# https://yt3.ggpht.com/-hYkspDLYXMQ/AAAAAAAAAAI/AAAAAAAAAAA/t8z0vauo_7A/s900-c-k-no/photo.jpghttp://intranet.epn.edu.ec/portal/images/articulos/buo_epn.jpgESCUELA POLITÉCNICA NACIONAL

ESCUELA DE FORMACIÓN DE TECNÓLOGOS

**PROGRAMACION TDSD214**

|  |  |
| --- | --- |
| ASIGNATURA: | Programación |
| PROFESOR: | Ing. Ivonne Maldonado |
| PERÍODO ACADÉMICO: | 2023-B |

# LABORATORIO - 4

|  |
| --- |
| TÍTULO:  **SENTENCIAS SIMPLES** |
|  |
| Operadores | dev-c |

**Nombre: GUERRA LOVATO JOSUÉ EDUARD**

### PROPÓSITO DE LA PRÁCTICA

Familiarizar al estudiante con el lenguaje C++.

### OBJETIVO GENERAL

Conocer un conjunto de expresiones que permiten ejecutar una determinada acción.

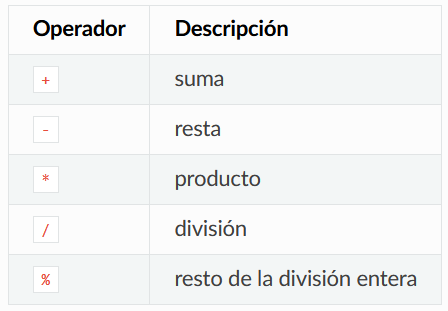
### OBJETIVOS ESPECÍFICOS

* Conocer sobre Operades Aritméticos
* Conocer sobre Operades de Asignación
* Conocer sobre Operades de iniciación
* Realizar conversión entre tipos de datos

**MARCO TEÓRICO**

**Operadores aritméticos**

Permiten realizar **operaciones aritméticas** sobre sus **operandos**.



### NOTA:

El operador aritmético **%** sólo es aplicable a datos enteros.

Por ejemplo: 23%5 es **3**.

Cada **operación** realizada con los **operandos** genera un **resultado** que se almacenará en alguna posición de memoria.

El **tipo** del valor generado y, por tanto, su **representación interna**, dependerá de los tipos de los operandos.

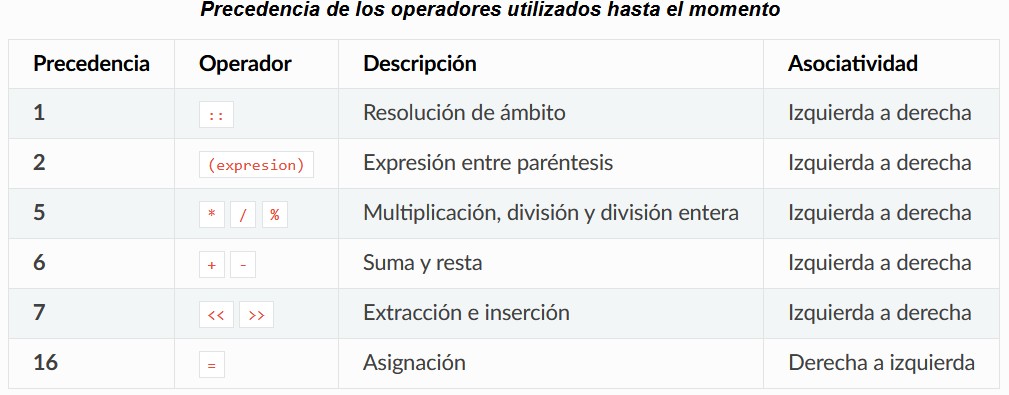
## Reglas de precedencia

Cuando se combinan diferentes operadores, el valor determinado por una expresión se calcula

**siguiendo un orden de evaluación** en función de unas **reglas de precedencia**.

En C++ y otros muchos lenguajes, se sigue el orden *normal* establecido en la **aritmética**, realizándose las operaciones **de izquierda a derecha** siguiendo el siguiente orden:

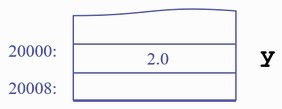
1. Se evalúan las expresiones **entre paréntesis**
2. Potencias (C++ no dispone de un **operador nativo**, pero otros lenguajes como Python sí)
3. Multiplicación, división y resto de la división entera
4. Suma y resta



La tabla completa de precedencias puede verse en este enlace: <https://en.cppreference.com/w/cpp/language/operator_precedence>

## Operadores de asignación

El operador **=** es un operador **sobrecargado**. Un operador se llama **sobrecargado** cuando dependiendo del contexto acomete distintas operaciones. En el contexto de la **asignación**, el operador **=** se utiliza para **modificar** el valor de las variables, es decir, altera el bloque de memoria asociado a una variable.

Supongamos que **y** es una variable de tipo **double** ligada a un bloque de memoria con valor **2.0**.

### Imagen que contiene Tabla Descripción generada automáticamenteNOTA:

y = y + 1.5;

El operador de asignación **no es equivalente** a la igualdad matemática: una sentencia como la anterior, y = y + 1.5;, no tiene sentido en matemáticas.

## Valores por la izquierda y valores por la derecha

Una expresión genera un **valor por la izquierda** (**Lvalue**) cuando el bloque de memoria donde se almacena el resultado obtenido es accesible desde el programa y, por tanto, persiste y podemos modificarlo posteriormente.

El nombre se debe a que estas expresiones normalmente se utilizan en el lado izquierdo del operador de asignación =. Típicamente son nombres de variables.

Por el contrario, una expresión genera un **valor por la derecha** (**Rvalue**) cuando el bloque de memoria donde será almacenado no tiene una posición específica: probablemente se almacenará temporalmente en algún registro de la CPU durante la ejecución del programa.

Una expresión **Rvalue** no puede aparecer nunca en el lado izquierdo del operador de asignación

**=**.

El operador de asignación es de los pocos operadores que se **evalúan de derecha a izquierda**.

## Operadores de asignación compuestos

En C, C++, Python, Java, etc. son muy utilizados los operadores de **asignación compuestos**.

La idea es que si tenemos una sentencia del tipo a = a OP b;, donde OP es típicamente uno de los **operadores aritméticos**, podemos abreviadamente utilizar la alternativa a OP= b;.

|  |  |  |
| --- | --- | --- |
| a += b; | // equivale : | a = a + b |
| a -= b; | // equivale : | a = a - b |
| a \*= b; | // equivale : | a = a \* b |
| a /= b; | // equivale : | a = a / b |
| a %= b; | // equivale : | a = a % b |

Debe tenerse en cuenta que estos operadores no solo permiten escribir una sentencia de una forma más abreviada. Internamente, el compilador generará instrucciones más eficientes, ya que le estamos informando de que uno de los operandos será el destino final de la expresión evaluada.

### NOTA:

**OP=** debe escribirse con los dos símbolos seguidos, sin espacios en blanco.

a + = b; // ¡ERROR!

## Operador de inicialización

El símbolo = también se usa como operador para **inicializar** las variables.

int num = 5;

El operador = es un operador **sobrecargado** y es interesante conocer la diferencia cuando se usa para **asignar** de cuando se utiliza para **inicializar**. Aunque el resultado final almacenado en la variable sea el mismo, la CPU necesitará hacer operaciones distintas para cada uno de los operadores.

La sentencia

int num = 5; // Inicialización por copia

es distinta que:

int num; // Variable sin inicializar num = 5; // Asignación del valor 5 a num

Es cierto que para el caso de los tipos de datos básicos (int, double, etc.), ambas secuencias de sentencias producen el mismo resultado, pero para tipos de datos más complejos esto no es así.

Una primera gran diferencia: **se inicializa una única vez; se asigna cuantas veces sea necesario**.

## Diferencia entre inicializar y asignar

Supongamos un problema en el cual tenemos un **contenedor** con **cubos** de colores blanco y negro (cada cubo representaría un **bit**: **0** o **1**).

Pedimos a una persona (**CPU**) que construya una **torre** (el **tipo de dato**) con cierto número de cubos sobre una mesa (la **memoria**).

Las únicas operaciones permitidas son las de una **pila**:

* + **apilar**: coger un cubo y depositarlo sobre la torre
  + **desapilar**: coger el cubo superior de la torre y dejarlo en el contenedor Arranca el programa. La mesa está vacía. Tenemos 3 sentencias.
  + 1ª sentencia: **inicialización**: formar una torre con 3 cubos blancos.

El proceso consiste en buscar un cubo blanco, cogerlo, dejarlo en cierta posición sobre la mesa (la **dirección de memoria**) e ir formando la torre.

* + 2ª sentencia: **asignación**: modificar la torre de tal forma que el cubo intermedio sea negro.

El proceso en este caso consiste en quitar sucesivamente dos cubos blancos, coger un cubo negro del contenedor y dejarlo sobre la torre y coger un cubo blanco del contenedor y dejarlo sobre la torre.

* + 3ª sentencia: **inicialización**: formar una nueva torre con 2 cubos blancos y uno negro, siendo este último el intermedio.

Las 2 torres actualmente sobre la mesa (memoria) son iguales, pero la CPU ha realizado operaciones diferentes para crearlas debido a que la primera se formó por asignación (ya existía una torre ahí) y la segunda por inicialización.

## Inicialización uniforme

**Inicialización por copia**, desde el estándar C++11, se dispone de una nueva sintaxis, conocida como **inicialización uniforme**.

int num{5};

int num{}; // Inicializa a 0

Una ventaja de esta forma de inicialización es que deduce el tipo de las constantes literales y el compilador nos avisa de un posible error.

int x = 3.3; // Posible aviso: trunca la parte decimal int num{3.3}; // Error

En el caso de la inicialización por copia, y dependiendo del compilador, tendremos un **aviso**

(**warning**).

## Sentencias de definición, inicialización y asignación

Las **sentencias de definición** introducen nuevas variables en el **ámbito** en el que son declaradas, es decir, el conjunto de sentencias situadas entre llaves { }.

{

// código int x;

// código

}

En el ejemplo que, tras la definición, el valor de la variable x es **indefinido**.

Por ello, siempre que sea posible, es conveniente **definir e inicializar** en el mismo paso.

{

// código int x = 0; int y{}; double z{};

// código

}

### NOTA:

Recuerde que con la **inicialización uniforme** la variable y se inicializa automáticamente 0 y la z

a 0.0.

Las **sentencias de asignación** evalúan la expresión situada a la derecha del operador = y asignan el resultado a la variable situada a la izquierda.

x = 2\*y;

## ¿Dónde realizar las sentencias de definición e inicialización?

C++ usa **tipado estático** lo que, entre otras cosas, implica que una variable tiene que estar

**declarada antes de poder utilizarla** en una expresión.

Para el caso de variables correspondientes a tipos nativos, no existe una restricción respecto al lugar en un programa donde realizar las sentencias de definición e inicialización de variables.

Sin embargo, por legibilidad y eficiencia lo lógico es hacerlo lo más cerca posible de la primera expresión en la que vayan a ser utilizadas las variables.

* + Damos información contextual a las líneas siguientes, como, por ejemplo, el tipo de dato involucrado en una operación.
  + Podemos estar seguros de que esa variable no afecta al programa aguas arriba.
  + Es más difícil que olvidemos inicializarla, pues la vamos a usar de forma inmediata.

## Conversión entre tipos de datos

El hecho de que C++ sea un lenguaje **estáticamente tipado** significa que los tipos asignados a las variables se comprueban en **tiempo de compilación**.

Así, el compilador verifica que las operaciones que el programador ha escrito en el código fuente están soportadas por el lenguaje. Si existe alguna incompatibilidad, el compilador avisará con un mensaje de error.

Difícilmente el compilador podría realizar esta tarea si previamente no hemos declarado los tipos de las variables antes de ser utilizadas.

No obstante, los lenguajes estáticamente tipados sí que permiten **mezclar** tipos de dato en las expresiones a pesar de que sus representaciones internas sean diferentes.

* **Conversión implícita**: El compilador se encarga de forma automática de adaptar las representaciones para poder efectuar los cálculos de forma correcta.
* **Conversión explícita**: El programador escribe explícitamente en el código fuente cómo quiere que se haga la conversión entre las representaciones.

## Conversión implícita

**Inicialización directa y asignaciones**

C++ evalúa la expresión a la derecha del operador = y el resultado se convierte implícitamente al tipo de dato situado a su izquierda.

* La conversión de una expresión con un valor de tipo real a una variable de tipo entero se trunca al entero resultante de eliminar la parte fraccionaria.
* La conversión de una expresión con un valor de tipo entero a una variable de tipo real se realizará según la norma [IEEE 754.](https://es.wikipedia.org/wiki/IEEE_754) Puede conllevar la pérdida de precisión en números grandes para los float, no así para los double, que tienen suficientes bits (una razón más para usar siempre double).
* Una expresión no boolena solo se asigna a false si el valor es 0. En caso contrario, el resultado de la asignación es true.

## Operaciones en expresiones

Cuando aparece una expresión con distintos tipos de datos mezclados, C++ **promueve** todos los tipos a un único tipo, el que tiene más **jerarquía** de entre los presentes. De esta forma, la operación (aritmética o lógica) se realiza según la representación del tipo de mayor jerarquía.

Para los tipos de datos, la jerarquía de menor a mayor en C++ es:

bool < int

## Conversión explícita

char < int < float < double

En ocasiones el programador necesita **forzar** la conversión.

La conversión forzada de tipos static\_cast<tipo\_de\_dato>(expresion) convierte la representación interna de expresion a la representación marcada por tipo\_de\_dato.

**Peligro:** Las conversiones descontroladas conllevan graves riesgos. Un ejemplo clásico es el ocurrido con el [cohete Ariane,](https://youtu.be/gp_D8r-2hwk) que se produjo por una conversión desde un número en coma flotante de 64 bits (que era mayor de 32768) a un número entero de 16 bits.

### INSTRUCCIONES

* Muestre con ejemplos sencillos cada tema visto en esta clase.
* Realice un organizador gráfico que resuma la base teórica presentada en este laboratorio.
* Consulte como definir “Constantes” en C++

### DESARROLLO

### EJERCICIOS DESARROLADOS EN CLASE

### Ejercicio 1 de precedencia

### 

### Ejercicio 2 declaración de variables

### 

### Ejercicio 3 - Conversión implícita

### 

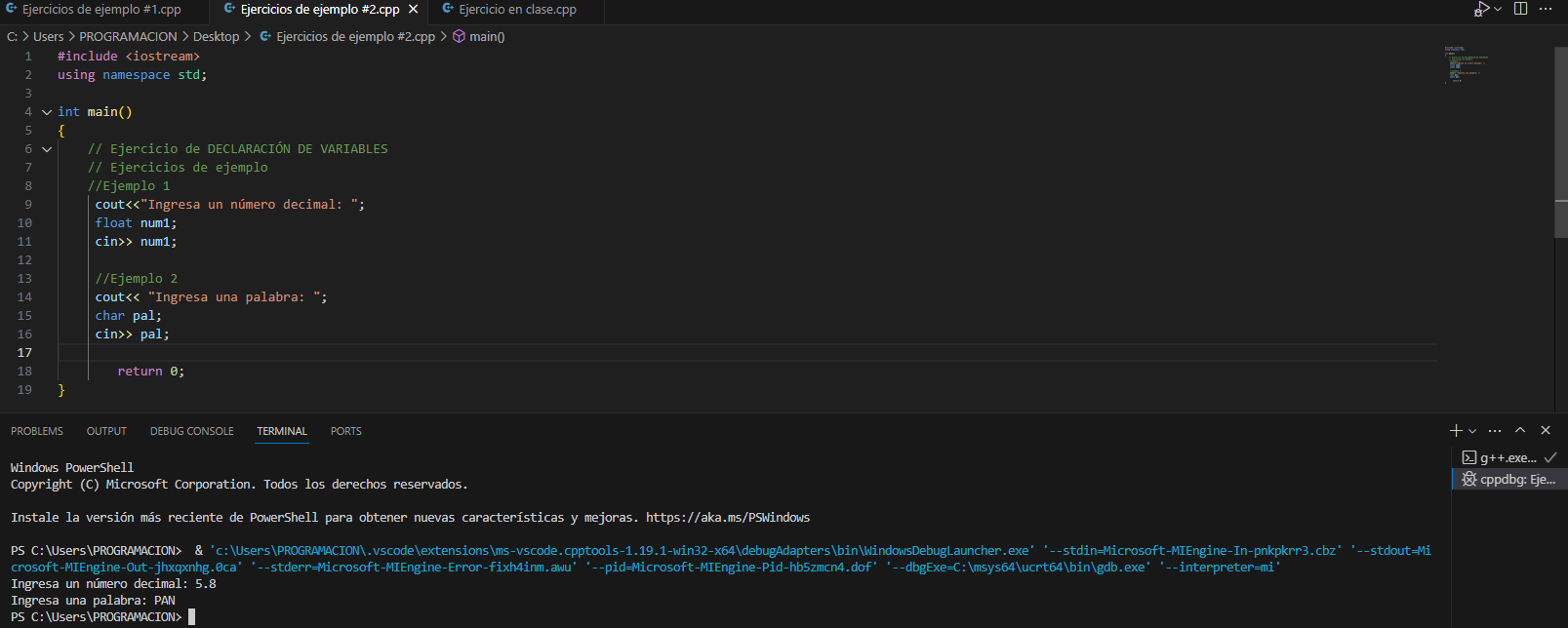
### Ejercicio 4 - Conversión explícita

### 

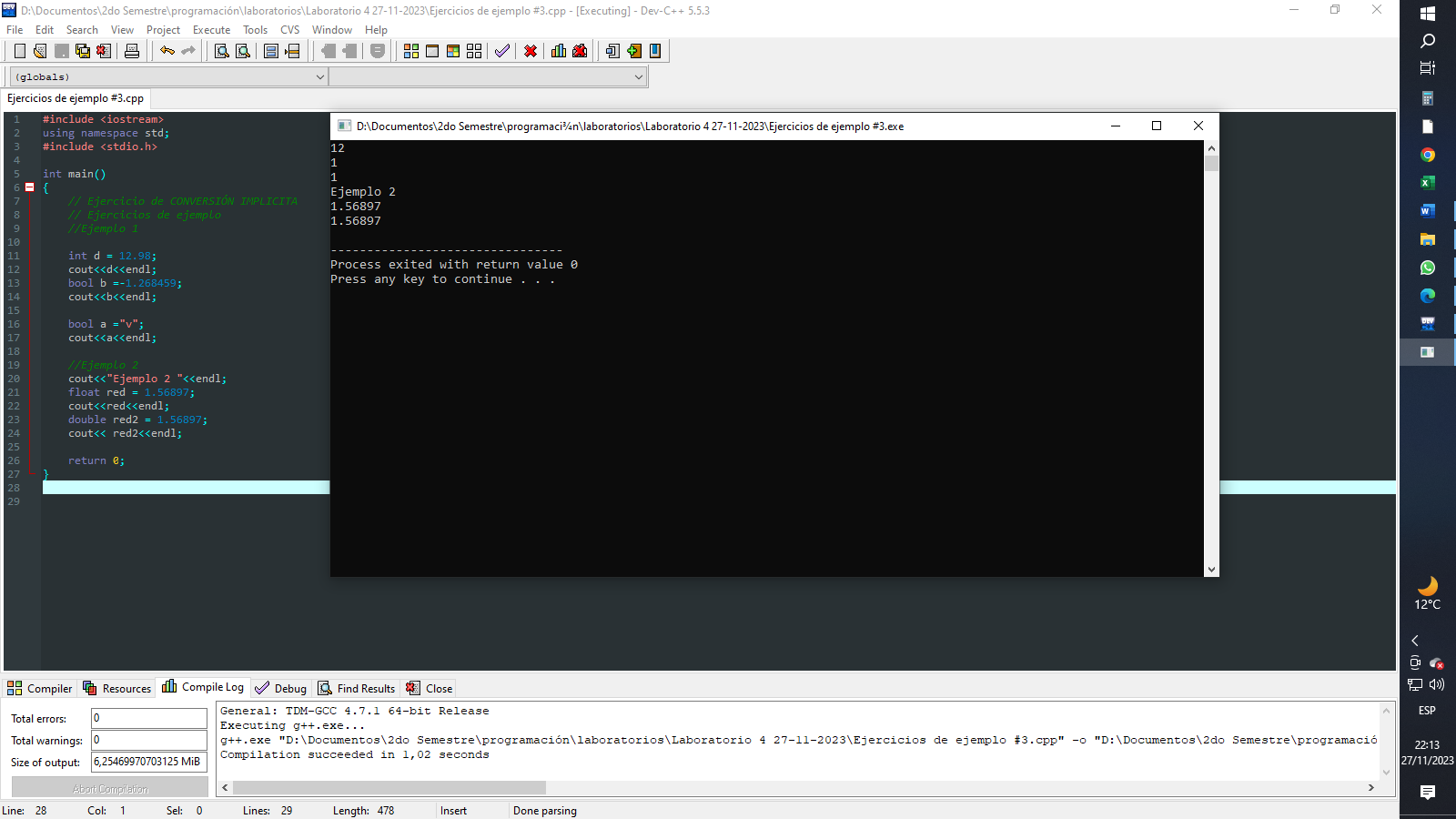
* **MUESTRE CON EJEMPLOS SENCILLOS CADA TEMA VISTO EN ESTA CLASE.**
* **EJEMPLO 1 - PROCEDENCIA**

### 

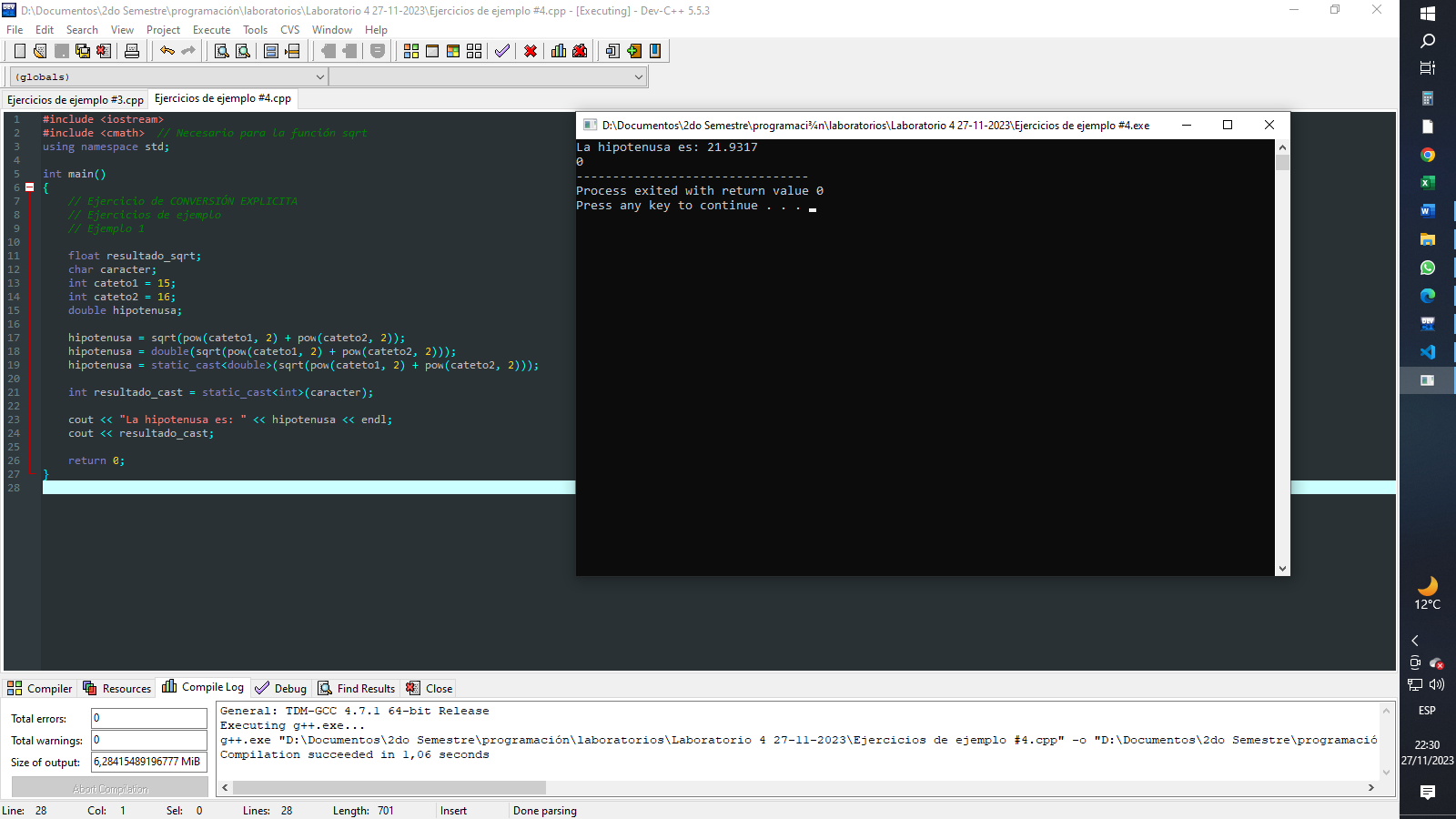
* **EJEMPLO 2 – DECLARACIÓN DE VARIABLES**

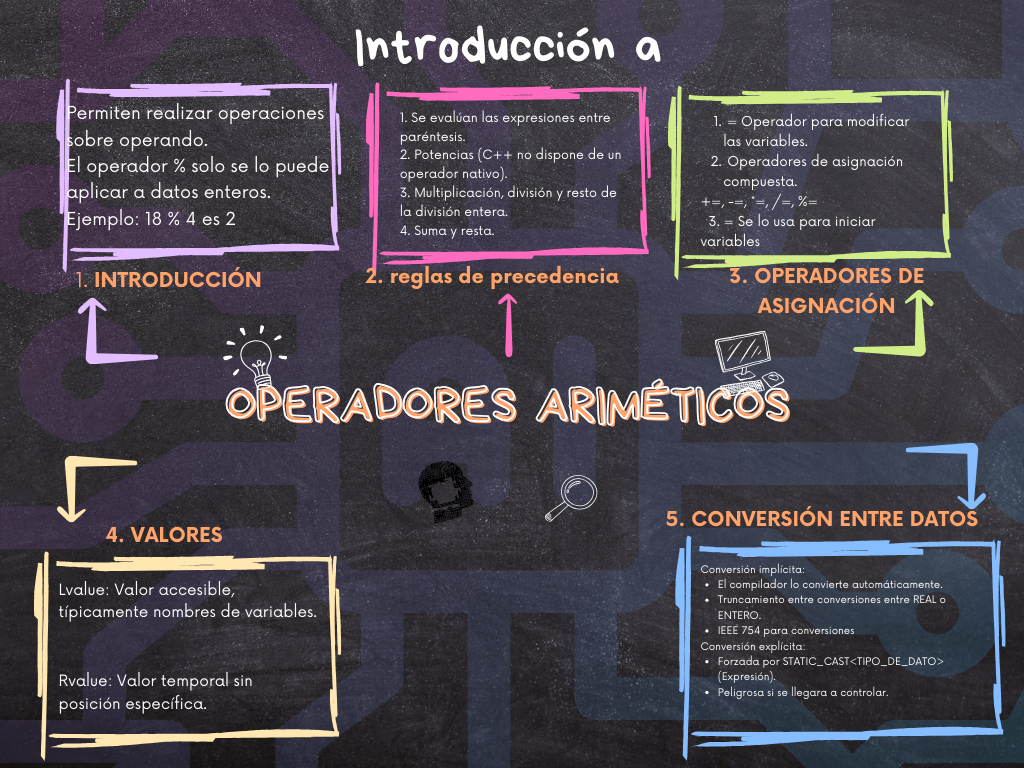
****

* **EJEMPLO 3 – CONVERSIÓN IMPLICITA**



* **EJEMPLO 4 – CONVERSIÓN EXPLICTA**



**REALICE UN ORGANIZADOR GRÁFICO QUE RESUMA LA BASE TEÓRICA PRESENTADA EN ESTE LABORATORIO.**

**CONSULTE COMO DEFINIR “CONSTANTES” EN C++**

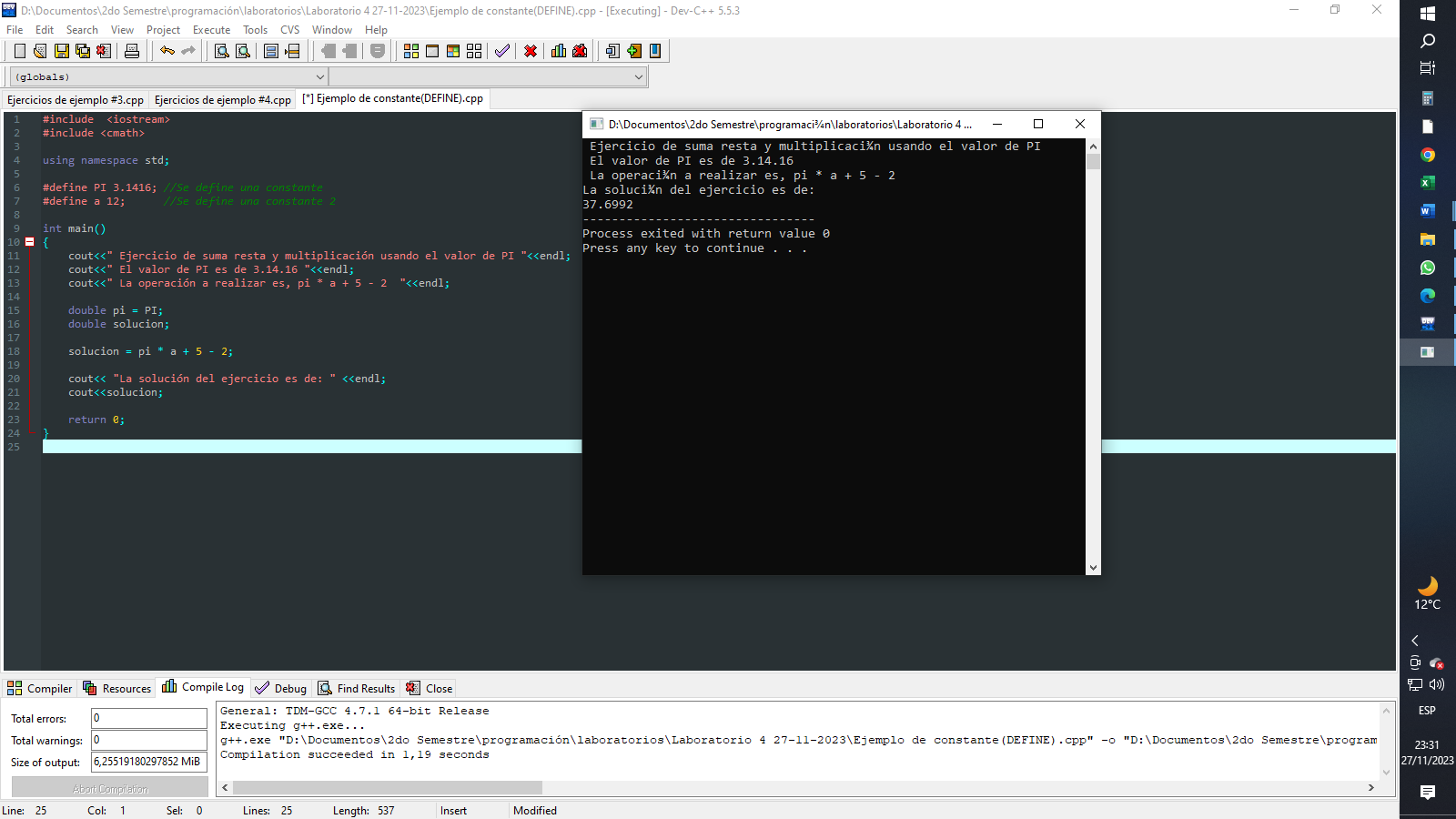
Constante: Tiene un valor fijo durante toda su ejecución, lo cual no cambia a lo largo del programa, estas son útiles para poder especificar el tamaño de un vector, dichas constantes se las declara después de haber declarado las librerias y antes de las funciones [1]

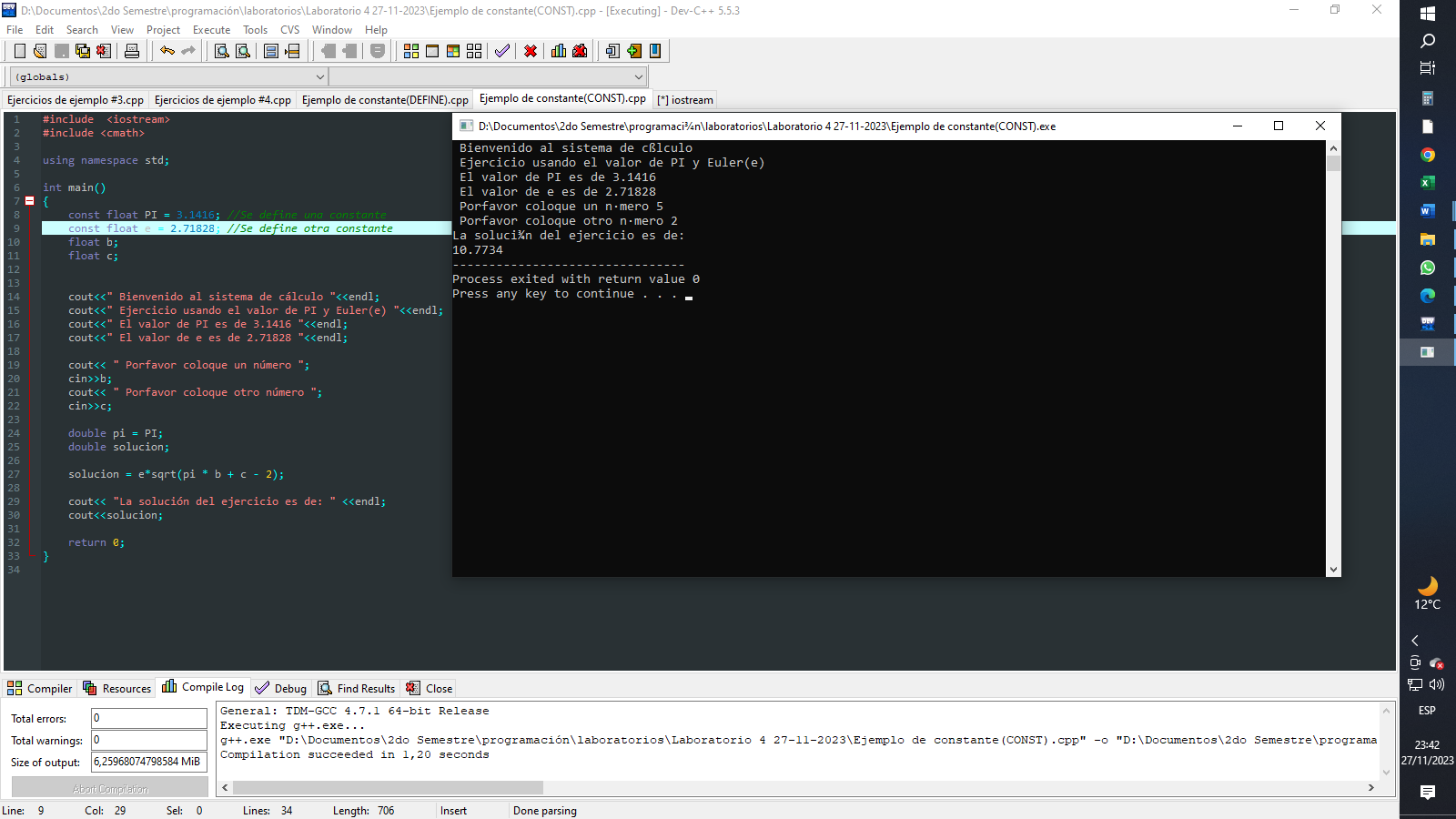
Dichas constantes se las puede definir de dos formas,

* #define nombre\_constante: permite declara constantes de manera rápida y sencilla, esto se debe hacer después del #include [2].
* const: Declara constantes de manera más adecuada y acorde, poseen dato asociado y son declaradas dentro del código como cualquier tipo. [2]

EJEMPLOS:

-Desarrollados en códigos :

**EJEMPLO 1 CON LA FUNCIÓN #define**

**EJEMPLO 2 CON LA FUNCIÓN #define**

### PRESENTACIÓN

### CONCLUCIONES

### Dicha información a cuál fue proporcionada nos puede ayudar a comprender conceptos fundamentales con respecto al lenguaje de programación de C++, enfatizando en la importancia de correcto uso e inicio de variables, proporcionando una estructura adecuada par adquirir conocimientos y habilidades que un programador como tal puede ir desarrollando.

### RECOMENDACIONES

### Se recomienda a las personas que recién van iniciando poder investigar más sobre el uso del lenguaje de C++, ya que de acuerdo a lo presentado anteriormente pueden existir opciones las cuales no se tiene conocimiento como tal, lo que se puede reforzar con la investigación, fomentando el crecimiento personal como programador.

# Referencias

|  |  |
| --- | --- |
| [1] | J. D. M. Gonzales, «Programar ya,» 2021. [En línea]. Available: https://www.programarya.com/Cursos/C++/Sistema-de-Tipos/Variables-y-Constantes. |
| [2] | J. D. m. Gonzales, «Programar ya,» 2021-2023. [En línea]. Available: https://www.programarya.com/Cursos/C++/Sistema-de-Tipos/Variables-y-Constantes. |