

## **Clase C++**

- ❑ **C++ como lenguaje**
  - Diferencias con respecto a C
  - Sintaxis
  - Namespaces
  - Punteros, referencias, smart pointers
  - Memoria dinámica

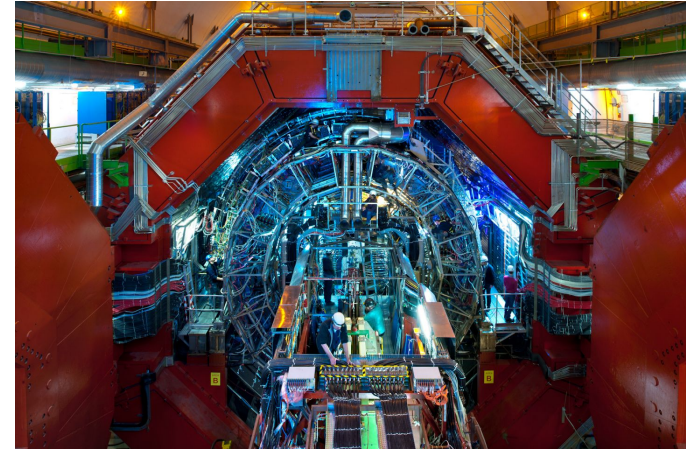
- ❑ **Clases**
  - Tipos de constructores
  - Initialization list
  - Copy constructor / copy assignment
  - Destructor
  - Detalles sobre constructores

- ❑ **Templates**

- ❑ **Librería estándar <stdlib.h>**
  - Collections
    - Iterators
    - For loops
  - Algorithms

- ❑ **Para el TP**
  - Funciones lambda
  - Librería de threads

- **Multiparadigma**
  - POO
  - Imperativa
  - Funcional
- Control total de la **memoria**
  - Asignación manual con new / delete
- **Templates**
- **Namespaces**
- Equilibrio entre **abstracción** y **rendimiento**
- **Compatibilidad** con C
- Ideas para sistemas **grandes** y **complejos**



<https://github.com/AliceO2Group/AliceO2>

En el experimento ALICE del LHC, la transmutación de  $\text{Pb} \rightarrow \text{Au}$  se midió y analizó con AliRoot/ROOT, el framework C++ estándar para el procesamiento de datos en alta energía.

# C y C++: ¿Qué los diferencia?

Característica	C	C++
Compilador	GCC (gcc -o file file.c)	G++ (g++ -o file file.cpp)
IO	printf, scanf	cout, cin (<iostream>)
Manejo de memoria	malloc / free	new / delete
Excepciones	X	try / catch
Sobrecarga de funciones y operadores	X	✓
Variables de referencia	X	✓
Polimorfismo	Manual con punteros	Virtual functions
Amistad (acceso privado)	X	friend classes

- En c++ las más comunes son:
  - o `<iostream>`: Entrada y salida estándar
  - o `<vector>`: Contenedor dinámico
  - o `<algorithm>`: Algoritmos de sorting, búsqueda, manipulación de datos
  - o `<string>`: Manipulación y almacenamiento de cadenas de texto
  - o `<memory>`: Manejo de memoria dinámica, smart pointers
  - o `<thread>`: Gestión de hilos de ejecución
- Sintaxis (No cambia con respecto a C):

```
#include <iostream> // Librería Estándar de C++  
#include "miArchivo.h" // Archivo propio
```

## Sintaxis básica

```
#include <iostream>
using namespace std;

void saludar(string nombre) {
    cout << "Hola " << nombre << "!" << endl;
}

int main() {
    string usuario = "ACSO";
    saludar(usuario);
}
```

Hola ACSO!

```
#include <iostream>
#include <vector>
using namespace std;

bool mayorDeEdad(int edad) {
    return edad >= 18;
}

int main() {
    vector<int> edades{20, 17, 34, 15};
    for (auto edad : edades) {
        if (mayorDeEdad(edad)){
            cout << edad << " → Sos mayor de edad." << endl;
        }
    }
    return EXIT_SUCCESS;
}
```

20 → Sos mayor de edad.

34 → Sos mayor de edad.

- Contenedores que organizan y agrupan identificadores (funciones, variables, clases) para evitar conflictos entre ellos
- Se accede con :: (*scope resolution operator*)

```
#include <iostream>

namespace my_namespace {
    void hello() {
        std::cout << "Hola desde my_namespace" << std::endl;
    }
}

namespace nested {
    namespace inner {
        void hello() {
            std::cout << "Hola desde nested::inner" << std::endl;
        }
    }
}

int main() {
    my_namespace::hello();
    nested::inner::hello();
    return EXIT_SUCCESS;
}
```

- `std::` es el namespace donde se encuentra la librería estándar de C++
  - `Std::cout` → Flujo de salida estándar. Imprime en la consola
  - `Std::cin` → Flujo de entrada estándar. Recibe datos de la consola
  - `Std::cerr` → Flujo de error estándar. Mostrar errores
  - `Std::endl` → Inserta un salto de línea y vacía el búfer de salida inmediatamente(flush).

```
#include <iostream>
```

```
int main() {  
    int number;  
    std::cout << "Ingresa un número: " << std::endl;  
    std::cin >> number;  
    std::cout << "El número que ingresaste es: " << number << std::endl;  
    std::cerr << "Este es un mensaje de error de ejemplo." << std::endl;  
    return 0;  
}
```



- Referencia: Son como los punteros pero desreferencian implícitamente. No requieren el uso de \*.
- Sintaxis:

```
data_type &ref = variable;
```

- Propiedades:
  - o Permiten compartir un objeto entre diferentes variables
  - o Asignación inmutable, no podemos cambiar a qué referencia
  - o No puede ser nula
  - o Cambios en la referencia afecta a la variable original
  - o Los atributos que son referencias tienen que inicializarse mediante initialization lists
  - o Se pueden retornar

```
data_type& functionName(parameters)
```

```
struct A {  
    A(int &i) : j{i} {} // OK  
    // A(int &i) { j = i; } // Error  
    int &j;  
};
```

## Referencias, punteros y smart pointers

```
...  
#include <memory>  
void modificar(int& ref) {ref += 10;}  
int main() {  
    int a, b = 10, 20;  
    int& ref = a; // Referencia: alias fijo a 'a'  
    ref = b; // asigna 20 a 'a', NO cambia referencia  
    int* ptr = &a; // Puntero: puede cambiar de dirección  
    cout << "ptr apunta a a: " << *ptr << endl;  
    ptr = &b;  
    cout << "ptr apunta a b: " << *ptr << endl;  
    modificar(a); // Función que modifica usando referencia  
    cout << "a modificado por referencia: " << a << endl;  
    unique_ptr<int> sp = make_unique<int>(99); // Smart pointer  
    cout << "smart pointer apunta a: " << *sp << endl;  
    ...  
}
```

## Memoria dinámica

```
#include <iostream>  
#include <cstdlib> // malloc, free  
using namespace std;  
  
int main() {  
    // C  
    int* c = (int*)malloc(sizeof(int));  
    *c = 42;  
    printf("%d\n", *c); // Output: 42  
    free(c);  
  
    // C++  
    int* cpp = new int(42);  
    cout << *cpp << endl; // Output: 42  
    delete cpp;  
}
```

- *Range-Based for loop* con referencias

```
for (auto val : vec) { } // Copia por cada elemento, no modifica el vector  
for (auto &val : vec) { } // Referencia por cada elemento, se puede modificar  
for (const auto &val : vec) { } // No se pueden modificar los elementos
```

Cuando usamos `auto`, le decimos al compilador que deduzca automáticamente el tipo de los elementos

## ¿Qué son?

Conceptualmente, una clase es un molde o plantilla que define cómo son y qué hacen los objetos

## ¿Por qué nos interesa estudiarlas?

Son una estructura fundamental para la POO, pues permite definir un tipo de dato personalizado que combina atributos y métodos encapsulados en una sola entidad.

## Además de atributos y métodos, ¿qué más tienen?

Constructor: se llama automáticamente al crear un objeto de la clase. Inician los atributos y aseguran que el objeto empiece en un estado válido. Pueden sobrecargarse.

Destructor: se ejecuta cuando un objeto se destruye (al salir del scope o al eliminar el objeto). Libera recursos asignados, evitando fugas de memoria.

## ¿Qué otras características tienen?

Los métodos y atributos tienen visibilidad (public, protected, private).

Permite crear clases basadas en otras, permitiendo que se puedan compartir o extender comportamientos. Facilita reutilización y modularización

### Structs en C

```
struct Persona {  
    char nombre[50];  
    int edad;  
};
```

### Structs en C++

```
struct Persona {  
    std::string nombre;  
    int edad;  
    void saludar() {  
        std::cout << "Hola" << std::endl; }  
};
```

Miembros solo public

### Clases en C++

```
class Persona {  
    private:  
        int edad;  
    public:  
        std::string nombre;  
        Persona(std::string nom, int e) {  
            nombre = nom;  
            edad = e; }  
        void saludar() {  
            std::cout << "Hola" << std::endl; }  
};
```

Distinta visibilidad,  
constructores,  
destructores

1. **Default constructor**
2. **Parameterized constructor**
3. **Copy constructor**
4. **Move Constructor** (No vamos a usarlo en este curso)

**Default:** aquel que no recibe parámetros. C++ genera automáticamente un constructor por defecto para nuestra clase si no hemos definimos ningún constructor. Si definimos cualquier otro constructor, el compilador ya no creará el constructor por defecto.

```
class Persona {  
public:  
    Persona() {} // Equivale a un constructor  
};                por defecto
```

**Parameterized:** puede tomar parámetros para permitir la inicialización personalizada de los atributos del objeto al momento de crearlo. Podemos crear un constructor por default explícito

```
class Persona {  
private:  
    std::string nombre;  
    int edad;  
public:  
    Persona(const std::string& n, int e){//parámetros  
        nombre = n;  
        edad = e;  
        std::cout << "Creando a " << nombre  
        << " de " << edad << " años\n";};};
```

## Copy constructor

- Constructor que permite crear un nuevo objeto como copia exacta de otro objeto existente de la misma clase
- Sintaxis:

```
className (const className &obj)
```

## Copy Assignment

- Operador que permite copiar el contenido de un objeto a otro objeto ya existente de la misma clase. No es un constructor en sí.
- Sintaxis:

```
className& operator=(const className &obj)
```



# Copy constructor vs Copy assignment

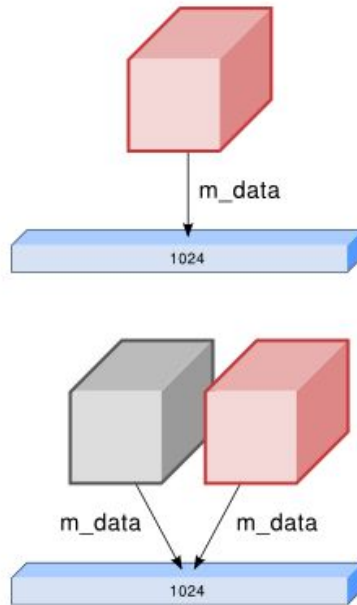
---

- Similitudes:
  - Ambos se utilizan para inicializar un objeto a partir de otro
  - Por default realizan *shallow copy*
  - Se debe sobrecargar el constructor o el operador para realizar *deep copy*
  - Permiten prohibir la copia

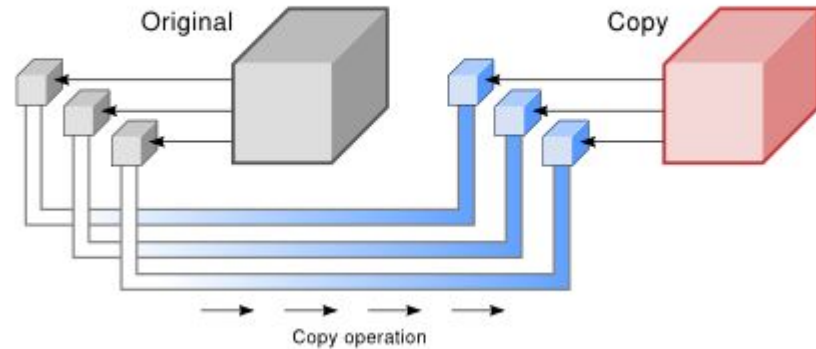
```
className (const className &obj) = delete  
className& operator=(const className &obj) = delete
```

# Copy constructor vs Copy assignment

## Shallow Copy



## Deep Copy



# Copy constructor vs Copy assignment

- Diferencias:

Copy constructor	Copy assignment
Se llama cuando un nuevo objeto se crea a partir de un objeto existente, como una copia del objeto existente	Se llama cuando a un objeto ya inicializado se le asigna un nuevo valor a partir de otro objeto existente
Crea un bloque de memoria separado para el nuevo objeto	No lo hace
Es un constructor sobrecargado	Es un operador a nivel de bits

# Copy constructor vs Copy assignment

---


- Cuando se llama al Copy Constructor?
  - Un objeto de la clase es retornado por valor
  - Un objeto se pasa por argumentos a una función
  - Un objeto es construido basándose en otro de la misma clase
  - Cuando el compilador genera un objeto temporal

# Copy constructor vs Copy assignment

Cuál es la salida del siguiente código?

```
class Test {  
public:  
    Test(const Test& t)  
    {  
        cout << "Copy constructor called " << endl;  
        return;  
    }  
  
    Test& operator=(const Test& t)  
    {  
        cout << "Assignment operator called " << endl;  
        return *this;  
    }  
};
```

`this` apunta a la instancia del objeto que llama a la función

```
int main()  
{  
    Test t1, t2;  Default constructor  
    t2 = t1;  
    Test t3 = t1;  
    Test t4 (t1);  
    return EXIT_SUCCESS;  
}
```

Output:

Assignment operator called  
Copy constructor called  
Copy constructor called

Son una forma de inicializar los miembros de un objeto antes de que se ejecute el cuerpo del constructor. Es más eficiente porque inicializa directamente los miembros en lugar de construirlos por defecto y luego asignarles valor.

Las constantes y referencias solo pueden ser inicializadas, no asignadas. Por eso no funcionan sin initialization list.

```
class Persona {  
    private:  
        const int id;  
        std::string nombre;  
    public:  
        Persona(int i, std::string n): id(i), nombre(n) {}  
        void mostrar() {std::cout << "ID: " << id << ", Nombre: " << nombre << std::endl;}  
};  
  
int main() {  
    Persona p(1, "Tiziano");  
    p.mostrar();}
```

# Constante sin initialization list. ¿Va a funcionar el código?

```
class Punto {  
private:  
    const int x;  
    const int y;  
public:  
    Punto(int valorX, int valorY) {  
        x = valorX;  
        y = valorY;  
    }  
};  
int main() {  
    Punto punto(3, 4);  
    return 0;  
}
```

```
(base) tiziano@tiziano:~/Documents/ACSO/clase_preparacion$ g++ main.cpp -o programa  
main.cpp: In constructor 'Punto::Punto(int, int)':  
main.cpp:8:5: error: uninitialized const member in 'const int' [-fpermissive]  
    8 |     Punto(int valorX, int valorY) {  
      |     ^~~~~~  
main.cpp:3:15: note: 'const int Punto::x' should be initialized  
    3 |     const int x; // Miembro constante  
      |     ^  
main.cpp:8:5: error: uninitialized const member in 'const int' [-fpermissive]  
    8 |     Punto(int valorX, int valorY) {  
      |     ^~~~~~  
main.cpp:4:15: note: 'const int Punto::y' should be initialized  
    4 |     const int y;  
      |     ^  
main.cpp:9:11: error: assignment of read-only member 'Punto::x'  
    9 |         x = valorX; // Error: 'x' no se puede asignar aquí porque es const  
      |         ^~~~~~  
main.cpp:10:11: error: assignment of read-only member 'Punto::y'  
   10 |         y = valorY; // Error: 'y' no se puede asignar aquí porque es const  
      |         ^~~~~~
```

# Referencia sin initialization list. ¿Va a funcionar el código?

```
class Wrapper {  
private:  
    int &ref;  
public:  
    Wrapper(int &valor) {  
        ref = valor;  
    }  
};  
  
int main() {  
    int x = 5;  
    Wrapper w(x);  
    return 0;  
}
```

```
(base) tiziano@tiziano:~/Documents/ACSO/clase_preparacion$ g++ main.cpp -o programa  
main.cpp: In constructor 'Wrapper::Wrapper(int&)':  
main.cpp:7:5: error: uninitialized reference member in 'int&' [-fpermissive]  
    7 |     Wrapper(int &valor) {  
      |     ^~~~~~  
main.cpp:3:10: note: 'int& Wrapper::ref' should be initialized  
    3 |     int &ref; // Referencia a un entero  
      |     ^~~
```



Se llama automáticamente al destruir el objeto. Se define como `~Clase()`. Especialmente importante para casos como los siguientes:

- Cuando se asigna memoria dinámica
- Cuando la clase maneja recursos como archivos o sockets que debe asegurarse de cerrarlos adecuadamente.

Si no se define un destructor, el compilador proporcionará uno automáticamente que no realiza ninguna acción.

```
class Persona {  
    public:  
        ~Persona() {  
            std::cout << "Destructor llamado" << std::endl;  
        }  
};  
  
int main() {  
    Persona p;  
} // El destructor se llama al salir del scope
```

**new:** es un operador que se utiliza para asignar memoria dinámica para un objeto o un arreglo de objetos. Su uso implica la invocación del constructor de la clase. Devuelve un puntero al objeto recién creado.

**delete:** es un operador que libera la memoria asignada por new y llama al destructor del objeto.

```
MyClass* obj = new MyClass(); // Se llama al constructor  
delete obj; // Se llama al destructor y se libera la memoria
```

## ¿Y qué ocurre con malloc y free?

Son funciones de C y trabajan con tipos de datos primitivos. No proporcionan ningún mecanismo de inicialización o destrucción de objetos. `delete` espera un puntero que fue creado con `new`, y `free` espera un puntero que fue creado con `malloc`.

```
MyClass* obj = (MyClass*)malloc(sizeof(MyClass)); // No se llama al constructor
```

Permite que una función sea llamada sin pasar uno o más argumentos finales.

# Python

```
def add(a: int, b: int = 1129, c: int = 195) -> int:  
    return a + b + c
```

// C++

```
int add(int a, int b = 1129, int c = 195) {  
    return a + b + c;  
}
```

```
int main(void) {  
    cout << add(13, 14, 15) << endl;  
    cout << add(13, 14) << endl;  
    cout << add(13) << endl;  
    // cout << add() << endl; Compilation error  
    return EXIT_SUCCESS;  
}  
  
42  
222  
1337
```

# Overloading Functions & Operators

Permite especificar más de una definición para una **función** u **operador** dentro del mismo *scope*.

```
int add(int a, int b) {  
    return a + b;  
}
```

```
double add(double a, double b) {  
    return a + b;  
}
```

```
string add(const string &a, const string &b) {  
    return to_string(stoi(a) + stoi(b));  
}
```

```
int main(void) {  
    cout << add(1, 3) << endl;  
    cout << add(1.53, 1.61) <<  
endl;  
    cout << add("1", "-5") << endl;  
    return EXIT_SUCCESS;  
}  
4  
3.14  
-4
```

# Overloading Functions & Operators

Permite especificar más de una definición para una **función** u **operador** dentro del mismo *scope*.

```
class ComplexNumber {
    private:
        double real, imaginary;
    public:
        ComplexNumber(double r, double i) : real(r), imaginary(i) {}
        ComplexNumber operator+(const ComplexNumber &other) const {
            return ComplexNumber(real + other.real,
                                imaginary + other.imaginary);
        }
        friend ostream &operator<<(ostream &os, const ComplexNumber &cn) {
            return os << cn.real
                << (cn.imaginary >= 0 ? " + " : " - ")
                << abs(cn.imaginary) << "j";
        }
};
```

```
int main(void) {
    ComplexNumber c1(1.9, 2.1);
    ComplexNumber c2(2.1, -0.1);
    ComplexNumber c3 = c1 + c2;
    // ComplexNumber c4 = c1 - c2;
    // Compilation error
    cout << c1 << endl;
    cout << c2 << endl;
    cout << c3 << endl;
    return EXIT_SUCCESS;
}

1.9 + 2.1j
2.1 - 0.1j
4 + 2j
```

# Default Arguments + Overloading Functions

```
int add(int a = 1, int b = 2) {  
    return a + b;  
}
```

```
double add(double a = 1.1, double b = 2.2) {  
    return a + b;  
}
```

```
int main(void) {  
    cout << add(3, 4) << endl;           7  
    cout << add(3.3, 4.4) << endl;       7.7  
    cout << add(3) << endl;              5  
    cout << add(3.3) << endl;            5.5  
    // cout << add() << endl;  
    // Compilation error: call of overloaded 'add()' is ambiguous  
    return EXIT_SUCCESS;  
}
```

Las **templates** de C++ permiten definir **funciones** y **clases** genéricas que pueden operar con distintos tipos, sin necesidad de reescribir el código para cada uno.

```
template<typename identifier> declaration;
```

```
template<class identifier> declaration;
```



[Bjarne Stroustrup](#)

# Function Templates

```
/* Overloading Functions */
int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

string add(const string &a,
           const string &b) {
    return a + b;
}

ComplexNumber add(ComplexNumber a,
                  ComplexNumber b) {
    return a + b;
}
```

```
/* Templates */
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main(void) {
    cout << add(1, 2) << endl;endl;
    cout << add(1.1, 2.2) << endl;<< endl;
    cout << add(string("Hello, "), string("World!")) << endl;
    cout << add(ComplexNumber(1, 2), ComplexNumber(3, 4)) << endl;
    return EXIT_SUCCESS;      ComplexNumber(3, 4)) << endl;
} return EXIT_SUCCESS;

3
3.3
Hello, World!
4 + 6j
```



```
template <typename T1, typename T2>
class Pair {
    private:
        T1 first;
        T2 second;
    public:
        Pair(T1 f, T2 s) : first(f), second(s) {}
        T1 getFirst() const { return first; }
        T2 getSecond() const { return second; }
        void display() const {
            cout << "(" << first
                 << ", " << second
                 << ")" << endl;
        }
};
```

```
int main(void) {
    Pair<int, double> intDoublePair(42, 3.14);
    Pair<string, string> stringPair("Hello", "World");
    Pair stringComplexPair("Templates can't be real!",
                           ComplexNumber(1, 2));

    intDoublePair.display();
    stringPair.display();
    stringComplexPair.display();
    return EXIT_SUCCESS;
}

(42, 3.14)
(Hello, World)
(Templates can't be real!, 1 + 2j)
```

La **STL (Standard Template Library)** provee un amplio rango de **function** y **class templates** que implementan los **algoritmos** y **estructuras de datos** más usados (entre muchas otras *features*), que están disponibles en **standard C++**.

Cada elemento de la C++ Standard Library está definido en un **header**.

C++ library headers				
<code>&lt;algorithm&gt;</code>	<code>&lt;iomanip&gt;</code>	<code>&lt;list&gt;</code>	<code>&lt;ostream&gt;</code>	<code>&lt;streambuf&gt;</code>
<code>&lt;bitset&gt;</code>	<code>&lt;ios&gt;</code>	<code>&lt;locale&gt;</code>	<code>&lt;queue&gt;</code>	<code>&lt;string&gt;</code>
<code>&lt;complex&gt;</code>	<code>&lt;iosfwd&gt;</code>	<code>&lt;map&gt;</code>	<code>&lt;set&gt;</code>	<code>&lt;typeinfo&gt;</code>
<code>&lt;deque&gt;</code>	<code>&lt;iostream&gt;</code>	<code>&lt;memory&gt;</code>	<code>&lt;sstream&gt;</code>	<code>&lt;utility&gt;</code>
<code>&lt;exception&gt;</code>	<code>&lt;istream&gt;</code>	<code>&lt;new&gt;</code>	<code>&lt;stack&gt;</code>	<code>&lt;valarray&gt;</code>
<code>&lt;fstream&gt;</code>	<code>&lt;iterator&gt;</code>	<code>&lt;numeric&gt;</code>	<code>&lt;stdexcept&gt;</code>	<code>&lt;vector&gt;</code>
<code>&lt;functional&gt;</code>	<code>&lt;limits&gt;</code>			

La **Containers Library** es una colección de templates de clases y algoritmos que permiten implementar **estructuras de datos**. Existen tres clases de contenedores:

- Sequence Containers.

`std::array`  
`std::vector`  
`std::deque`  
`std::forward_list`  
`std::list`

- Ordered/Unordered Associative Containers.

`std::set`  
`std::map`  
`std::unordered_set`  
`std::unordered_map`

- Container Adaptors.

`std::stack`  
`std::queue`  
`std::priority_queue`

# Sequence Containers

---

```
#include <array>
```

```
array<string, 3> courses = {  
    "ACSO",  
    "ML",  
    "F2"  
};
```

```
#include <vector>
```

```
vector<ComplexNumber> complexNumbers;  
int n;  
cin >> n;  
complexNumbers.reserve(n);  
for (int i = 0; i < n; ++i) {  
    double real, imaginary;  
    cin >> real >> imaginary;  
    complexNumbers.emplace_back(real, imaginary);  
}
```

# For Loops sobre Containers

```
vector<int> v = {1, 2, 3, 4, 5};
```

```
// Index-based for loop
```

```
for (int i = 0; i < v.size(); ++i) {  
    cout << v[i] << " ";  
}
```

```
cout << endl;
```

```
// Iterator-based for loop
```

```
for (auto it = v.begin(); it != v.end(); ++it) {  
    cout << *it << " ";  
}
```

```
cout << endl;
```

```
// Range-based for loop
```

```
for (const auto &elem : v) {  
    cout << elem << " ";  
}
```

```
cout << endl;
```

```
1 2 3 4 5
```

```
1 2 3 4 5
```

```
1 2 3 4 5
```

# For Loops sobre Containers

```
vector<int> v = {1, 2, 3, 4, 5};
```

```
for (auto elem : v) {  
    elem *= 2;  
}
```

```
printVector(v);
```

1 2 3 4 5

2 4 6 8 10

```
for (auto &elem : v) {  
    elem *= 2;  
}
```

```
printVector(v);
```

# Ordered/Unordered Associative Containers

```
#include <unordered_map>
```

```
unordered_map<string, double> stockPrices = {  
    {"AAPL", 150.25},  
    {"GOOGL", 2800.50},  
    {"AMZN", 3400.75},  
    {"MSFT", 299.99}  
};  
  
for (const auto &[ticker, price] : stockPrices) {  
    cout << ticker << ": $" << price << endl;  
}
```

```
MSFT: $299.99  
AMZN: $3400.75  
GOOGL: $2800.5  
AAPL: $150.25
```

# Container Adaptors

```
#include <queue>

queue<int> q;
for (int i = 0; i < 10; ++i) {
    q.push(i);
}
while (!q.empty()) {
    cout << q.front() << endl;
    q.pop();
}
```

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

```
#include <stack>

stack<int> s;
for (int i = 0; i < 10; ++i) {
    s.push(i);
}
while (!s.empty()) {
    cout << s.top() << endl;
    s.pop();
}
```

9  
8  
7  
6  
5  
4  
3  
2  
1  
0



La **Algorithms Library** define template de funciones que operan sobre contenedores e implementan **algoritmos** de ordenamiento, búsqueda, etc.

```
#include <algorithm>
```

<code>sort ()</code>	Sort the elements of the container.
<code>copy ()</code>	Copy elements within a given range.
<code>move ()</code>	Move the given range of elements.
<code>swap ()</code>	Exchange values of two objects.
<code>merge ()</code>	Merge sorted ranges.
<code>replace ()</code>	Replace the value of an element.
<code>remove ()</code>	Remove an element.

```
vector<int> v = {8, 3, 2, 5, 9, 1, 4, 6, 7};
```

```
auto it = find(v.begin(), v.end(), 2);
```

2

```
cout << *it << endl;
```

2

```
cout << it - v.begin() << endl;
```

2

```
cout << distance(v.begin(), it) << endl;
```

1 2 3 4 5 6 7 8 9

9 8 7 6 5 4 3 2 1

```
sort(v.begin(), v.end());
```

5 6 7 8 9 4 3 2 1

```
printVector(v);
```

```
sort(v.begin(), v.end(), greater<int>());
```

```
printVector(v);
```

```
sort(v.begin(), v.end() - v.size() / 2);
```

```
printVector(v);
```

```
vector<int> v = {8, 3, 2, 5, 9, 1, 4, 6, 7};
```

```
auto ti = max_element(v.begin(), v.end());
```

```
cout << *ti << endl;
```

```
reverse(v.begin(), v.end());
```

```
printVector(v);
```

```
sort(v.begin(), v.end());
```

```
printVector(v);
```

```
shuffle(v.begin(), v.end(), default_random_engine());
```

```
printVector(v);
```

9

7 6 4 1 9 5 2 3 8

1 2 3 4 5 6 7 8 9

4 1 5 8 6 2 9 7 3

```
vector<int> v = {8, 3, 2, 5, 9, 1, 4, 6, 7};
```

```
sort(v.begin(), v.end());
```

```
printVector(v);
```

1 2 3 4 5 6 7 8 9

4

5

```
auto it = lower_bound(v.begin(), v.end(), 5);
```

```
cout << distance(v.begin(), it) << endl;
```

```
it = upper_bound(v.begin(), v.end(), 5);
```

```
cout << distance(v.begin(), it) << endl;
```

Una **expresión lambda** permite construir un *closure*, esto es, una función anónima capaz de capturar variables en *scope*.

```
[captures] (params) { body }
```

```
[captures] <tparams> (params) { body }
```



<code>captures</code>	Define las variables externas que son accesibles desde el <i>body</i> de la función lambda <code>[capture-default, capture-list] -&gt; [&amp; o =, var1, var2, ...]</code>
<code>params</code>	Es la lista de parámetros de la función
<code>tparams</code>	Define los parámetros de template que pueden ser utilizados en una lambda genérica
<code>body</code>	Se trata del cuerpo de la función

# Lambda Expressions

```
string tincho = "Tincho";
auto logGreeting = [&](const string &name) {
    cout << name << " was greeted by " << tincho << endl;
    tincho += "!";
};
logGreeting("Tizi");
logGreeting("Juan");
logGreeting("Robbie");

string acso = "ACSO is the best";
auto print = [=]() mutable {
    acso += "!";
    cout << acso << endl;
};
print();
print();
print();
cout << acso << endl;
```

Tizi was greeted by Tincho  
Juan was greeted by Tincho!  
Robbie was greeted by Tincho!!

```
auto duplicate = []<typename T>(T x) {
    return x + x;
};
auto duplicate = [](auto x) {
    return x + x;
};
cout << duplicate(5) << endl;
cout << duplicate(string("Hello ")) << endl;
```

ACSO is the best!  
ACSO is the best!!  
ACSO is the best!!!  
ACSO is the best  
Hello Hello

10

# Lambda Expressions

```
vector<ComplexNumber> complexNumbers = {  
    ComplexNumber(1.0, 1.0),  
    ComplexNumber(1.0, -2.0),  
    ComplexNumber(-3.0, 1.0),  
    ComplexNumber(2.0, 3.0),  
    ComplexNumber(0.0, -1.0)  
};  
  
sort(complexNumbers.begin(), complexNumbers.end(),  
    [](const ComplexNumber &lhs, const ComplexNumber &rhs) {  
        auto squaredComplexNorm = [](const ComplexNumber &cn) {  
            return cn.getReal() * cn.getReal() +  
                   cn.getImaginary() * cn.getImaginary();  
        };  
        return squaredComplexNorm(lhs) < squaredComplexNorm(rhs);  
    });  
  
for_each(complexNumbers.begin(), complexNumbers.end(),  
    [](const auto &cn) { cout << cn << endl; });
```

0 - 1j  
1 + 1j  
1 - 2j  
-3 + 1j  
2 + 3j

C++ incluye soporte para **threads**, operaciones atómicas, exclusión mutua, *condition variables* y *futures*. Los threads le permiten a los programas ejecutar a través de varios núcleos del procesador.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
```

```
using namespace std;
```

```
mutex mtx;
condition_variable cv;
bool ready = false;
```

```
Main: Notifying worker thread...
Thread: Working!
```

```
void worker() {
    unique_lock<mutex> lock(mtx);
    cv.wait(lock, [] { return ready; });
    cout << "Thread: Working!\n";
}

int main() {
    thread t(worker);
    {
        lock_guard<mutex> lock(mtx);
        ready = true;
        cout << "Main: Notifying worker thread...\n";
    }
    cv.notify_one();
    t.join();
    return EXIT_SUCCESS;
}
```



**¿Preguntas?**