MÓDULO 1 : INTRODUCCIÓN A C++

Conceptos básicos sobre C++

Lo primero que tenemos que saber sobre C++ es que se trata de un lenguaje de programación **compilado** y de **alto nivel**. Para entender estos conceptos tenemos que partir de la base de lenguaje de programación.

Entendemos por **lenguaje de programación** una gramática artificial que proporciona a una persona la capacidad y habilidad de escribir una serie de instrucciones en órdenes en forma de **algoritmos**. Gracias a esto, los **programas** son capaces de ejecutar determinadas tareas.

A raíz de esto podemos hacer la primera diferenciación de lenguajes de programación entre **compilados** e **interpretados**.

Un lenguaje **compilado** como C++ precisa de ser previamente compilado para su posterior ejecución, lo que hace que todas las líneas se tengan que leer para ser compilado mostrando así errores previos a ejecución (";", llaves sin cerrar, mala sintaxis …). Mientras que los lenguajes **interpretados** carecen del paso de compilación por lo que pueden ser ejecutados cuando quieres pero no aparecen los errores hasta que te encuentras en medio de la ejecución.

Ambos tipos de lenguaje son válidos y presentan tanto sus ventajas como desventajas que van desde la gestión de errores, dificultad de compilación, tamaño de los archivos e incluso la eficiencia en la gestión de memoria y recursos del dispositivo.

La siguiente diferenciación se trata de los lenguajes de **alto** o **bajo nivel.** Podemos entender por lenguajes de bajo nivel aquellos que se encuentran lo más próximo al lenguaje máquina (binario) como pueden ser el **binario**, **ensamblador, C, LISP** … mientras que podemos clasificar al resto de lenguajes como de alto nivel como pueden ser **Java, C++, JavaScript, TypeScript, Python** …

También tenemos que decir que C++ es un lenguaje **orientado a objetos**, **Fuerte, Estático y Nominativo**.

Lenguajes y Paradigmas de Programación

Para abordar este tema vamos a definir primero lo que es un **paradigma de programación**.

Entendemos por **paradigma de programación** como una propuesta tecnológica adaptada por una comunidad de programadores incuestionable a que unívocamente trata de resolver uno o varios problemas claramente delimitados.

Dentro de C++ podemos tratar de varios tipos de paradigma que comentaremos una a una. Habitualmente a la hora de resolver un problema se mezclan todos los tipos de paradigmas a la hora de hacer la programación. De esa manera se origina la **programación multiparadigma**, pero el que actualmente es más usado de todos los próximos paradigmas es el de la **programación orientada a objetos**.

★ Programación Imperativa

La **programación imperativa** se centra en describir paso a paso cómo realizar una tarea. En este enfoque, un programa se compone de una serie de instrucciones que modifican el estado del programa a lo largo del tiempo. Es como darle a un ordenador una lista detallada de tareas a realizar, una tras otra.

Los principios fundamentales de esta paradigma son:

- **Secuencia de Instrucciones:** Las instrucciones se ejecutan secuencialmente, una después de la otra. Esto significa que el orden en el que se escriben las instrucciones importa y afecta el resultado final del programa.
- **Estado mutable:** Los programas imperativos hacen un uso extensivo de variables y estructuras de datos mutables. Estos valores pueden cambiar a lo largo de la ejecución del programa, lo que permite a los desarrolladores realizar seguimiento de cambios y manipular datos según sea necesario.
- **Estructuras de control:** Los bucles, las estructuras condicionales y las funciones son elementos fundamentales en la programación imperativa. Estas construcciones controlan el flujo de ejecución del programa y permiten tomar decisiones basadas en ciertas condiciones.

★ Programación funcional

La **programación funcional** se centra en el uso de verdaderas funciones matemáticas. En este estilo de programación las funciones son ciudadanas de primera clase, porque sus expresiones pueden ser asignadas a variables como se haría con cualquier otro valor; además de que pueden crearse funciones de orden superior.

Esta tiene sus raíces en el **cálculo lambda**, un sistema formal desarrollado en los 30 para investigar la naturaleza de las funciones, la naturaleza de la computabilidad y su relación con la recursión.

★ Programación Lógica

La **programación lógica** es un tipo de paradigma de programación dentro del paradigma de la **programación declarativa**.

La programación lógica encuentra su hábitat natural en aplicaciones de inteligencia artificial o relacionadas con:

- **Sistemas expertos:** Donde un sistema de información imita las recomendaciones de un experto sobre algún dominio de conocimiento.
- **Demostración automática** de teoremas, donde un programa genera nuevos teoremas sobre una teoría existente.
- Reconocimiento de **lenguaje natural**, donde un programa es capaz de comprender la información contenida en una expresión lingüística humana.

★ Declarativo

La **programación declarativa** no determina el cómo, sino que funciona con un nivel de abstracción más alto que la **programación imperativa**. A diferencia de esta, la programación declarativa deja margen para la optimización. Este tipo de paradigma da como resultado un SW mejor preparado para el futuro, ya que no es necesario determinar mediante qué procedimiento se alcanza un resultado.

Su mayor ventaja es que tiene la capacidad de describir problemas de forma más corta y precisa que el lenguaje imperativo

★ POO (Programación Orientada a Objetos) [OOP]

La **POO** es un paradigma orientado a objetos que organiza el código en entidades llamadas “objetos” que pueden contener datos en forma de atributos y funciones, facilitando la modularidad, la reutilización de código y el mantenimiento del SW.

Un **objeto en programación** va más allá de ser una simple entidad que contiene datos y funciones. Es la materialización de la POO, encapsulando la ide a de la abstracción.

La implementación de la POO en programación se lleva a cabo a través de lenguajes específicos diseñados para admitir este paradigma Java, Python o C++.

★ Por procedimientos

La **programación procedural** o **procedimental** se deriva de la **programación estructurada**. Consiste en dividir el código en secciones lógicas llamadas rutinas o

procedimientos, donde cada procedimiento resuelve una tarea específica, y se ejecuta cada vez que sea necesario. Cabe destacar que un procedimiento puede llamar a otros procedimientos. Algunos de los primeros lenguajes procedurales son FORTRAN, ALGOL, COBOL Y BASIC.

Compilador C++

Para entender el compilador de C++ tenemos que entender lo que es un compilador.

Un **compilador** es un software que transcribe el código escrito en un lenguaje de **alto nivel** en **lenguaje máquina**. Un compilador usualmente genera código ensamblador primero luego traduce este ensamblador a lenguaje máquina.

Un compilador consta de 3 partes fundamentales:

- Analizador Léxico: Es la primera parte del proceso de compilación. Responsable de dividir el programa en **Tokens**, estos tokens se basan en una tabla definida para el propio lenguaje.

- Analizador Sintáctico: Es la segunda parte del compilador y se encarga de generar un **Árbol Sintáctico,** que no es nada más que una estructura de datos en forma de árbol que representa de forma simple el programa.

- Analizador Semántico: Es el último paso antes de la compilación. Este prepara el código para ser compilado. El propio analizador comienza desde el árbol abstracto y tiene como propósito validar los puntos del programa. Valida la compatibilidad de tipos, los valores usados y el contexto. Es el encargado de generar los mensajes de errores sobre nuestro código.

Unidad Entrada\Salida (cin\cout)

Las bibliotecas estándar de C++ proporcionan un amplio conjunto de capacidades de entrada/salida (E/S).

C++ utiliza E/S a prueba de tipos. Cada operación de E/S se realiza automáticamente en una forma sensible con respecto al tipo de datos.

La E/S de C++ se da en flujos de bytes. Un flujo es simplemente una secuencia de bytes. En las operaciones de entrada, los bytes fluyen desde un dispositivo (ratón, teclado, …)hacia la memoria principal. En operaciones de salida los bytes fluyen de la memoria principal hacia un dispositivo (pantalla, impresora, disco…).

La aplicación asocia significado a los bytes. Pueden representar carácterse ASCII, o cualquier otro tipo de información que pueda requerir una aplicación.

★CIN / COUT

- CIN : Es el flujo de entrada estándar que normalmente es el teclado.
- COUT: Es el flujo de salida estándar que por lo general es la pantalla.

★Librería Estándar C++

Para poder usar la librería estándar de c++ tenemos que poner la línea de código:

```
using namespace std;
```

Esto hace que usemos la librería estándar lo que nos permite no tener que poner la instrucción al completo en c++

```
// instrucciones sin librería estándar
std::cout << "Ejemplo de salida sin std" << std::endl;
unsigned i = 0;
std::cin >> i;
// instrucciones con librería estándar
cout << "Ejemplo de salida con std" << endl;
unsigned i = 0;
cin >> i;
```

Estructura del programa && Hello World

Para estas clases vamos a usar la estructuración que uso yo en mi día a día que, aunque sea algo complicada a la hora de comenzar con ficheros o proyectos pequeños de cara a leer los programas a futuro es más fácil.

Para poder ver la sintaxis que uso voy a copiar un programa hecho y lo explicaré a continuación:

```
//Includes ac1137962
#include <iostream>
#include <math.h>
#include <fstream>
#include <time.h>
#include <mpi.h>

using namespace std;

//Estructura para medir el tiempo
double get_time() {
    struct timespec t;
    clock_gettime ( CLOCK_REALTIME, &t );
    return (double) t.tv_sec + ((double) t.tv_nsec)/1e9;
}
/**
 * @param [in] talla (int) Talla del programa
 *
 * @param [in] particulas (int) Numero de particulas
 *
 * @param [in] t_inicial (double) Tiempo donde inicia la paralelizacion del
programa
 *
 * @param [in] t_final (double)
 * */
 float getFlops(int talla,int particulas, double t_inicial, double t_final);
// Main pricipal
int main(int argc, char *argv[]) {
    /*
        1. Primer parámetro -> argv[1] -> N (talla del programa)
        2. Segundo parámetro -> argv[2] -> M (número de partículas)
        3. Tercer parámetro -> argv[3] -> 0 (depuración) | 1 (experimentacion)
        4. Cuarto parámetro -> argv[4] -> fichero_salida.txt
     */
```

```
    // Declaracion de variables
    int N =atoi(argv[1]); // Talla del programa
    int M = atoi(argv[2]); // Numero de particulas
    int modo =atoi(argv[3]); // Modo de ejecucion
    float* Atenuacion = new float[M]; // Atenuación
    float EnergiaAcumulada = 0; // Energia acumulada
    float *nodeX = new float[N];
    float *nodeY = new float[N];
    float *nodeE = new float[N];
    double t0 = 0,t1 = 0;
    // Variables de la paralelización MPI
    int rank, size;
    // Comienza la paralelización por MPI
    MPI_Init(&argc, &argv); //
    // Obtenemos le id del proceso y los procesos totales
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Ahora hacemos que la inicialización solo se produzca en el proceso root
( 0 )
    if (rank == 0) {
        // Sacamos el tiempo en el root ya que esperamos que tardara mas
        t0 = MPI_Wtime();
        //Inicialización
        if (modo == 0) { // Caso depuracion
            M = 3;
            N = 4; //Talla del programa fija para depuracion
            // Inicializacion del vector para depurar
            nodeX[0] = 0; nodeX[1] = 2; nodeX[2] = 3;
            nodeY[0] = 0; nodeY[1] = 0; nodeY[2] = 2;
            nodeE[0]=100; nodeE[1]=200; nodeE[2]= 50;
        }
        else if (modo == 1) { // Caso experimentacion
            for (int i = 0; i < M; ++i) {
            nodeX[i] = (long) ((N-1) * (rand() / (float)RAND_MAX) + 0.5); // x
            nodeY[i] = (long) ((N-1) * (rand() / (float)RAND_MAX) + 0.5); // y
            nodeE[i] = 100+10000 * (rand() / (float)RAND_MAX);            // e
            }
        }
        else { //Caso has introducido mal el modo de ejecución
            cerr << "Error: Modo de ejecución no reconocido \n ";
            return 0;
        }
```

```
    }
    //
    // Todos tienen las variables
    // Matriz de Energia dinamica
    float **EnergiaSuperficial = new float*[N];
    for (int i = 0; i < N; ++i) {
        EnergiaSuperficial[i] = new float[N];
    }
    // Vector de energia total por particula
    float *EnergiaTotal = new float[M];
    for (int i = 0; i < M; ++i) {
        EnergiaTotal[i] = 0;
    }
    // Ponemos a 0 toda la matriz dinamica de energia
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            EnergiaSuperficial[i][j] = 0;
        }
    }
    MPI_Bcast(&M,  1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&N,  1, MPI_INT, 0, MPI_COMM_WORLD);
    // Variables locales para la paralelización
    float* localX = new float[M/size];
    float* localY = new float[M/size];
    float* localE = new float[M/size];
    float* localEnergiaTotal = new float[M/size];
    // Inicializamos los vectores locales
    for (int i = 0; i < M/size;  ++i) {
        localEnergiaTotal[i] = 0;
    }
    float **localEnergiaSuperficial = new float*[N];
    for (int i = 0; i < N; ++i) {
        localEnergiaSuperficial[i] = new float[N];
    }
    // Vamos a paralelizar los bucles internos mandando partes de vectores a
todos los procesos y devolver la
    //       matriz de EnergiaSuperficial y el vector de Energia acumulada
    // Ahora tenemos que mandar a todos los procesos los datos que queremos
procesar
    // Mandamos a todos partes de los  3 vectores
    //
```

```
    MPI_Scatter(&nodeX, M/size, MPI_FLOAT, localX, M/size, MPI_FLOAT, 0,
MPI_COMM_WORLD); // Mandamos X
    MPI_Scatter(&nodeY, M/size, MPI_FLOAT, localY, M/size, MPI_FLOAT, 0,
MPI_COMM_WORLD); // Mandamos Y
    MPI_Scatter(&nodeE, M/size, MPI_FLOAT, localY, M/size, MPI_FLOAT, 0,
MPI_COMM_WORLD); // Mandamos E
    //
    // EJECUCION DEL PROGRAMA
    // Hemos paralelizado el bucle externo
    for (int i = 0; i < M/size ; ++i) {
        for (int X = 0; X < N; ++X) {
            for (int Y = 0; Y < N; ++Y) {
                Atenuacion[i] =
exp(sqrt(((nodeX[i]-X)*(nodeX[i]-X))+((nodeY[i]-Y)*(nodeY[i]-Y))));
                EnergiaAcumulada = nodeE[i] / (Atenuacion[i] * N * N);
                localEnergiaTotal[i] += EnergiaAcumulada;
                localEnergiaSuperficial[X][Y] += EnergiaAcumulada;
            }
        }
    }
    // Recolectamos los datos procesados
    MPI_Gather(localEnergiaTotal, M/size, MPI_FLOAT, EnergiaTotal, M/size,
MPI_FLOAT, 0, MPI_COMM_WORLD );
    // Realizamos un sum de nuestras energias superficiales parciales
    MPI_Reduce(localEnergiaSuperficial, EnergiaSuperficial,  N, MPI_FLOAT,
MPI_SUM, 0, MPI_COMM_WORLD);


    if (rank == 0) // Solo el root
        t1 = MPI_Wtime(); //Tiempo en el que finalizamos la paralelizacion
    //Acabamos la paralelizacion
    MPI_Finalize();

    // Salida de datos
    if (modo == 1) {
        ofstream file(argv[4], ios::app); // Abrimos el fichero y volcamos los
datos
        if (file) {
            file << "\n Tiempo total de ejecucion --> \n\t" << t1 -t0  << "
segundos"<< endl;
            file << "\n MFlops -->  \n\t" << getFlops(N,M,t0,t1) << endl;
            file << " M --> \n\t" << M << "\n N --> \n\t" << N << endl;
```

```
            file << " NumProcesos --> \n\t" << size << endl;
            cout << "\n Datos guardados en : " << argv[4] << endl;
        }
        file.close();
    }
    return 0;
}
/**
 * @brief Get the Flops object
 *
 * @param talla
 * @param particulas
 * @param t_inicial
 * @param t_final
 * @param procesos
 * @return float
 */
 float getFlops(int talla,int particulas, double t_inicial, double t_final,
int procesos) {
        float flops = 0;
        /*Formula secuencial*/
        flops =talla*talla*particulas*29;
        flops = flops / (t_final - t_inicial);
        return flops / 1e6;
    }
```

Como podemos hay abundante cantidad de comentarios que ya comentaremos como hacerlos para que queden bien y no molesten así como comentar funciones.

La estructura a utilizar es :

1. Declaraciones de los imports a utilizar
2. Librería standar
3. Tipos y estructuras personalizadas
4. Cabeceras de funciones
5. Programa principal
6. Declaración de funciones