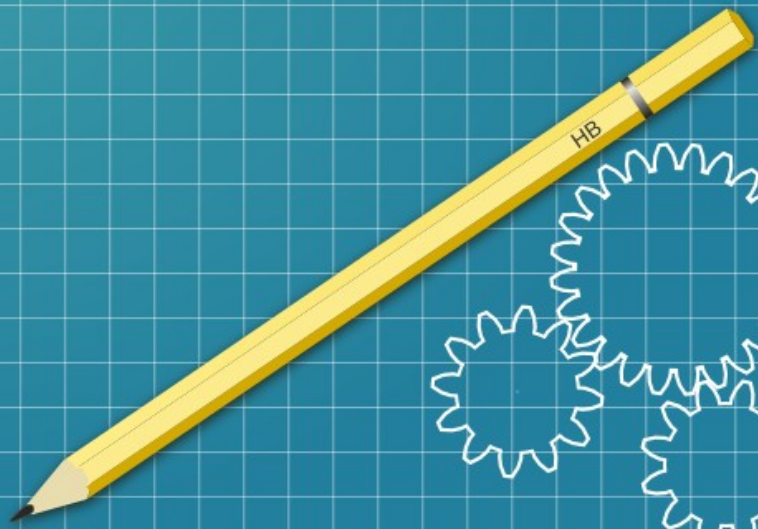
Introducción al lenguaje C++

```
#include<stdio.h>

int main(){
  int a = 10;
  int b;
  b ~= a;
  return 0;
}
```



Palomeque Nestor Levi

Introducción al lenguaje de programación C++

C++ es un lenguaje de programación de propósito general que fue creado como una extensión del lenguaje C por Bjarne Stroustrup en los años 80. Combina programación estructurada y orientada a objetos. A lo largo de los años, han surgido varios estándares, desde C++98 hasta los más recientes como C++17, C++20, y C++23. Estos estándares añaden nuevas características y mejoras al lenguaje.

Entrada/salida en C++

En C++, la entrada y salida estándar se manejan a través de la biblioteca <iostream>, que proporciona flujos de datos para interactuar con el usuario a través de la consola. Los dos **objetos** más comunes son **std::cout** y **std::cin**.

```
#include <iostream>


int main() {
    std::string nombre;
    std::cout << "Ingresa tu nombre: ";
    std::cin >> nombre;
    std::cout << "Hola, " << nombre << "!" << std::endl;
    return 0;
}
```

Cuando usas **std::cin**, el operador de extracción (>>) lee datos desde la entrada estándar (teclado) y los convierte al tipo de dato especificado en la variable a la que se asigna.

De manera similar, cuando usas **std::cout << variable;**, el operador de inserción (<<) determina cómo convertir el tipo de variable en una secuencia de caracteres que se puede mostrar en la consola.

Tipo de dato **bool**

El tipo **bool** representa valores booleanos y puede ser **true** o **false**. También se puede asignar un número, típicamente 1 o 0, teniendo en cuenta que sólo el 0 se considera **false**, cualquier otro número será **true.**

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

int main() {
    int numero;
    cout << "Ingresa un número: ";
    cin >> numero;

   // Evaluar si el número es positivo
    bool esPositivo;
    if (numero > 0)  esPositivo = true;
    else esPositivo = false;

    cout << "El número es positivo: " << esPositivo << endl;
    return 0;
}
```

string en C++

En C++, tienes la clase std::string que proporciona una manera más segura y conveniente de manejar cadenas.

Dicha clase maneja la memoria automáticamente, por lo que no tienes que preocuparte por desbordamientos ni por liberar memoria. También posee operadores sobrecargados: puedes usar operadores como + para concatenar cadenas, == para comparar, etc. Esto hace que el código sea más legible y menos propenso a errores. Ofrece muchas funciones útiles como **size()**, **length()**, **substr()**, **find()**, y muchas más, que facilitan la manipulación de cadenas. Se puede acceder a caracteres individuales con [ ] o con la función **at()**. Respecto a la seguridad maneja internamente los detalles de los punteros y la memoria, lo que reduce los errores comunes asociados con las cadenas de C.

```cpp
#include <iostream>
#include <string>

int main() {
    std::string cadena1 = "Hola, ";
    std::string cadena2 = "mundo!";

    // Concatenar usando el operador +
    std::string resultado = cadena1 + cadena2;

    std::cout << "Resultado: " << resultado << std::endl;  // Imprime "Hola, mundo!"

    return 0;
}
```

vector en C++

En C++, la clase plantilla **std::vector** es un contenedor de la biblioteca estándar que ofrece una colección dinámica de elementos. A diferencia de los arreglos tradicionales, los vectores pueden cambiar su tamaño durante la ejecución del programa. Esto significa que puedes añadir o eliminar elementos sin preocuparte por la gestión manual de la memoria. Posee funciones como push_back(), pop_back(), size(), empty(), at(), etc. También se puede acceder a un elementos mediante el operador [ ].

```cpp
#include <iostream>
#include <vector> // Necesario para usar std::vector

int main() {
    std::vector<int> numeros;   // Declarar un vector de enteros

    numeros.push_back(10);       // Añadir elementos al vector
    numeros.push_back(20);
    numeros.push_back(30);

    std::cout << "Elementos del vector: ";     // Mostrar los elementos del vector
    for (int i = 0; i < numeros.size(); ++i) {
        std::cout << numeros[i] << " ";  // Acceso mediante operador []
    }
    std::cout << std::endl;

    numeros[1] = 50; // Modifica el segundo elemento (índice 1)
    std::cout << "Elemento modificado: " << numeros[1] << std::endl;

    numeros.pop_back();        // Eliminar el último elemento
    std::cout << "Tamaño del vector después de pop_back: " << numeros.size() << std::endl;
    return 0;
}
```

Referencias y parámetros por referencia

Una referencia en C++ es un alias para una variable existente. Se debe inicializarse en su definición y no puede reasignarse como alias de otra variable. Una vez que una referencia se ha establecido para una variable, puedes usar ese nuevo nombre para acceder o modificar el valor de la variable original.

Se define una referencia usando el símbolo **&** después del tipo de dato, por ejemplo:

```
int original = 10;
int& referencia = original;
```

En este caso, ***referencia*** es una referencia a ***original***. Cualquier operación realizada sobre ***referencia*** afectará a ***original***, y viceversa.

A continuación otro ejemplo de cómo funcionan las referencias en parámetros de funciones:

```
#include <iostream>

void modificarValor(int& ref) {
    ref = 20; // Modifica el valor de la variable original a través de la referencia
}

int main() {
    int numero = 10;
    std::cout << "Antes de modificar: " << numero << std::endl;

    modificarValor(numero); // Llama a la función pasando la variable por referencia

    std::cout << "Después de modificar: " << numero << std::endl;
    return 0;
}
```

Aquí **int& ref** en la función **modificarValor** es una referencia a un entero. Al pasar **numero** a la función, se pasa como referencia, no como una copia. Dentro de la función, al cambiar el valor de **ref**, el valor de **numero** también cambia.

Las referencias son más seguras y fáciles de usar que los punteros. Una referencia siempre debe referirse a una variable válida y no puede ser nula. Una vez inicializada, una referencia no puede cambiar para referirse a otra variable. A diferencia con los punteros que si pueden ser nulos, requieren manejo explícito de la memoria para evitar errores como el uso de punteros no inicializados o el acceso a memoria fuera de límites. También los punteros pueden ser reasignados para apuntar a diferentes variables.

Funciones inline

Las funciones inline sugieren al compilador que inserte el código de la función en lugar de hacer una llamada normal a la función, para mejorar el rendimiento.

Ejemplo: definir una función inline para calcular el cuadrado de un número.

```
#include <iostream>

inline int cuadrado(int x) {
    return x * x;
}
int main() {
    int numero = 4;
    std::cout << "El cuadrado de " << numero << " es " << cuadrado(numero) << std::endl;
    return 0;
}
```

Las funciones inline y las macros son dos mecanismos que se pueden usar en C++ para realizar sustituciones de código en el momento de la compilación, pero tienen diferencias significativas en su funcionamiento, ventajas y desventajas.

Macros:

- Las macros se expanden directamente en el código fuente durante la fase de preprocesamiento. No se realiza ninguna comprobación de tipo ni se lleva a cabo ninguna verificación en tiempo de compilación. Las macros son simples sustituciones de texto.
- Las macros no tienen un ámbito o encapsulamiento. Pueden llevar a efectos secundarios inesperados si no se manejan cuidadosamente.

Función inline:

- El compilador intenta insertar el código de la función inline directamente en el lugar de la llamada para evitar la sobrecarga de llamada a funciones. Las funciones inline tienen verificación de tipos y todas las reglas de las funciones normales se aplican, lo que ayuda a evitar errores.
- Las funciones inline tienen un ámbito definido, lo que ayuda a evitar efectos secundarios inesperados.

Sobrecarga de funciones

La sobrecarga de funciones en C++ permite definir múltiples funciones con el mismo nombre, siempre y cuando tengan diferentes listas de parámetros. La sobrecarga de funciones en C++ se basa en la lista de parámetros de la función, es decir, el número, tipo y orden de los parámetros.

```cpp
#include <iostream>

int sumar(int a, int b) {
    return a + b;
}

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

int main() {
    std::cout << "Suma de enteros: " << sumar(2, 3) << std::endl;
    std::cout << "Suma de decimales: " << sumar(2.5, 3.1) << std::endl;
    return 0;
}
```

Plantilla de funciones

Las plantillas de funciones permiten crear funciones genéricas que pueden trabajar con cualquier tipo de datos. En lugar de especificar un tipo concreto para los parámetros de una función, se usa un tipo de plantilla que se determina en tiempo de compilación en función del tipo de argumento pasado a la función.

Por ejemplo: crear una función que permita intercambiar los valores de dos variables, sin importar el tipo de dato que tengan.

```cpp
#include <iostream>

// Plantilla de función para intercambiar dos valores
template <typename T>
void intercambiar(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    std::cout << "Antes del intercambio: x = " << x << ", y = " << y << std::endl;
    intercambiar(x, y);  // Intercambia dos enteros
    std::cout << "Después del intercambio: x = " << x << ", y = " << y << std::endl;

    char c1 = 'A', c2 = 'B';
    std::cout << "Antes del intercambio: c1 = " << c1 << ", c2 = " << c2 << std::endl;
    intercambiar(c1, c2);  // Intercambia dos caracteres
    std::cout << "Después del intercambio: c1 = " << c1 << ", c2 = " << c2 << std::endl;
    return 0;
}
```

También podemos definir una función de plantilla que tome dos parámetros de tipo diferentes:

```cpp
#include <iostream>

// Plantilla de función con dos parámetros de tipo
template <typename T1, typename T2>
void imprimirDoble(T1 valor1, T2 valor2) {
    std::cout << "Valor 1: " << valor1 << ", Valor 2: " << valor2 << std::endl;
}

int main() {
    imprimirDoble(10, 3.14);    // T1 se deduce como int y T2 como double
    imprimirDoble('A', "Hola");// T1 se deduce como char y T2 como const char*
    imprimirDoble(42, true);  // T1 se deduce como int y T2 como bool

    return 0;
}
```

Asignación dinámica de memoria en C++

En C++, new se usa para asignar memoria dinámicamente, y delete para liberar esa memoria cuando ya no se necesita.

El operador **new** se utiliza para asignar memoria dinámica. Devuelve un puntero al tipo de dato solicitado.

El operador **delete** se utiliza para liberar la memoria asignada dinámicamente con **new**. Es importante liberar la memoria para evitar fugas de memoria, que ocurren cuando la memoria asignada no es liberada correctamente.

Asignación de un solo elemento

```cpp
#include <iostream>

int main() {
    int* p = new int;  // 'p' apunta a un entero en memoria
    //tambien se puede crear la memoria dinámica e inicializarza
    //int* p = new int(123);
    *p = 42;           // Asignar valor a la memoria asignada
    std::cout << "Valor: " << *p << std::endl;
    // Liberar la memoria
    delete p;    // Se libera la memoria asignada
    p = nullptr;  // Se asigna nullptr para evitar un puntero colgante
    return 0;
}
```

Asignación de arrays dinámicos

```cpp
#include <iostream>

int main() {
    int tamano;
    std::cout << "Ingresa el tamaño del array: ";
    std::cin >> tamano;

    int* array = new int[tamano];    // Asignación dinámica de memoria

    for (int i = 0; i < tamano; i++) { // Uso del array
        array[i] = i * 2;
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;

    delete[] array;              // Liberar la memoria asignada

    return 0;
}
```

Compilación en C++

Para compilar en C++, muy similar a gcc, podemos utilizar:

Compilar especificando el estándar y con optimizadores:

- `g++ -std=c++98 -O2 -o miPrograma main.cpp`

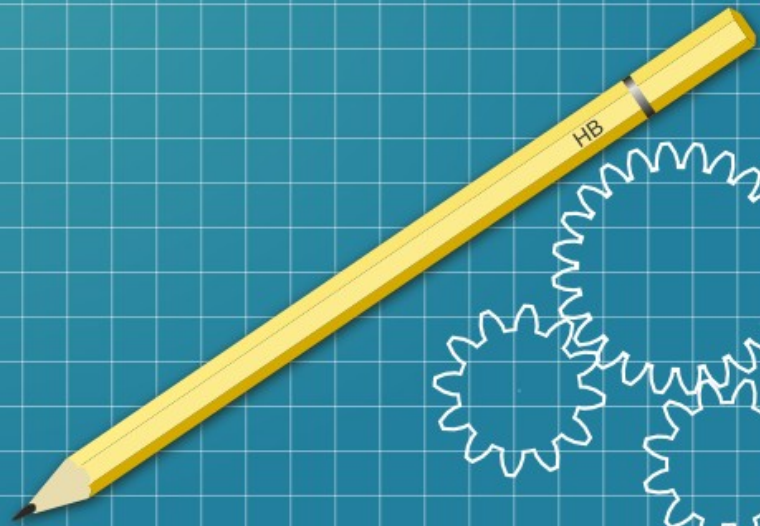
Compilar y mostrar advertencias:

- `g++ -Wall -o miPrograma main.cpp`

Compilar y enlazar varios archivos fuentes:

- `g++ -o miPrograma main.cpp funciones.cpp`





HB