x++ (Post-incremento):** El valor de x se **usa primero** en la expresión, y **luego** se incrementa.
- ++x (Pre-incremento): El valor de x se incrementa primero, y luego se usa en la expresión.
  La diferencia solo es relevante cuando el operador es parte de una expresión más grande.

Concepto 5: Operadores Relacionales y Lógicos 🤔

Estos operadores son cruciales para tomar decisiones en el código, ya que devuelven un valor booleano (true o false).

1. Operadores Relacionales (o de Comparación)

Comparan dos valores:

Operador	Descripción
==	Igual a
!=	Diferente de / No igual a

<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

2. Operadores Lógicos

Combinan expresiones booleanas:

Operador	**Descripción**
&&	Y lógico (AND): Verdadero si AMBOS son verdaderos.
`	
!	Negación lógica (NOT): Invierte el valor booleano.

Precedencia de Operadores: Los operadores tienen un orden de evaluación. En caso de duda o para mayor claridad, usa paréntesis () para forzar el orden deseado.

Concepto 6: Control de Flujo (Parte 1) - Sentencias Condicionales (if, else if, else) 🚦

Permiten que tu programa ejecute diferentes bloques de código basándose en condiciones.

1. if (Si)

Ejecuta un bloque de código si la condición es verdadera:

```
if (condicion) {
   // Código a ejecutar
}
```

2. if-else (Si-Sino)

Ejecuta un bloque si la condición es verdadera, y un bloque alternativo si es falsa:

```
if (condicion) {
   // Código si es verdadero
} else {
   // Código si es falso
}
```

3. if-else if-else (Si-Sino Si-Sino)

Para múltiples condiciones excluyentes. Se evalúan en orden, y se ejecuta el primer bloque

cuya condición sea verdadera. Si ninguna es verdadera, se ejecuta el else (si existe).

```
if (condicion1) {
   // Código si condicion1
} else if (condicion2) {
   // Código si condicion1 es falsa Y condicion2 es verdadera
} else {
   // Código si ninguna es verdadera
}
```

Concepto 7: Control de Flujo (Parte 2) - Bucles (for, while) 🔄

Permiten repetir un bloque de código múltiples veces. Esencial para procesar secuencias de datos (ej., fotogramas de vídeo).

1. Bucle while (Mientras)

Repite un bloque de código **mientras** una condición sea verdadera. La condición se evalúa antes de cada iteración.

```
while (condicion) {
   // Código a repetir
}
```

Es crucial que la condición se vuelva falsa en algún momento para evitar un bucle infinito.

2. Bucle for (Para)

Se usa cuando se sabe de antemano cuántas veces se quiere repetir el código, o para iterar sobre un rango. Tiene tres partes:

```
for (inicializacion; condicion; actualizacion) {
   // Código a repetir
}
```

1. **inicializacion**: Se ejecuta una vez al principio (ej., int i = 0;).
2. **condicion**: Se evalúa antes de cada iteración (ej., i < N;).
3. **actualizacion**: Se ejecuta después de cada iteración (ej., ++i;).

Concepto 8: Arrays y Estructuras (Structs) 📦

Permiten agrupar datos.

1. Arrays (Arreglos)

Una colección de elementos del **mismo tipo de dato**, almacenados contiguamente en memoria y accedidos por un **índice** (que comienza en 0). Tienen un **tamaño fijo** en tiempo de

compilación.
```
tipo_dato nombre_array[tamaño]; // Declaración
nombre_array[indice] = valor;   // Acceso
```

2. Estructuras (Structs)

Un tipo de dato definido por el usuario que agrupa variables de **diferentes tipos de datos** bajo un solo nombre, representando una entidad cohesionada.

```
struct NombreEstructura {
   tipo_dato miembro1;
   tipo_dato miembro2;
};

NombreEstructura mi_variable;
mi_variable.miembro1 = valor; // Acceso a miembros con el operador '.'
```

Concepto Adicional: Punteros (*) y Referencias (&) en C++ 🎯

Punteros (*)

Una variable que almacena la **dirección de memoria** de otra variable. Funcionan igual que en C.

- **Declaración:** tipo_dato* nombre_puntero;
- **Obtener dirección:** &variable;
- **Desreferenciación (acceder al valor al que apunta):** *nombre_puntero;
- **Acceder a miembros de struct/clase a través de puntero:** puntero->miembro;

Los punteros pueden ser nullptr (no apuntar a nada) y deben ser manejados con cuidado para evitar errores de memoria.

Referencias (&)

Una característica propia de C++. Una **referencia** es un **alias** (un nombre alternativo) para una variable ya existente.

- **Declaración e inicialización:** tipo_dato& nombre_referencia = variable_existente;
- **No se pueden "reapuntar":** Una vez inicializada, siempre se refiere a la misma variable.
- **No se pueden desreferenciar:** Se usan directamente como si fueran la variable original.
- **No pueden ser nulas:** Siempre deben referenciar a algo válido.

Las referencias son muy utilizadas para **pasar parámetros a funciones por referencia** (tipo_dato& o const tipo_dato&) para evitar copias costosas de objetos grandes y/o permitir la modificación del original. Ofrecen una sintaxis más limpia y son más seguras que los punteros

en muchos contextos.

Concepto 10: Clases y Programación Orientada a Objetos (POO) 🧱

La POO es un paradigma de programación que organiza el código en torno a **objetos**, que son instancias de **clases**. Una **clase** es un **plano** que define los datos (atributos) y las acciones (métodos) de una entidad.

Componentes Clave:

- **Atributos (Miembros de Datos):** Variables que describen el estado del objeto.
- **Métodos (Funciones Miembro):** Funciones que definen el comportamiento del objeto.
- **Constructor:** Método especial que se llama automáticamente al crear un objeto. Se usa para inicializar sus atributos.
- **Destructor:** Método especial que se llama automáticamente cuando un objeto es destruido. Se usa para liberar recursos.
- **Encapsulación:** Principio de ocultar los detalles internos de un objeto y exponer solo una interfaz controlada. Se logra con **especificadores de acceso**:
  - **public:**: Miembros accesibles desde fuera de la clase (la interfaz).
  - **private:**: Miembros accesibles solo desde dentro de la clase. Protege los datos.
  - **protected:**: Miembros accesibles desde la propia clase y sus clases derivadas (hijas).

Concepto 11: Herencia y Polimorfismo (POO Avanzada) 🧬

1. Herencia (Inheritance)

Permite que una **clase derivada (hija)** adquiera los atributos y métodos de una **clase base (padre)**. Modela relaciones "es-un" (ej., un DronAereo *es un* Dron). Promueve la reutilización y organización del código.

```
class ClaseBase { /* ... */ };
class ClaseDerivada : public ClaseBase { /* ... */ };
```

2. Polimorfismo (Polymorphism)

Significa "muchas formas". Permite que objetos de diferentes clases relacionadas por herencia sean tratados como objetos de su clase base común. La clave son los métodos **virtuales (virtual)**.

- **virtual**: Si un método de la clase base es virtual y es sobrescrito en una clase derivada, al llamar a ese método a través de un puntero o referencia a la clase base, se ejecutará la versión del método de la clase **derivada** (enlace dinámico).

- **Método Virtual Puro (virtual void metodo() = 0;):** Hace que la clase sea **abstracta** (no se pueden crear objetos directamente de ella) y obliga a las clases derivadas a implementar ese método.

Concepto 12: Librería Estándar de Contenedores (STL) - std::vector 📏

std::vector es el contenedor más utilizado de la STL. Es un **array dinámico** que puede crecer o encogerse en tiempo de ejecución.

Ventajas:

- **Tamaño Dinámico:** No necesitas conocer el tamaño de antemano.
- **Manejo Automático de Memoria:** std::vector gestiona new y delete internamente.
- **Seguridad:** Métodos como at() ofrecen comprobación de límites.
- **Funcionalidades Integradas:** Métodos como push_back(), size(), empty(), clear().

Uso:

- #include <vector>
- std::vector<TipoDeDato> nombre_vector;
- nombre_vector.push_back(valor);
- nombre_vector.size();
- nombre_vector[indice]; (sin comprobación de límites)
- nombre_vector.at(indice); (con comprobación de límites, lanza std::out_of_range)
- **Bucle for basado en rango:** for (const TipoDeDato& elemento : nombre_vector) { ... } (muy recomendado).

Concepto 13: STL - std::map, std::unordered_map y std::string 🏷️🔡

1. std::string (Cadenas de Texto)

El tipo de dato preferido en C++ para manejar cadenas de texto. Ofrece manejo automático de memoria y multitud de métodos para manipulación (concatenación con +, length(), find(), substr(), etc.).

- #include <string>

2. std::map (Mapa Ordenado Clave-Valor)

Almacena elementos en pares **clave-valor**, donde cada **clave es única** y los elementos se mantienen **ordenados por clave**. Ideal para diccionarios o configuraciones.

- #include <map>
- std::map<TipoClave, TipoValor> nombre_mapa;
- nombre_mapa[clave] = valor; (inserta o actualiza)
- **Iteración:** for (const auto& par : nombre_mapa) { par.first; par.second; }
- **Verificar existencia:** nombre_mapa.count(clave); o nombre_mapa.find(clave);

(preferible para evitar inserción por defecto).

3. std::unordered_map (Mapa No Ordenado Clave-Valor)

Similar a std::map, pero los elementos **no están ordenados**. Utiliza una tabla hash para un **rendimiento de acceso mucho más rápido** (O(1) en promedio).

- #include <unordered_map>
- std::unordered_map<TipoClave, TipoValor> nombre_mapa;

Cuándo usar cuál:

- **std::map**: Si necesitas los elementos ordenados por clave.
- **std::unordered_map**: Si la velocidad de acceso es la prioridad y el orden no importa.

Concepto 14: Manejo de Excepciones (try, catch, throw) 🚨

Mecanismo estructurado para detectar y manejar errores inesperados durante la ejecución, evitando que el programa falle.

- **try**: Bloque de código donde se espera que pueda ocurrir una excepción.
- **throw**: Se usa para **lanzar** una excepción cuando se detecta un error.
- **catch**: Bloque de código que se ejecuta cuando se **captura** una excepción de un tipo específico. Puede haber múltiples catch para diferentes tipos de error.
- **std::exception**: Clase base para todas las excepciones estándar.
  - std::runtime_error: Para errores de ejecución generales.
  - std::invalid_argument: Para argumentos de función no válidos.
  - std::out_of_range: Para accesos fuera de límites de contenedores.
- **e.what()**: Método de las excepciones estándar para obtener un mensaje descriptivo del error.
- **std::cerr**: Flujo de salida para mensajes de error.

Concepto 15: Plantillas (Templates) 🧩

Permiten escribir **código genérico** que puede operar con diferentes tipos de datos sin reescribir el mismo código.

1. Plantillas de Función

Permiten que una función opere con diferentes tipos:
template <typename T>
T funcion_generica(T a, T b) { /* ... */ }

El compilador genera automáticamente la versión específica para cada tipo usado.

2. Plantillas de Clase

Permiten que una clase opere con diferentes tipos de datos (ej., std::vector<int>, std::vector<DronTello>):

```
template <typename T>
class ClaseGenerica {
   T dato;
public:
   ClaseGenerica(T val) : dato(val) {}
   // ...
};
```

Las plantillas son fundamentales para la flexibilidad y eficiencia de librerías como la STL, Eigen y OpenCV, y las verás extensamente en ORB-SLAM3 y ROS 2.

Concepto 16: Punteros Inteligentes (std::unique_ptr, std::shared_ptr) 🧠

Objetos de la STL que actúan como punteros pero **gestionan automáticamente la memoria** a la que apuntan, liberándola cuando ya no es necesaria. Esto previene fugas de memoria y punteros colgantes. Requieren #include <memory>.

1. std::unique_ptr

- **Propiedad Única:** Solo un unique_ptr puede poseer el objeto.
- **No se Copia, Se Mueve:** La propiedad se transfiere con std::move().
- **Eficiencia:** Muy eficiente, sin sobrecarga de conteo de referencias.
- **Creación preferida:** std::make_unique<Tipo>(argumentos_constructor);

2. std::shared_ptr

- **Propiedad Compartida:** Múltiples shared_ptrs pueden compartir la propiedad del mismo objeto.
- **Conteo de Referencias:** La memoria se libera cuando el contador de referencias llega a cero.
- **Creación preferida:** std::make_shared<Tipo>(argumentos_constructor);

Los punteros inteligentes son esenciales para un manejo de memoria seguro y robusto en C++ moderno, especialmente en sistemas complejos donde los objetos se asignan y comparten dinámicamente.

Concepto 17: CMake - Gestión de Proyectos en C++ 🏗️

CMake es un **generador de sistemas de construcción** multiplataforma. Lee archivos de configuración (CMakeLists.txt) y genera archivos de construcción nativos (ej., Makefile en Linux) para compilar proyectos C++ complejos con múltiples archivos, librerías y dependencias.

Estructura de Directorios del Ejemplo:

```
proyecto_dron_cmake/
├── CMakeLists.txt
├── src/
│   ├── main_dron.cpp
│   └── utilidades/
│       ├── control_dron.h
│       └── control_dron.cpp
```

Contenido de los Archivos:
proyecto_dron_cmake/CMakeLists.txt (en el directorio raíz):

```
# Version minima de CMake requerida
cmake_minimum_required(VERSION 3.10)

# Nombre de tu proyecto
project(MiProyectoDron CXX) # CXX indica que es un proyecto C++

# Especificar el estandar de C++ a usar (¡importante para C++ moderno!)
# C++11 es el minimo para ROS 2 y muchas librerias, C++14/17/20 son comunes ahora.
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Añadir el subdirectorio 'src' para que CMake procese su CMakeLists.txt interno
add_subdirectory(src)

# Opcional: Configuraciones adicionales, busqueda de librerias, etc.
# find_package(OpenCV REQUIRED) # Buscar la libreria OpenCV
# include_directories(${OpenCV_INCLUDE_DIRS}) # Añadir rutas de cabeceras de OpenCV
```

proyecto_dron_cmake/src/CMakeLists.txt (en el subdirectorio src/):

```
# No necesitas project() ni cmake_minimum_required() aqui, se heredan del padre.

# Añadir el subdirectorio 'utilidades'
add_subdirectory(utilidades)

# Definir un ejecutable
# add_executable(nombre_ejecutable archivo1.cpp archivo2.cpp ...)
add_executable(dron_control_app main_dron.cpp)

# Enlazar el ejecutable con la libreria de utilidades
# target_link_libraries(nombre_ejecutable PUBLIC nombre_libreria)
target_link_libraries(dron_control_app PUBLIC DronControlUtilidades) # El nombre de la
libreria definida en utilidades
```

proyecto_dron_cmake/src/utilidades/CMakeLists.txt (en el subdirectorio src/utilidades/):

```
# Definir una libreria (estatica o compartida)
# add_library(nombre_libreria [STATIC|SHARED] archivo1.cpp archivo2.cpp ...)
add_library(DronControlUtilidades STATIC control_dron.cpp) # Crea una libreria estatica

# Especificar las rutas de inclusion para esta libreria (donde estan sus .h)
# target_include_directories(nombre_libreria PUBLIC ruta_cabeceras)
# PUBLIC significa que quien use esta libreria tambien necesitara estas cabeceras.
target_include_directories(DronControlUtilidades PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

proyecto_dron_cmake/src/utilidades/control_dron.h:

```
#ifndef CONTROL_DRON_H
#define CONTROL_DRON_H

#include <string>
#include <iostream>

// Clase simple para el ejemplo
class ControladorDron {
public:
    ControladorDron(std::string id);
    void despegar_simulado();
    void aterrizar_simulado();

private:
    std::string id_controlador;
};

#endif // CONTROL_DRON_H
```

proyecto_dron_cmake/src/utilidades/control_dron.cpp:

```
#include "control_dron.h" // Incluye la cabecera de la propia clase

ControladorDron::ControladorDron(std::string id) : id_controlador(id) {
    std::cout << "Controlador Dron " << id_controlador << " inicializado." << std::endl;
}

void ControladorDron::despegar_simulado() {
    std::cout << id_controlador << ": Comando de despegue simulado ejecutado." << std::endl;
    // Aqui iria la logica real de despegue
}
```

```
void ControladorDron::aterrizar_simulado() {
    std::cout << id_controlador << ": Comando de aterrizaje simulado ejecutado." << std::endl;
    // Aqui iria la logica real de aterrizaje
}
```

proyecto_dron_cmake/src/main_dron.cpp:

```
#include <iostream>
#include <string>
#include "utilidades/control_dron.h" // Incluye la cabecera de la utilidad

int main() {
    std::cout << "--- Iniciando Aplicacion Principal del Dron ---" << std::endl;

    // Crear un objeto de la clase ControladorDron definida en la libreria
    ControladorDron mi_controlador("Dron-Principal");

    mi_controlador.despegar_simulado();
    mi_controlador.aterrizar_simulado();

    std::cout << "--- Aplicacion Principal del Dron Finalizada ---" << std::endl;

    return 0;
}
```

Pasos para Compilar con CMake en la Terminal:

1. **Crea la estructura de directorios** en tu escritorio (o donde prefieras):
   mkdir -p ~/Desktop/proyecto_dron_cmake/src/utilidades

2. **Crea los archivos CMakeLists.txt y los archivos .cpp / .h** con el contenido exacto que te proporcioné, asegurándote de que estén en los directorios correctos.
3. **Abre tu terminal** y sigue los pasos de compilación:
   cd ~/Desktop/proyecto_dron_cmake
   mkdir build
   cd build
   cmake ..
   make
   ./src/dron_control_app # O podria ser ./dron_control_app dependiendo de la version de cmake o tu sistema

    - Si make no funciona, prueba cmake --build . en el directorio build.
    - Si el ejecutable no está en ./src/, intenta buscarlo directamente en ./dron_control_app.
4. **Observa la salida:** Si todo va bien, deberías ver los mensajes de inicialización y control

de tu dron.

CMake es el estándar para proyectos C++ grandes y es fundamental para trabajar con ORB-SLAM3 y ROS 2, ya que todos los paquetes de ROS 2 utilizan CMake para su construcción.