PROGRAMACIÓN II

CARRERA: Análisis De Sistemas

SEMESTRE: Tercero

INTRODUCCION A LA ASIGNATURA

El objetivo general del presente libro electrónico es permitir al estudiante a través de la lectura adquirir los conocimientos y habilidades necesarios para programar en C++ y Python, a través de una estructura organizada y con ejercicios prácticos que refuercen su aprendizaje.

Cada capítulo incluirá:

- Explicación teórica de los conceptos clave.
- Ejemplos de código y ejercicios prácticos.
- Secciones de "Conocer", "Hacer" y "Ser" para desarrollar las competencias.
- Estrategias de aprendizaje y evaluación sugeridas.
- Referencias bibliográficas y recursos adicionales.

Guido R. Portillo .Z

- T.S.U en Informatica y Docente Universitario -

Una publicación de:

CONTENIDOS

1. **Introducción a C++ y Python**

    - Orígenes y generalidades de los lenguajes de programación
    - Estructura básica de un programa en C++ y Python
    - Compilación y manejo de memoria
    - Declaración y tipos de variables
    - Sentencias de asignación: constantes, operadores y expresiones

2. **Funciones y Expresiones**

    - Funciones de control: constantes, operadores y expresiones
    - Función "Main"
    - Funciones como procedimiento
    - Recursividad en las funciones

3. **Control de Flujo**

    - Partes de una estructura de control.
    - Sentencias o instrucciones
    - Estructuras de selección if / else
    - Estructuras condicionales anidadas
    - Estructura While, For y Do While

4. **Arreglos y Punteros**

    - Arreglos bidimensionales y multidimensionales
    - Creación y administración de punteros

5. **Desarrollo de Programas con Funciones**

    - Concepto, elementos y uso de funciones
    - Importación y llamado de módulos
    - Invocación de funciones
    - Funciones como módulos

6. **Ejercicios Propuestos y Resueltos**

- Ejercicios de aplicación en C++
- Ejercicios de aplicación en Python

7. **Soluciones y explicaciones detalladas**

8. **Recursos Adicionales**

9. **Recomendaciones y recursos para profundizar en C++ y Python**

ÍNDICE

INTRODUCCION A LA ASIGNATURA 2

ÍNDICE 5

UNIDAD I 9

Orígenes de C++: **9**

Orígenes de Python: **9**

Creación de un proyecto en C++ **11**

Descarga del entorno para programar con C++. 11

Creación de un proyecto en Python **12**

Conceptos de Variables, Constantes y Tipos de Datos **12**

Variables 12

Constantes 12

Tipos de Datos 13

Forma General del Programa en C, C++ y Python: Biblioteca y Enlazado C y C++: 14

Analizaremos la estructura del programa: 15

Uso de namespaces en C++ 17

Funciones de Entrada/Salida **27**

Funciones de Entrada/Salida en C++ 27

Funciones de Entrada/Salida en Python 29

Palabras Reservadas en C++ **33**

Palabras Reservadas en Python 35

Comentarios: **37**

Comentarios en C++ 37

Comentarios en Python 39

Modificadores de formato **41**

Modificadores de Formato en C++ 41

Compilación y Mapa de Memoria **44**

Compilación: 44

Mapa de Memoria: **44**

Compilación y Enlazado en C y C++ ... **51**
Compilación: ... 51
Enlazado: ... 52
Compilación y Ejecución en Python ... 53
Interpretación: ... 53
Compilación a Bytecode: ... 53
Unidad II ... 60
Funciones y Expresiones en C++ y Python ... 60
Funciones de Control: Constantes, Operadores y Expresiones ... **60**
Constantes: ... 60
Operadores: ... 61
Expresiones: ... 61
Función "Main" ... 61
"C" como Bloque de Funciones: ... **62**
Funciones como Procedimiento ... 62
Recursividad en las Funciones ... **65**
UNIDAD III ... 75
CONTROL DE FLUJO EN C++ Y PYTHON ... 75
Partes de una Estructura de Control ... **75**
Estructuras de Selección: `if` / `else` ... **76**
Estructuras Condicionales Anidadas ... **79**
Estructura `While`, `For` y `Do While` ... **82**
Estructura `While` ... 83
Estructura `For` ... 85
Estructura `Do While` ... 88
Resumen sobre las estructuras condicionales en C++: ... **91**
Estructuras de Selección en C++ ... 91
Estructuras Condicionales Anidadas ... 93
Operadores de Comparación ... 93
Operadores Lógicos ... 94

Estructuras condicionales en Python: **94**
Estructuras de Selección en Python 94
Estructuras Condicionales Anidadas 96
Operadores de Comparación 96
Operadores Lógicos 97
EJERCICIOS PROPUESTOS **98**
Soluciones a los Ejercicios Propuestos **99**
UNIDAD IV 111
ARREGLOS Y PUNTEROS EN C++ Y PYTHON 111
Arreglos Bidimensionales **111**
Arreglos Multidimensionales **113**
Creación de Punteros **116**
Administración de Punteros 119
Ejercicios Propuestos **122**
Soluciones a los Ejercicios Propuestos **123**
Técnicas De Administración De Memoria **134**
Asignación Dinámica de Memoria 134
Arreglos Dinámicos: 134
Gestión de Recursos 135
Punteros a Punteros 135
Evitar Fugas de Memoria 136
UNIDAD V 142
DESARROLLO DE PROGRAMAS CON FUNCIONES EN C++ Y PYTHON 142
Concepto, Elementos y Uso de Funciones **142**
Pasos para Declarar y Definir Funciones en C++ 142
Declaración de la Función: 142
Definición de la Función: 143
Llamada a la Función: 144
Uso de la Función `main`: 144

Resumen de los Pasos 146
Ejemplo de Declaración y Definición....146
Desglose del Ejemplo 148
Para definir y utilizar funciones en Python, se siguen estos pasos: 148
1. Definición de la función: 148
2. Invocación de la función: 149
3. Funciones como módulos: 149
4. Funciones anónimas (lambda): 150
Las principales diferencias entre las funciones en Python y C++ son: 151
Declaración y Definición 151
Tipos de datos numéricos 152
Tipos de texto 154
Tipos booleanos 154
Tipos de secuencias 155
Tipos de mapeo 157
Tipos de conjuntos 158
Tipos de bytes 159
Tipos de datos principales en C++.... 161
Paso de Argumentos.... 164
Retorno de Valores.... 164
Funciones Anónimas.... 164
Importación y Llamado de Módulos 165
Invocación de Funciones....165
Funciones como Módulos....166
Avances de C++ y Python, Proyecciones Futuras y la Importancia de su Uso.... 183
Avances Recientes 183
Proyecciones Futuras 184
Importancia de su Uso 184

UNIDAD I

INTRODUCCIÓN AL LENGUAJE C++ Y PHYTON

En esta unidad se proporciona una introducción teórica y práctica a los conceptos fundamentales de C++ y Python, junto con ejemplos y ejercicios que permiten a los estudiantes aplicar lo aprendido.

Orígenes de C++:

C++ es un lenguaje de programación de propósito general desarrollado por Bjarne Stroustrup en los Laboratorios Bell a principios de la década de 1980. C++ se originó como una extensión del lenguaje de programación C, agregando características de programación orientada a objetos (POO).

Otras mejoras. C++ fue diseñado para ser un lenguaje de programación eficiente y de bajo nivel, manteniendo la compatibilidad con C, pero con la capacidad de crear programas más complejos y modulares.

Orígenes de Python:

Python es un lenguaje de programación de alto nivel, interpretado y multiparadigma, creado por Guido van Rossum en 1991. Python se diseñó con el objetivo de ser un lenguaje fácil de leer y escribir, con una sintaxis clara y concisa, lo que lo hace accesible para principiantes y programadores experimentados.

Python se ha convertido en un lenguaje popular y ampliamente utilizado en una variedad de aplicaciones, desde el desarrollo web y la ciencia de datos hasta la automatización de tareas y el aprendizaje automático.
Correlación entre C++ y Python:

Ambos lenguajes de programación son ampliamente utilizados y tienen una gran comunidad de desarrolladores.

C++ se considera un lenguaje de programación de bajo nivel, orientado a la eficiencia y el rendimiento, mientras que Python se considera un lenguaje de alto nivel, más enfocado en la legibilidad y la facilidad de uso.

C++ se utiliza principalmente en el desarrollo de software de sistemas, aplicaciones de alto rendimiento y juegos, mientras que Python se utiliza en una amplia gama de aplicaciones, desde el desarrollo web hasta la ciencia de datos y la inteligencia artificial.

Aunque tienen diferentes enfoques y características, C++ y Python pueden complementarse y utilizarse juntos en proyectos de desarrollo de software. Por ejemplo, se puede usar C++ para implementar componentes de bajo nivel y Python para crear interfaces de usuario y scripts de automatización.

Ambos lenguajes de programación siguen evolucionando y adaptándose a las necesidades cambiantes de la industria y la tecnología, manteniendo su relevancia y popularidad en el mundo de la programación.

Ejercicio Propuesto:

Investiga y presenta un breve informe sobre las aplicaciones actuales de C++ y Python en la industria.

Creación de un proyecto en C++

Descarga del entorno para programar con C++.

Existen muchos compiladores de C++, tanto para Windows, Mac, Linux etc, nosotros utilizaremos el Visual C++ de Microsoft que lo podemos descargar de aquí: https://visualstudio.microsoft.com/es/

El Visual Studio Community 2019 es gratuito y tiene entre otras herramientas el C++ (puede utilizar el Visual Studio 2017 si ya lo tiene instalado), también existen otros entornos de programación como **Dev-C++** que lo podemos descargar de aquí:

https://www.bloodshed.net/

Así como también podemos usar entornos de programación online como:

- https://paiza.io/es/languages/online-cpp-compiler
- https://www.programarya.com/Cursos/C++/Entornos

Creación de un proyecto en Python

Para la descarga del lenguaje Python lo hacemos del sitio: python.org (descargar la versión más actual 3.8)

Guía para instalar Python en su equipo.

- https://www.tutorialesprogramacionya.com/pythonya/detalleconcepto.php?punto=2&codigo=2&inicio=0

Si el editor que viene por defecto con Python no le convence por ser muy limitado y aplicado fundamentalmente en el aprendizaje de programación, puede consultar otros editores disponibles para Python.

Se recomendación es el editor VS Code.

También puedes usar entorno web que facilitaran tu aprendizaje como pyCharrm el cual puedes conocer y descargar en el siguiente enlace:

- https://www.jetbrains.com/es-es/pycharm/download/

Conceptos de Variables, Constantes y Tipos de Datos

Variables

Las variables son espacios de almacenamiento en la memoria que tienen un nombre y un tipo de dato asociado. Su valor puede cambiar durante la ejecución del programa.

Constantes

Las constantes son valores que no cambian durante la ejecución del programa. Se definen una vez y su valor permanece fijo.

Tipos de Datos

Los tipos de datos especifican el tipo de valor que una variable o constante puede almacenar. Los tipos de datos comunes incluyen:

Enteros: Números enteros (por ejemplo, `int` en C++ y `int` en Python).

Punto Flotante: Números con decimales (por ejemplo, `float` y `double` en C++, `float` y `float` en Python).

Caracteres: Un solo carácter (por ejemplo, `char` en C++).

Cadenas: Secuencias de caracteres (por ejemplo, `string` en C++ y `str` en Python).

Booleanos: Valores de verdad (`true` o `false`).

Tabla 1

Comparativa De Variables entre C++ y Python

Característica	C++	Python
Declaración de Variables	`int x;`	`x = 10`
Constantes	`const int x = 10;`	`x = 10` (usualmente en mayúsculas para indicar que es constante)
Tipos de Datos	`int`, `float`, `char`, `double`, `bool`, `string`	`int`, `float`, `str`, `bool`
Asignación de Valores	`x = 5;`	`x = 5`
Tipo Dinámico	No (tipo estático)	Sí (tipo dinámico)
Cadenas	`std::string`	`str`
Booleanos	`bool` (true/false)	`bool` (True/False)

Fuente: *https://www.perplexity.ai*

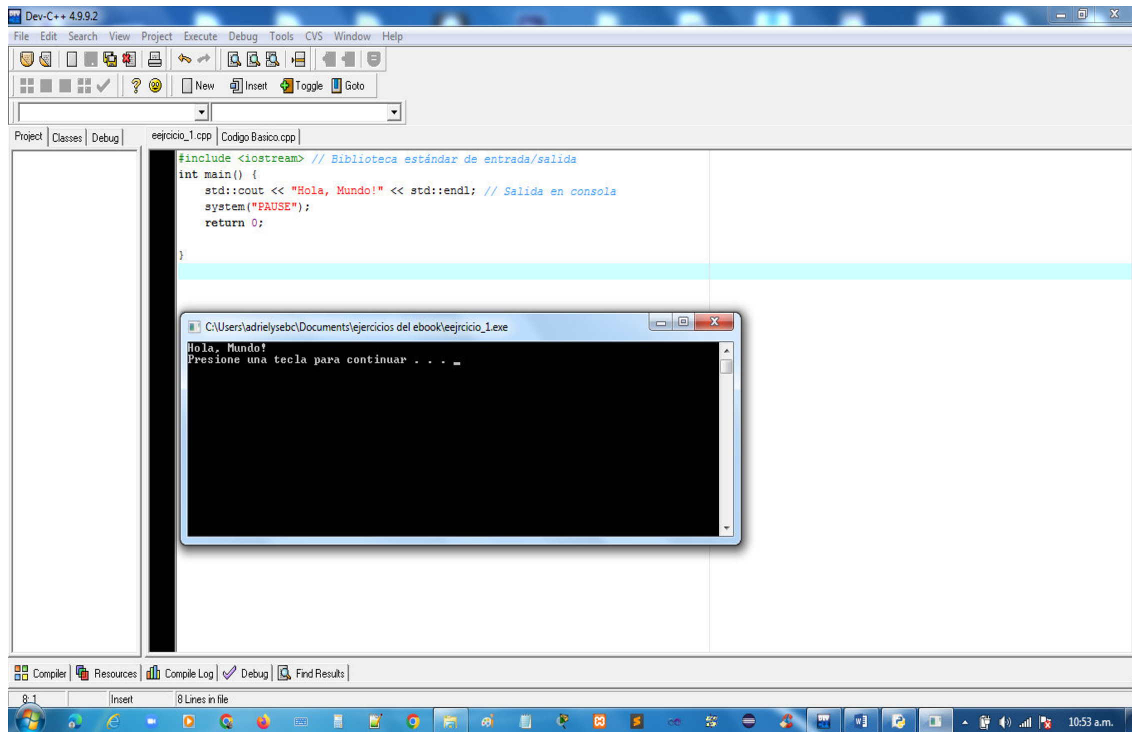
Las variables y constantes son fundamentales en la programación, y los tipos de datos determinan cómo se almacenan y manipulan los valores. La tabla comparativa muestra las diferencias clave entre C++ y Python en cuanto a la declaración y uso de variables y constantes.

Forma General del Programa en C, C++ y Python: Biblioteca y Enlazado

C y C++:

La estructura básica de un programa incluye la inclusión de bibliotecas y la función `main()`.

```cpp
#include <iostream> // Biblioteca estándar de entrada/salida
int main() {
    std::cout << "Hola, Mundo!" << std::endl; // Salida en consola
    return 0;
}
```



Analizaremos la estructura del programa:

#include <iostream.h>

- La parte del **#include** se refiere a la biblioteca de funciones que vamos a utilizar. Es decirpara llamar a una biblioteca en particular debemos hacer lo siguiente:

#include <librería_solicitada>

- El estándar de C++ incluye varias bibliotecas de funciones, y dependiendo del compiladorque se esté usando, puede aumentar el número.

int main(void){

- Todo programa en C++ comienza con una función **main()**, y sólo puede haber una.
- En C++ el **main()** siempre regresa un entero, es por eso se antepone "**int**" a la palabra "**main**". Los paréntesis que le siguen contienen lo que se le va a mandar a la función.
- En este caso se puso la palabra **"void"** que significa vacío, es decir que a la función main no se le está mandando nada, podría omitirse el void dentro de los paréntesis, el compilador asume que no se enviará nada.
- La llave que se abre significa que se iniciará un bloque de instrucciones.

cout<<"Hola, Mundo"<<endl;

- Esta es una instrucción. La instrucción **cout** está definida dentro de la biblioteca **iostream.h,** que previamente declaramos que íbamos a utilizar.
- Una función, en este caso main() siempre comienza su ejecución con una instrucción (la que se encuentra en la parte superior), y continúa así hasta que se llegue a la última instrucción (de la parte inferior).
- Para terminar una instrucción siempre se coloca "**;**".
- Pero además de instrucciones se pueden invocar funciones definidas por el usuario (por supuesto diferentes de main) como se verá más adelante.
- **system("PAUSE")** se utiliza para que podamos ver la ejecución ya que la computadora lo hace muy rápido, permite pausar la consola antes que se cierre. Esto es útil en entornos de Windows, pero ten en cuenta que no es portable. En sistemas Unix, podrías usar `std::cin.get()` para lograr un efecto similar.
- **return 0**, es un aviso al SO que el programa acabo con éxito, si termina con otro número le indica que termino de forma incorrecta.

Uso de namespaces en C++

Un namespace, **no es más que una forma de crear un bloque**, y que todas las funciones que estén dentro del mismo, estén asociadas a ese namespace (o espacio de nombres), al cual se le asigna un nombre para identificarlo.

Por ejemplo:

```
namespace hola
   void f()
   {
      std::cout << “hola”;
   }
```

En este ejemplo, la función f está asociada al namespace hola.

¿Por qué surgieron los namespaces?

Todo se remonta al lenguaje C, en el que no existe el concepto de namespace.

En este lenguaje, a la hora de importar una librería, si esa librería tenía, por ejemplo, una función print, si nuestro programa contenía ya esa función, teníamos un problema, ya que **ni en C ni en C++ se pueden declarar y definir funciones con el mismo nombre**.

La solución para que eso no ocurriera fue comenzar a utilizar acrónimos, para que las funciones empezaran o acabaran con ese nombre.

Por ejemplo, OpenGL, todas sus funciones comienzan por gl, de esta forma cuando se llama a una función gl_print, sabemos que es de OpenGL y no tendremos problemas, ya que es extraño que otra persona tenga en su código una función llamada gl_print.

Para evitar esto, se añadieron los namespace a C++.

Ejemplos de namespaces

Por ejemplo, creamos una función f, la compilamos y la llamamos, nos mostrará la palabra "adios":

```
void f()

{

   std::cout << "adios";

}
```

Si posteriormente añadimos otra función f, que escriba "hola", la ejecutamos, se produce un error de compilación, ya que la función ya existe.

```
void f()

{

  std::cout << "adios";

}


void f()

{
```

```
  std::cout << "hola";
}
```

Al añadir el namespace se evita ese problema. Para utilizar el namespace hola, solo tenemos que indicar el nombre del mismo, seguido dos puntos dobles, que es el ámbito al que pertenece la función, y llamar a la función que queremos.

Con esto no solo **podemos encapsular funciones**, sino también **atributos**, **objetos**, **clases** y muchas cosas más de C++.

```
namespace hola
{
   void f() { std::cout << "hola";
}
void f()
{
   std::cout << "adios";
}
int main()
{
   f();
hola::f();
getchar();
return 0;
```

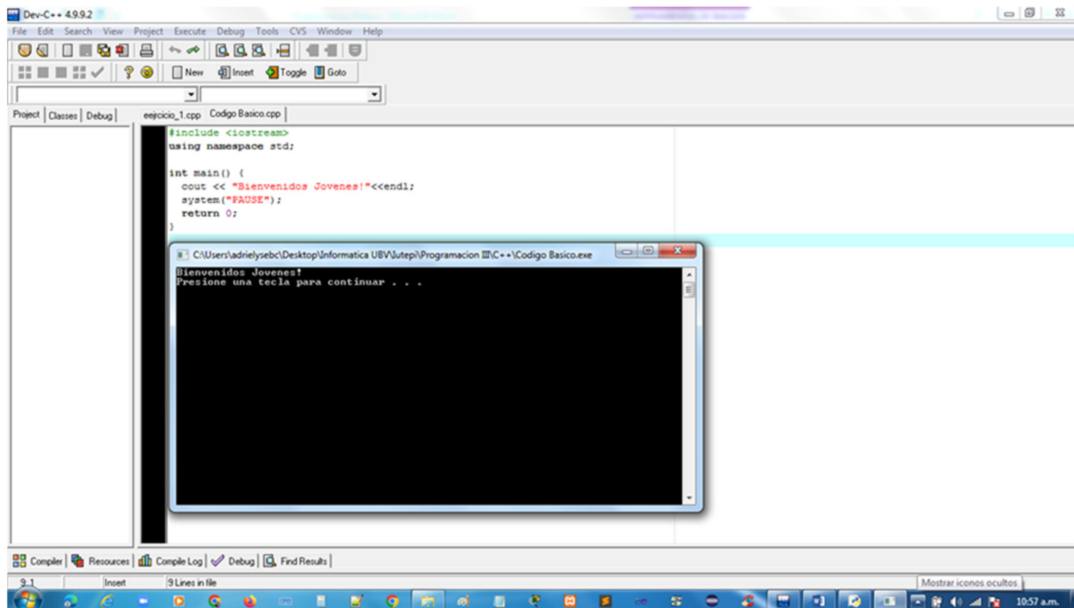
```
}
```

Al ejecutar ese código se nos muestra “adioshola”.

Hay una forma para evitar usa estos encapsulamientos, que sería así:

```
using namespace std;

int main()

{

   f();

hola::f();

cout << “loquesea”;

getchar();

return 0;

}
```

De esta forma, al indicar el namespace std, que es el namespace dónde está incluida toda la librería estándar de C++, ya no tenemos que indicar std::cout, sino que directamente indicamos cout.



Al utilizar este sistema hay que tener **cuidado**, porque se puede presentar algún **problema de ambigüedad**. Por ejemplo:

```
namespace hola
{
   void f() { std::cout << “hola”;
}
void f()
{
   std::cout << “adios”;
}
using namespace std;
using namespace hola;
```

```
int main()
{
    f();
hola::f();
cout << "loquesea";
getchar();
return 0;
}
```

Se produce una ambigüedad al emplear el using namespace, ya que tanto std como hola tienen incluida la función f. **Lo mismo ocurriría en caso de otra función que estuviese incluida en ambos namespaces**.

Mejora las habilidades de tus desarrolladores

Para evitar este problema de ambigüedad habría que separar esos namespaces, por ejemplo así:

```
namespace hola<
{
void f() { std::cout << "hola";
}

namespace otro
{
void f()
```

```
{
    std::cout << “adios”;
}
}
…
using namespace std;
using namespace hola;
int main()
{
f();
otro::f();
cout << “loquesea”;
getchar();
return 0;
}
```

Es importante saber separar los using namespace, y no se recomienda utilizar el using namespace std a menos que se esté muy seguro de que no va haber ninguna función que se utilice.

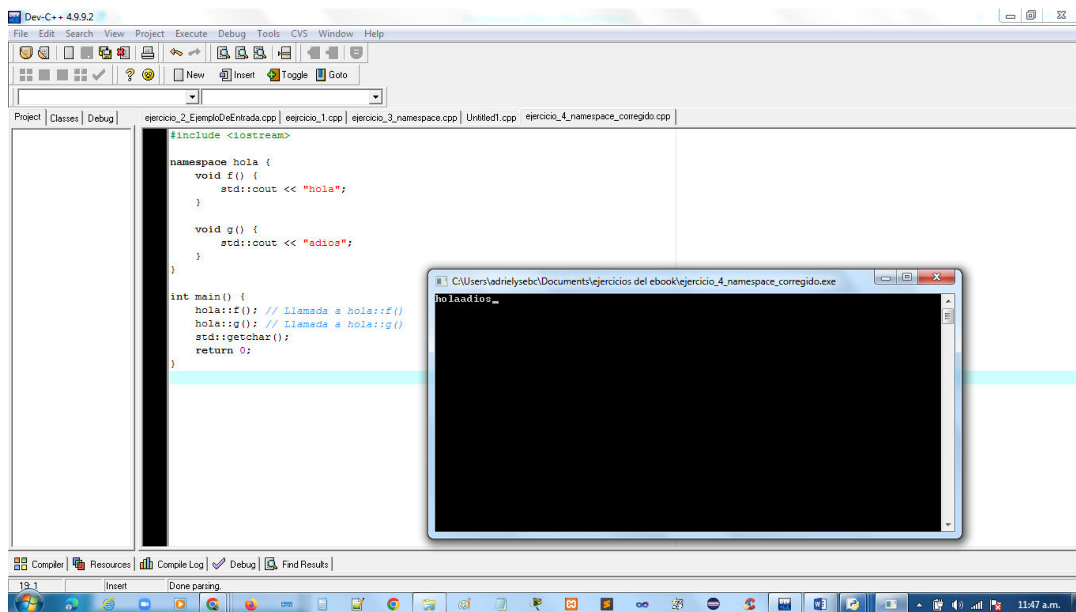
Además **no es recomendable utilizar using namespace dentro de los archivos de cabecera *.hpp**, que son incluidos en otros ficheros, y que puede provocar inclusiones recursivas, con lo que puede fallar y no que si lo provoca nuestro código o a una librería externa.

os código que proporcionado tienen algunos errores de sintaxis y Lproblemas de ámbito.

Aquí está la versión corregida:

```cpp
#include <iostream>


namespace hola {
    void f() {
        std::cout << "hola";
    }


    void g() {
        std::cout << "adios";
    }
}


int main() {
    hola::f(); // Llamada a hola::f()
    hola::g(); // Llamada a hola::g()
    std::getchar();
```

```
    return 0;
}
```



Explicación de los cambios:

1. Se Agregó `#include <iostream>` para poder usar `std::cout` y `std::getchar`.

2. Se corrigió el cierre de llave `}` al final del namespace `hola`. Antes estaba fuera de lugar.

3. Se cambió el nombre de la segunda función de `f()` a `g()` para evitar la redefinición de `f()` dentro del namespace.

4. Se agregó `hola::` antes de `g()` en `main()` para llamar a la función `g()` dentro del namespace `hola`.

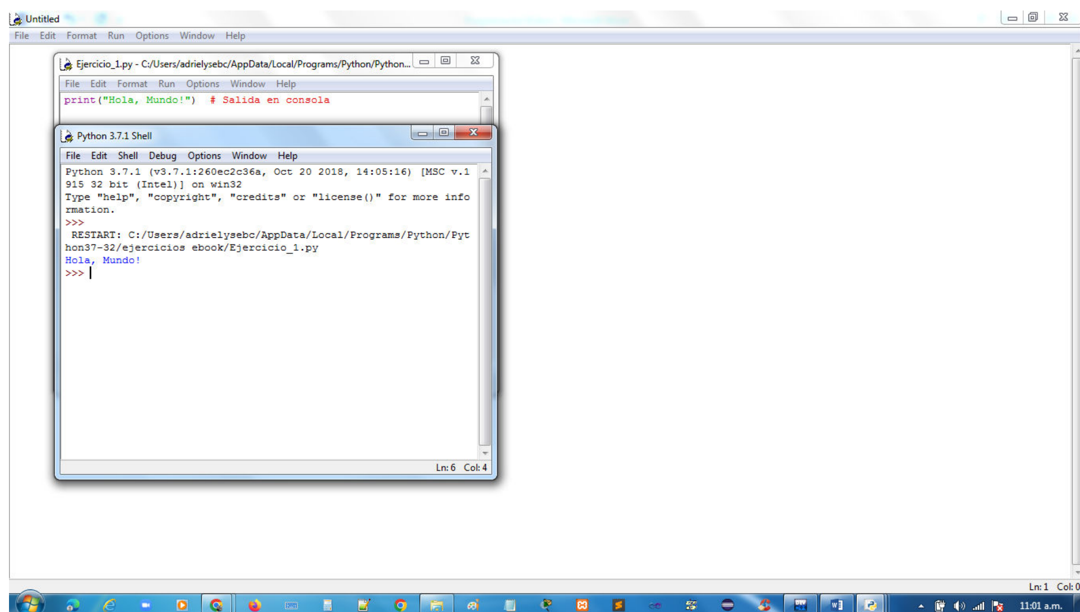
5. Se cambió `getchar()` a `std::getchar()` para usar la función de la biblioteca estándar.

6. Se elimino el cierre de llave `}` extra al final del código.

Con estos cambios, el código debería compilar y ejecutarse correctamente. Cuando se llama a `hola::f()` en `main()`, se imprime "hola" en la consola. Luego, cuando se llama a `hola::g()`, se imprime "adios". Finalmente, el programa espera a que el usuario presione Enter antes de terminar.

Python: La estructura es más simple, no requiere declaración de tipo.

```python
print("Hola, Mundo!")  # Salida en consola
```



Funciones de Entrada/Salida

Las funciones de entrada/salida (E/S) son fundamentales en la programación, ya que permiten interactuar con el usuario y manejar datos. A continuación, se presenta un resumen de las funciones de E/S en C++ y Python.

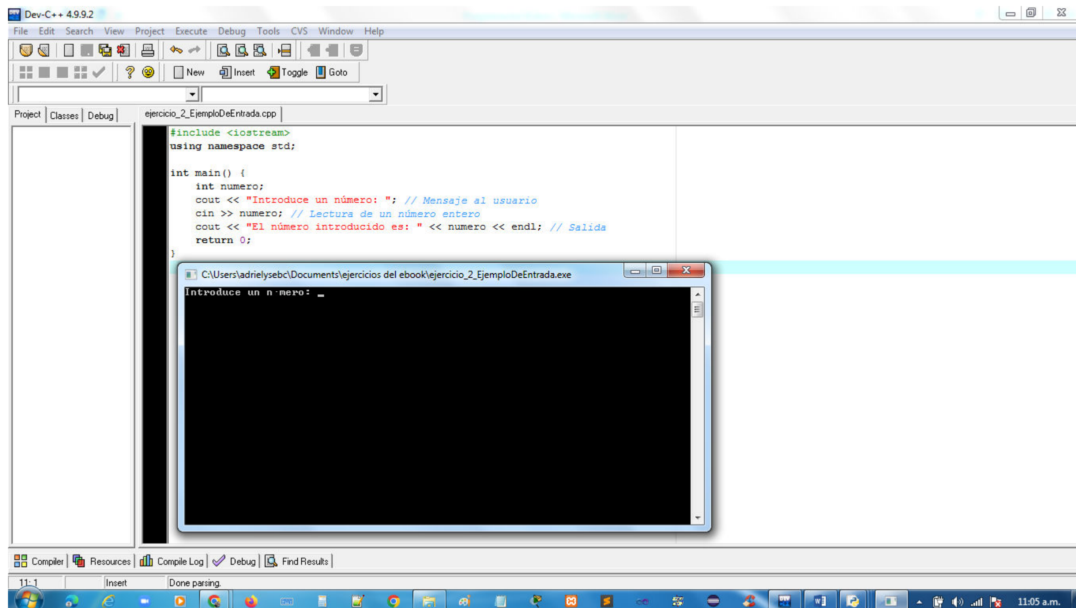
Funciones de Entrada/Salida en C++

Entrada

En C++, la entrada se realiza comúnmente utilizando la biblioteca estándar **`<iostream>`**, que proporciona el objeto **`cin`** para leer datos desde la entrada estándar (teclado).

Ejemplo de Entrada:

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


int main() {
    int numero;
    cout << "Introduce un número: "; // Mensaje al usuario
    cin >> numero; // Lectura de un número entero
    cout << "El número introducido es: " << numero << endl; // Salida
    return 0;
}
```

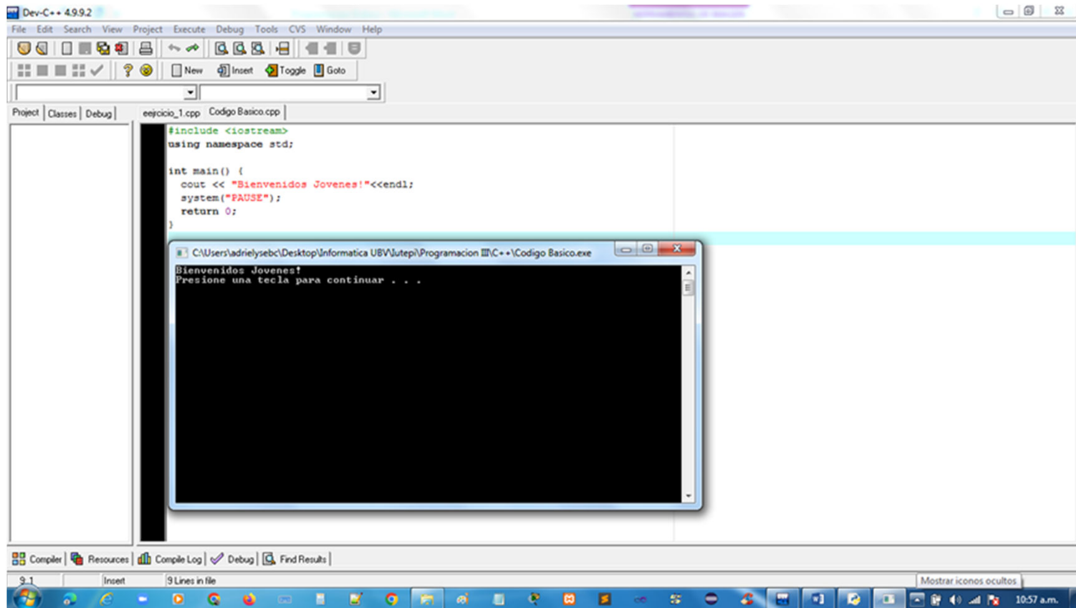


Salida

La salida se realiza utilizando el objeto `**cout**`, que permite mostrar datos en la salida estándar (pantalla).

Ejemplo de Salida:

```cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Hola, Mundo!" << endl; // Mensaje en pantalla
system("PAUSE"); // se utiliza para detener la ejecución del programa.
    return 0;
}
```



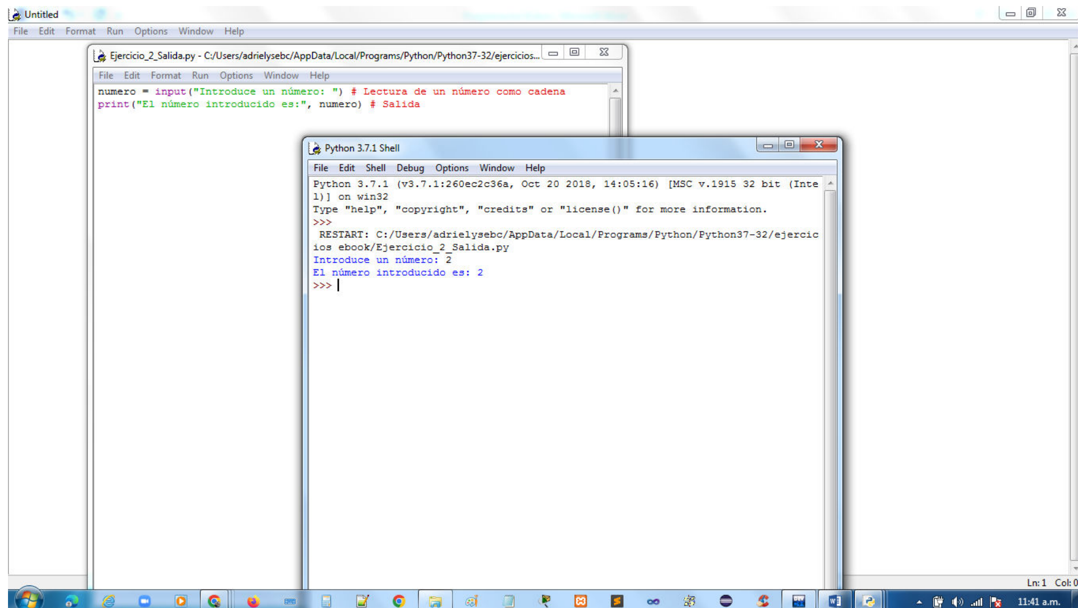
Funciones de Entrada/Salida en Python

Entrada

En Python, la entrada se realiza utilizando la función `**input()**`, que permite leer datos desde la entrada estándar (teclado).

Ejemplo de Entrada:

```python
numero = input("Introduce un número: ") # Lectura de un número como cadena
print("El número introducido es:", numero) # Salida
```



Salida

La salida se realiza utilizando la función `print()`, que muestra datos en la salida estándar.

Ejemplo de Salida:

```python
print("Hola, Mundo!") # Mensaje en pantalla
```

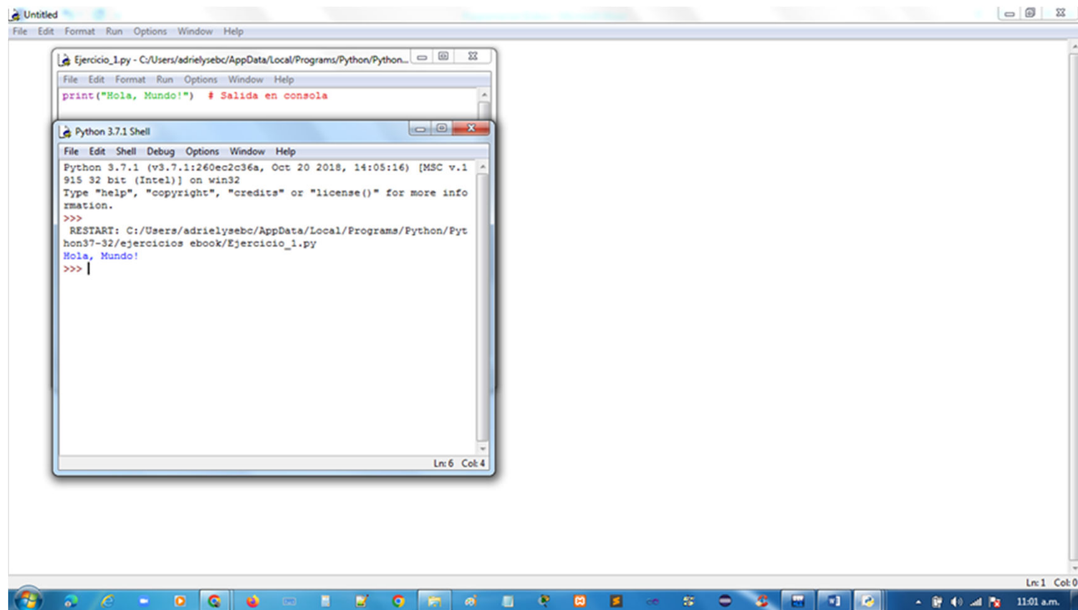


Tabla 2

Comparación de funciones de E/S

Característica	C++	Python
Entrada	`cin` para entrada estándar	`input()` para entrada estándar
Salida	`cout` para salida estándar	`print()` para salida estándar
Formato	Se utiliza manipuladores como `std::endl` para saltos de línea.	`print()` maneja automáticamente los saltos de línea.

Fuente: https://www.perplexity.ai

Los operadores << y >> son operadores de inserción y extracción de flujo respectivamente, y no deben confundirse con los de desplazamiento de bits. Estos operadores son muy eficaces porque no es necesario especificar

formatos para presentación o lectura, ellos los presentan en función al tipo de datos de la variable.

Aunque en ocasiones podría necesitar de nuestra ayuda para obtener los resultados específicos que queremos, y para eso están los modificadores de formato.

Ejercicio Propuesto:

- Escribe un programa en C++ y Python que imprima tu nombre y edad.

Palabras Reservadas en C++

Las palabras reservadas son términos que tienen un significado especial en un lenguaje de programación y no pueden ser utilizados como identificadores (nombres de variables, funciones, etc.).

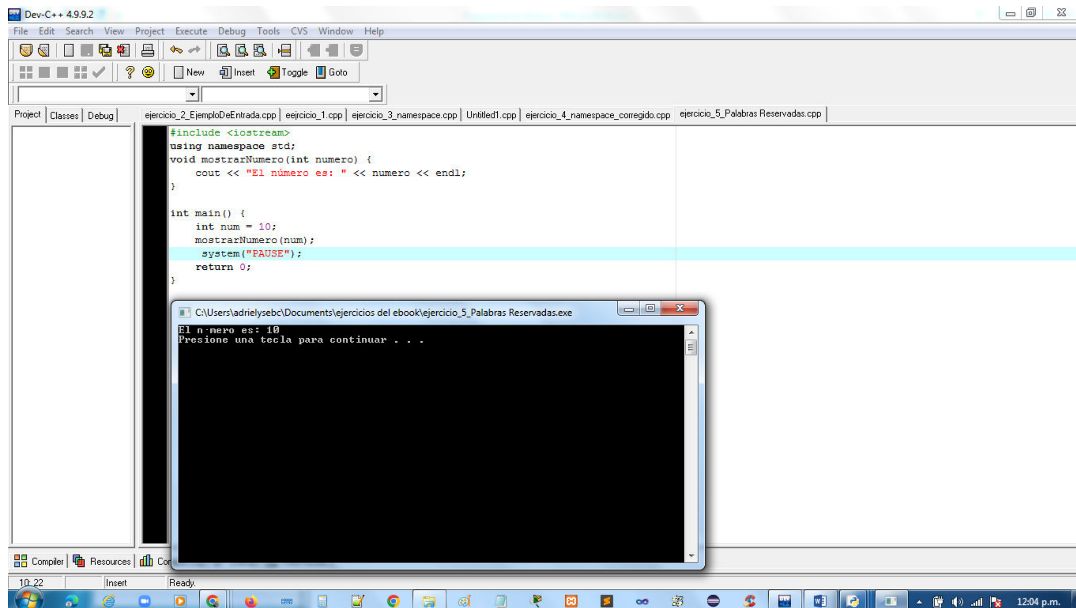
A continuación, se presentan las palabras reservadas en C++ y Python, junto con su uso general.

C++ tiene un conjunto de palabras reservadas que son fundamentales para la sintaxis del lenguaje. Algunas de las más comunes son:

- `**int**`: Define un tipo de dato entero.
- `**float**`: Define un tipo de dato de punto flotante.
- `**double**`: Define un tipo de dato de doble precisión.
- `**char**`: Define un tipo de dato de carácter.
- `**void**`: Indica que una función no retorna un valor.
- `**if**`: Inicia una estructura condicional.
- `**else**`: Proporciona una alternativa en una estructura condicional.
- `**for**`: Inicia un bucle for.
- `**while**`: Inicia un bucle while.
- `**return**`: Devuelve un valor de una función.
- `**class**`: Define una clase en programación orientada a objetos.
- `**public**`, `private`, `protected`: Modificadores de acceso en clases.

Ejemplo de Uso en C++

```cpp
#include <iostream>
using namespace std;
void mostrarNumero(int numero) {
    cout << "El número es: " << numero << endl;
}

int main() {
    int num = 10;
    mostrarNumero(num);
    return 0;
}
```



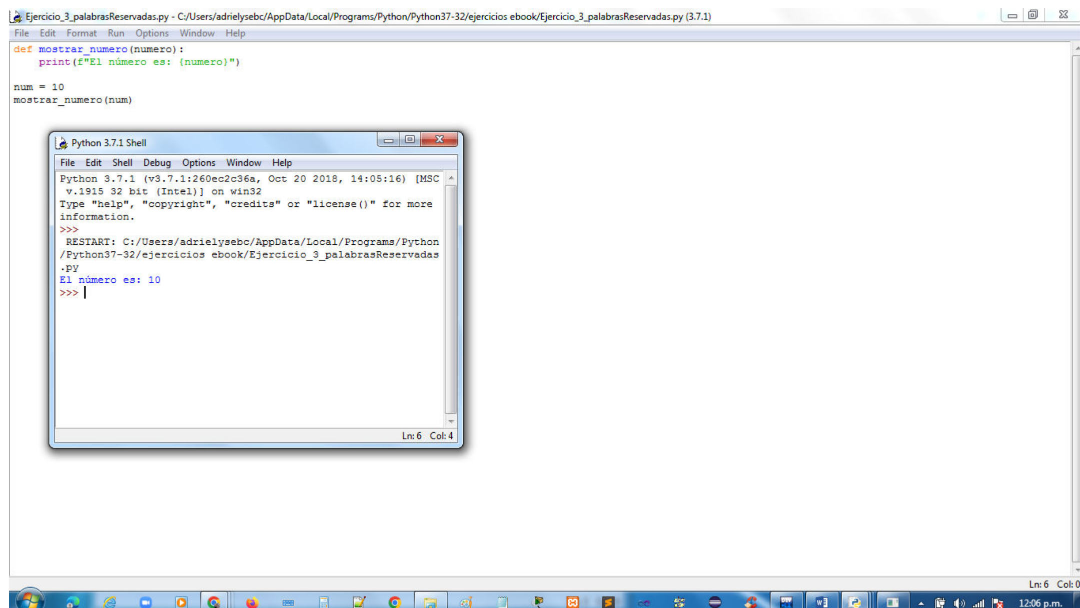
Palabras Reservadas en Python

Python también tiene un conjunto de palabras reservadas que son esenciales para su sintaxis. Algunas de las más comunes son:

- `**def**`: Define una función.
- `**return**`: Devuelve un valor de una función.
- `**if**`: Inicia una estructura condicional.
- `**elif**`: Proporciona una alternativa en una estructura condicional.
- `**else**`: Proporciona una alternativa final en una estructura condicional.
- `**for**`: Inicia un bucle for.
- `**while**`: Inicia un bucle while.
- `**class**`: Define una clase en programación orientada a objetos.
- `**import**`: Importa módulos.
- `**try**`, `**except**`: Manejo de excepciones.

Ejemplo de Uso en Python

```python
def mostrar_numero(numero):
    print(f"El número es: {numero}")


num = 10
mostrar_numero(num)
```



Las palabras reservadas son fundamentales para la estructura y el funcionamiento de los lenguajes de programación. En C++ y Python, estas palabras tienen significados específicos que definen la lógica y el flujo de los programas. Es esencial conocerlas para escribir código correcto y efectivo en ambos lenguajes.

Comentarios:

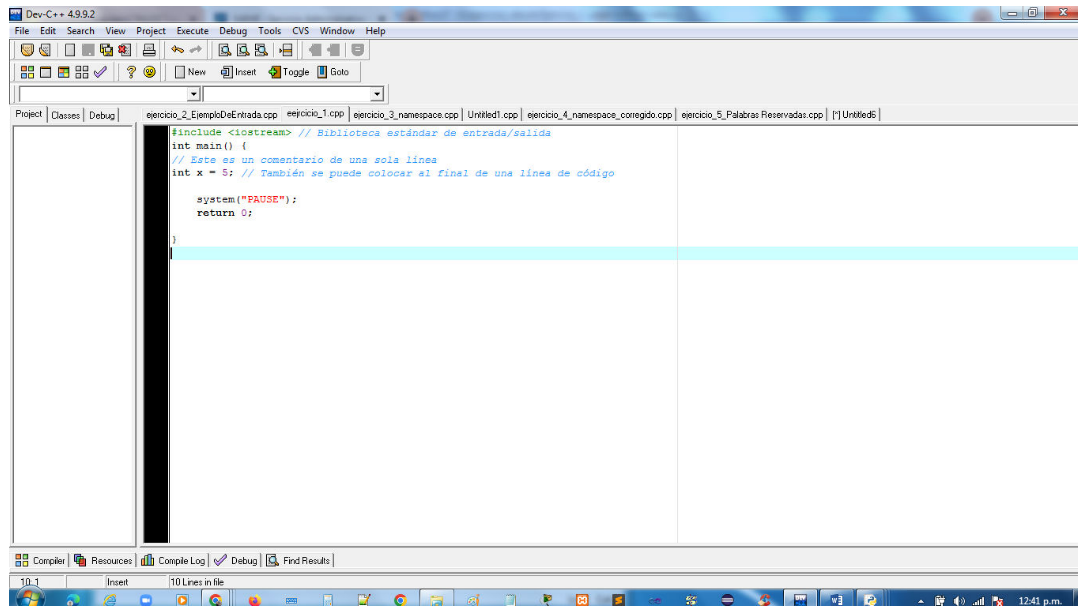
Los comentarios son una herramienta valiosa para mejorar la legibilidad y mantenibilidad del código en C++ y Python. Usarlos de manera efectiva y consistente es una buena práctica de programación.

Comentarios en C++

En C++, los comentarios se utilizan para agregar notas o explicaciones al código. Hay dos formas de escribir comentarios:

1. Comentarios de una sola línea: Se inician con `//` y se extienden hasta el final de la línea.

```cpp
// Este es un comentario de una sola línea
int x = 5; // También se puede colocar al final de una línea de código
```



2. Comentarios de múltiples líneas: Se encierran entre `/*` y `*/`. Pueden abarcar varias líneas.

```cpp

*/**

Este es un comentario

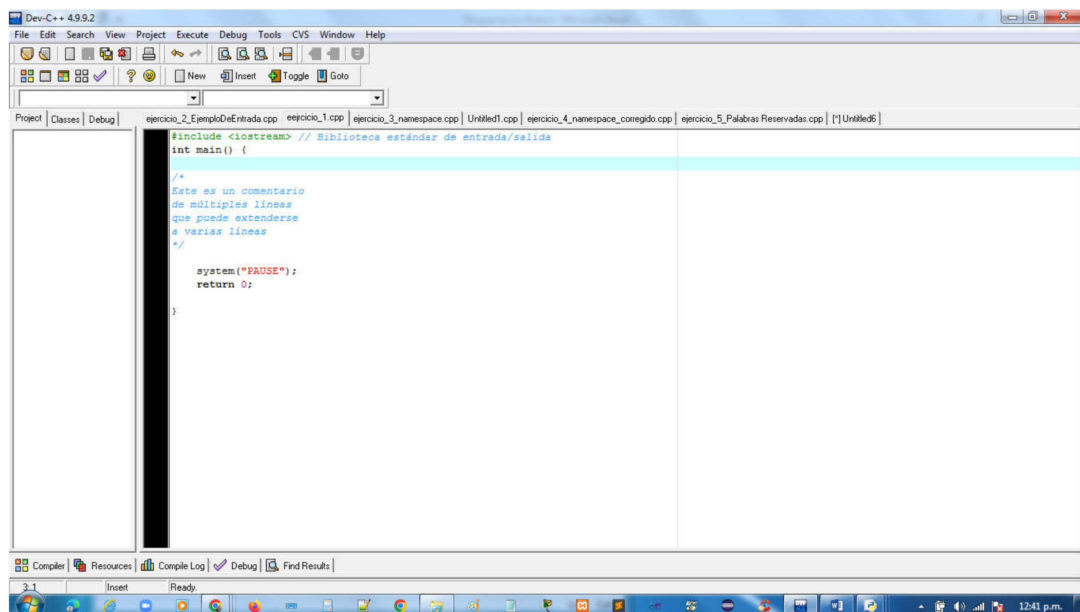
de múltiples líneas

que puede extenderse

a varias líneas

**/*

`



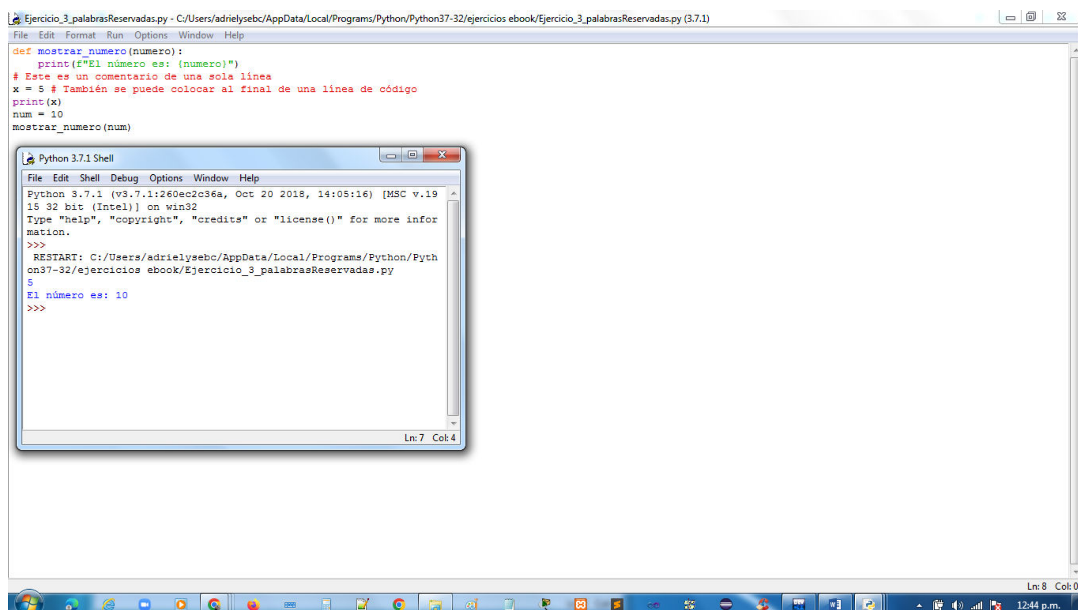
``

Comentarios en Python

Python también utiliza comentarios para agregar notas o explicaciones al código. Al igual que en C++, hay dos formas de escribir comentarios:

1. Comentarios de una sola línea: Se inician con `#` y se extienden hasta el final de la línea.

```python
# Este es un comentario de una sola línea
x = 5 # También se puede colocar al final de una línea de código
```



2. Comentarios de múltiples líneas: Se pueden usar cadenas de texto entre comillas simples, dobles o triples para crear comentarios de múltiples líneas.

```python
'''
```

Este es un comentario

de múltiples líneas

que puede extenderse

a varias líneas

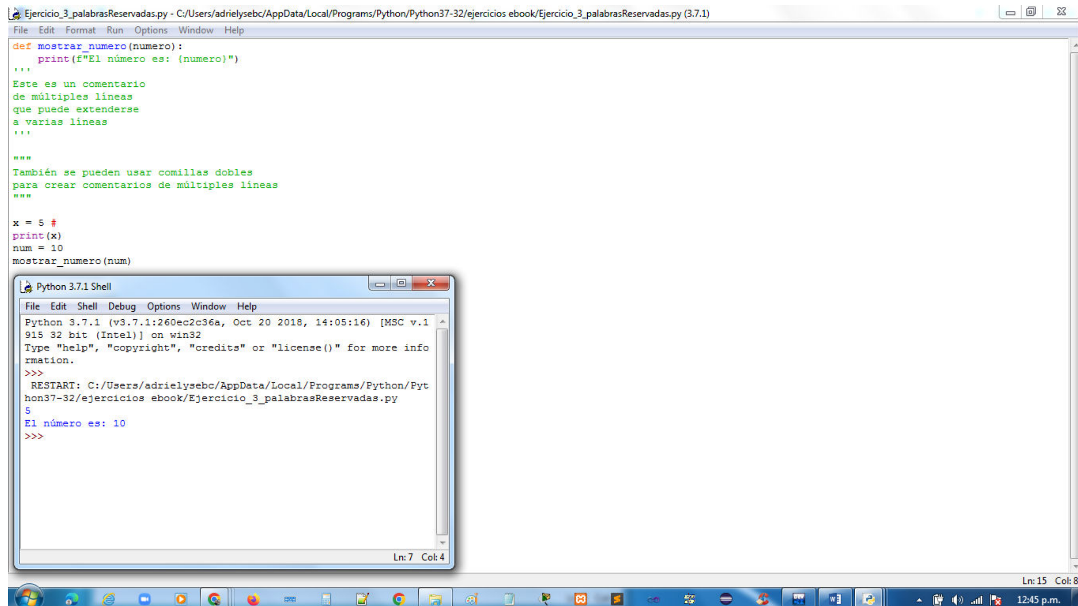
'''

"""

También se pueden usar comillas dobles

para crear comentarios de múltiples líneas

"""

```


```

Buenas Prácticas para Comentarios

- Usa comentarios para explicar el propósito y la funcionalidad del código.
- Mantén los comentarios actualizados y precisos.
- Evita comentar código que ya es lo suficientemente claro por sí mismo.
- Usa un estilo consistente para los comentarios en todo tu código.
- Comenta las funciones, clases y módulos para explicar su propósito y uso.
- Utiliza comentarios para explicar algoritmos complejos o decisiones de diseño.

Modificadores de formato

Los modificadores de formato en C++ y Python se utilizan para dar formato a la salida de datos. Aquí te presento un resumen de los principales modificadores en cada lenguaje:

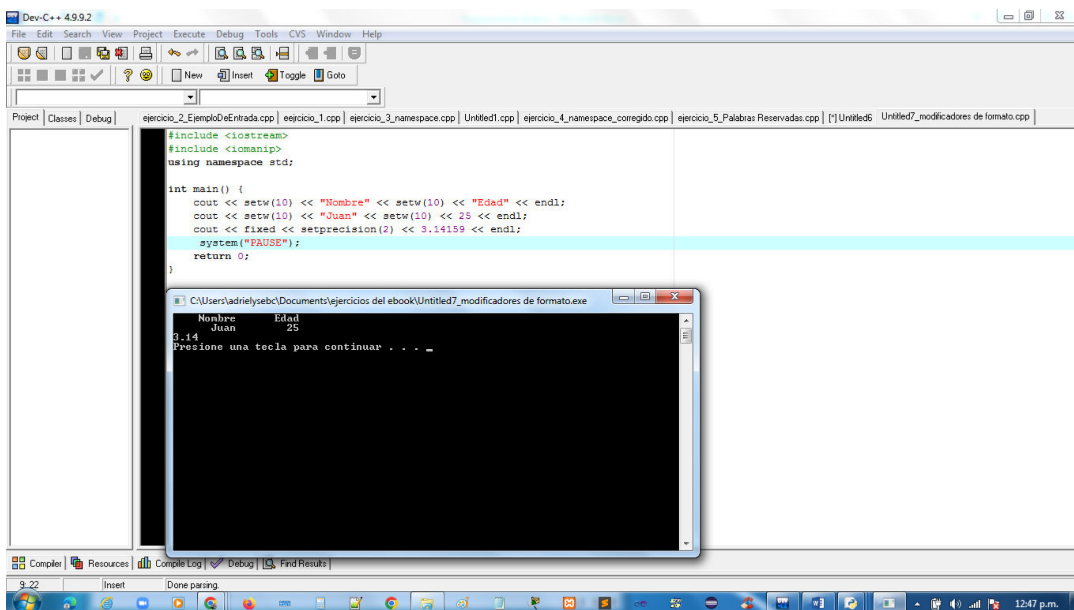
Modificadores de Formato en C++

En C++, se utilizan los manipuladores de flujo para dar formato a la salida. Algunos de los más comunes son:

- **`endl`**: Inserta un salto de línea y hace un flush del búfer.
- **`setw(n)`**: Establece el ancho de campo para el siguiente valor.
- **`setprecision(n)`**: Establece la precisión para valores de punto flotante.
- **`setfill(c)`**: Establece el carácter de relleno para campos de ancho fijo.
- **`setiosflags(flag)`**: Establece banderas de formato.

Ejemplo:

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout << setw(10) << "Nombre" << setw(10) << "Edad" << endl;
    cout << setw(10) << "Juan" << setw(10) << 25 << endl;
    cout << fixed << setprecision(2) << 3.14159 << endl;
    return 0;
}
```



Modificadores de Formato en Python

En Python, se utilizan los métodos de formato de cadenas para dar formato a la salida. Algunos de los más comunes son:

- `**{variable}**`: Inserta el valor de una variable.
- `**{variable:width**}`: Establece el ancho de campo para la variable.
- `**{variable:.precision f}**`: Establece la precisión para valores de punto flotante.
- `**{variable:character<|>|^width}**`: Alinea el valor a la izquierda, derecha o centro.

Ejemplo:

```python
nombre = "Juan"

edad = 25

pi = 3.14159


print("{:10}{:10}".format(nombre, edad))

print("{:10}{:10}".format("María", 30))

print("{:.2f}".format(pi))
```

Tanto C++ como Python ofrecen formas de dar formato a la salida de datos, aunque la sintaxis y los métodos varían entre los dos lenguajes. Los modificadores de formato permiten controlar el ancho de campo, la alineación, la precisión y otros aspectos de la presentación de los datos.

Compilación y Mapa de Memoria

Compilación:

- **En C y C++,** el código fuente se compila a código máquina.
- **En Python** se interpreta en tiempo de ejecución.

Mapa de Memoria:

- **En C/C++**, la memoria se organiza en segmentos: código, datos, heap y stack.
- **Python** maneja la memoria automáticamente.

Ejercicio Propuesto:

Describe el proceso de compilación en C++ y cómo se diferencia de la interpretación en Python.

Declaración y Tipos de Variables

C++: Se declaran con tipo explícito.

```cpp
int edad = 25;
float altura = 1.75;
char inicial = 'A';
```

```
int edad = 25;
float altura = 1.75;
char inicial = 'A';
```

Python: No requiere declaración de tipo.

```Python
edad = 25
altura = 1.75
inicial = 'A'
```

```
File   Edit   Format
edad = 25
altura = 1.75
inicial = 'A'
```

Ejercicio Propuesto:

- Crea un programa en ambos lenguajes que declare variables para un estudiante (nombre, edad, promedio) y muestre la información.

5. Sentencias de Asignación: Constantes, Operadores y Expresiones

Constantes: En C++ se definen con `const`

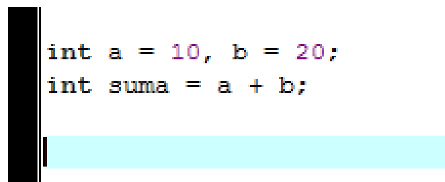
```cpp
const float PI = 3.14;
```

```
const float PI = 3.14;
```

Operadores: Ambos lenguajes utilizan operadores aritméticos, lógicos y de comparación.

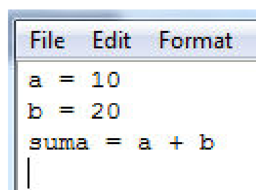
Ejemplo en C++:

```cpp
int a = 10, b = 20;
int suma = a + b;
```



Ejemplo en Python:

```Python
a = 10
b = 20
suma = a + b
```



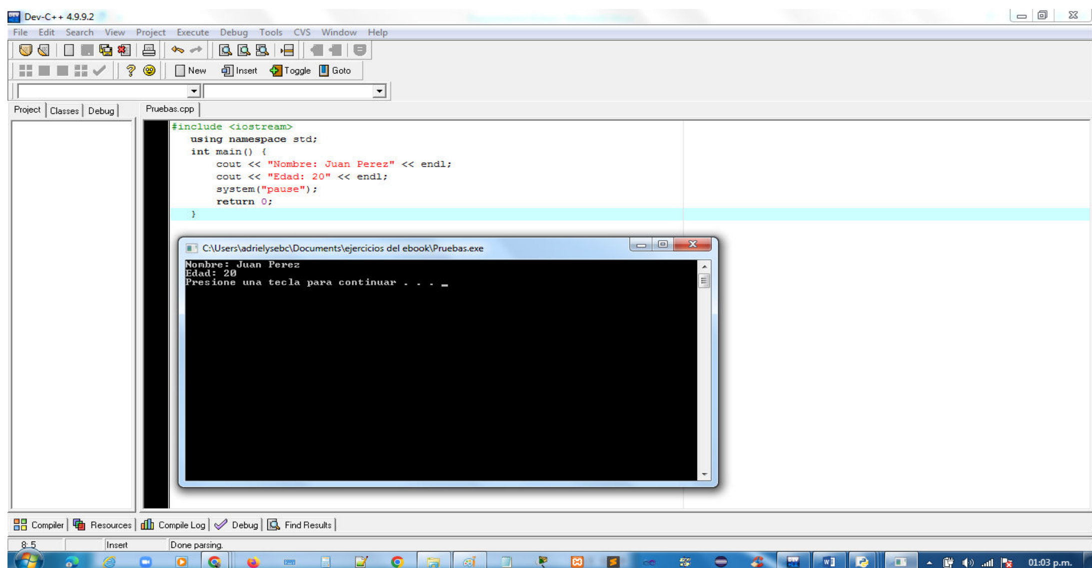
Ejercicio Propuesto:

- Escribe un programa que calcule el área de un círculo utilizando la constante PI y la fórmula `area = PI * r^2`.

Soluciones a los Ejercicios Propuestos

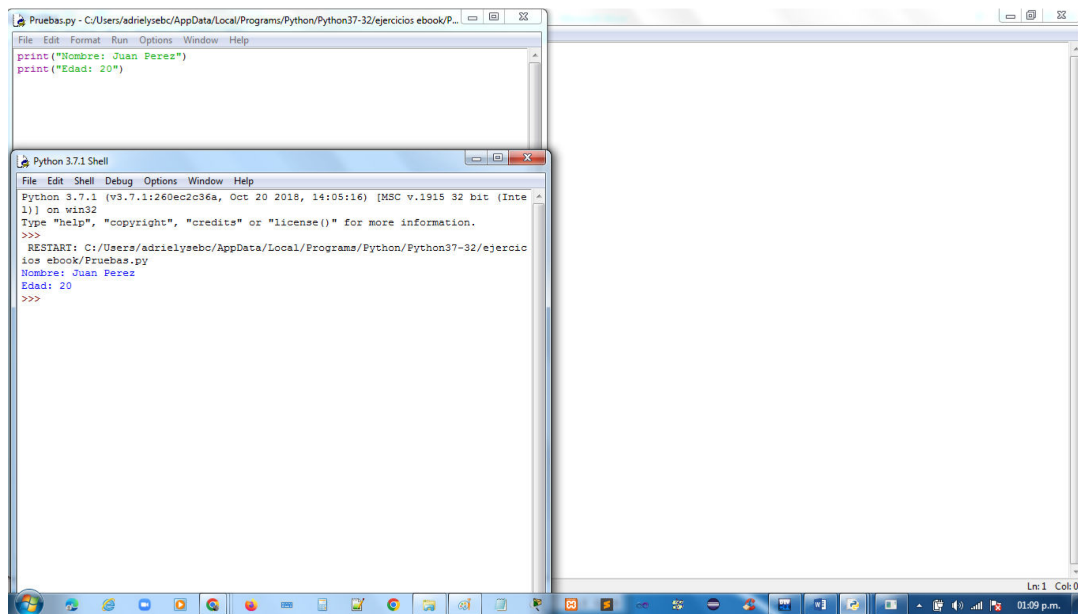
1. **Informe sobre aplicaciones de C++ y Python:** (Ejemplo de contenido a investigar y presentar).
2. **Escribe un programa en C++ y Python que imprima tu nombre y edad. C++:**

```cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Nombre: Juan Perez" << endl;
    cout << "Edad: 20" << endl;
   system("pause");
    return 0;    }
```



Python:

```python
print("Nombre: Juan Perez")
print("Edad: 20")
```



3. **Descripción del proceso de compilación: (Ejemplo de contenido a investigar y presentar).**

La compilación y el enlazado son dos procesos fundamentales en la programación de lenguajes como C y C++, mientras que Python sigue un enfoque diferente debido a su naturaleza interpretada.

Fuente:http://somebooks.es/capitulo-3-componentes-software-sistema-operativo/2/

A continuación se describe cómo se realizan estos procesos en cada lenguaje:

Compilación y Enlazado en C y C++

Compilación:

La compilación es el proceso de traducir el código fuente escrito en un lenguaje de programación (como C o C++) a código máquina que puede ser ejecutado por el sistema operativo.

En C y C++, se utilizan compiladores (como GCC o Visual C++) para llevar a cabo este proceso. El compilador analiza el código fuente, verifica errores de sintaxis y genera un archivo objeto (.o o .obj) que contiene el código máquina.

Ejemplo de Compilación en C++:

```bash
g++ -c mi_programa.cpp  # Genera un archivo objeto mi_programa.o
```

Enlazado:

El enlazado es el proceso de combinar uno o más archivos objetos generados por el compilador en un único archivo ejecutable. Este proceso también puede incluir la vinculación de bibliotecas externas.

El enlazador (linker) toma los archivos objeto y los combina, resolviendo referencias entre ellos y creando el archivo ejecutable final.

Ejemplo de Enlazado en C++:

```bash
g++ mi_programa.o -o mi_programa  # Genera el ejecutable mi_programa
```

Sigue el siguiente enlace para saber más sobre este proceso:

https://www.youtube.com/watch?v=EghLfJqG92Q

Compilación y Ejecución en Python

Interpretación:

Python es un lenguaje interpretado, lo que significa que no se compila a código máquina de la misma manera que C o C++. En su lugar, el código fuente se ejecuta directamente por el intérprete de Python.

Cuando ejecutas un script de Python, el intérprete lee el código línea por línea y lo ejecuta en tiempo real.

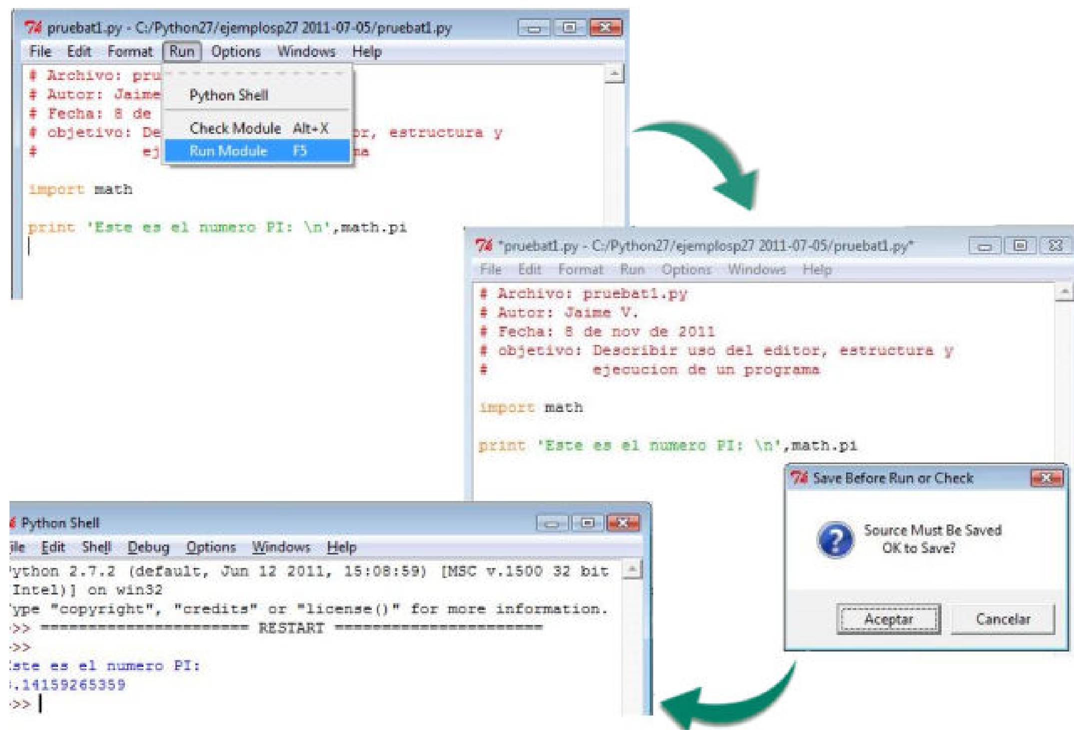
Ejemplo de Ejecución en Python:

```bash
python mi_programa.py  # Ejecuta directamente el script
```

Compilación a Bytecode:

Aunque Python no necesita un proceso de compilación explícito, el intérprete convierte el código fuente en bytecode, que es una representación intermedia. Este bytecode se almacena en archivos con extensión `.pyc` en un directorio `__pycache__`.

Este proceso de compilación a bytecode ocurre automáticamente cuando se ejecuta un script de Python.



Fuente:http://cursosonline.cartagena99.com/file.php/10/Sitio_algoritmos_programacion

Tabla 3

Comparación Resumida

Aspecto	C/C++	Python
Tipo de Lenguaje	Compilado	Interpretado
Proceso	Compilación → Enlazado → Ejecución	Interpretación directa
Producción de Archivos	Genera archivos objeto y ejecutables	Genera bytecode (.pyc)
Ejemplo de Compilación	`g++ -c mi_programa.cpp`	No requiere compilación explícita
Ejemplo de Ejecución	`./mi_programa`	`python mi_programa.py`

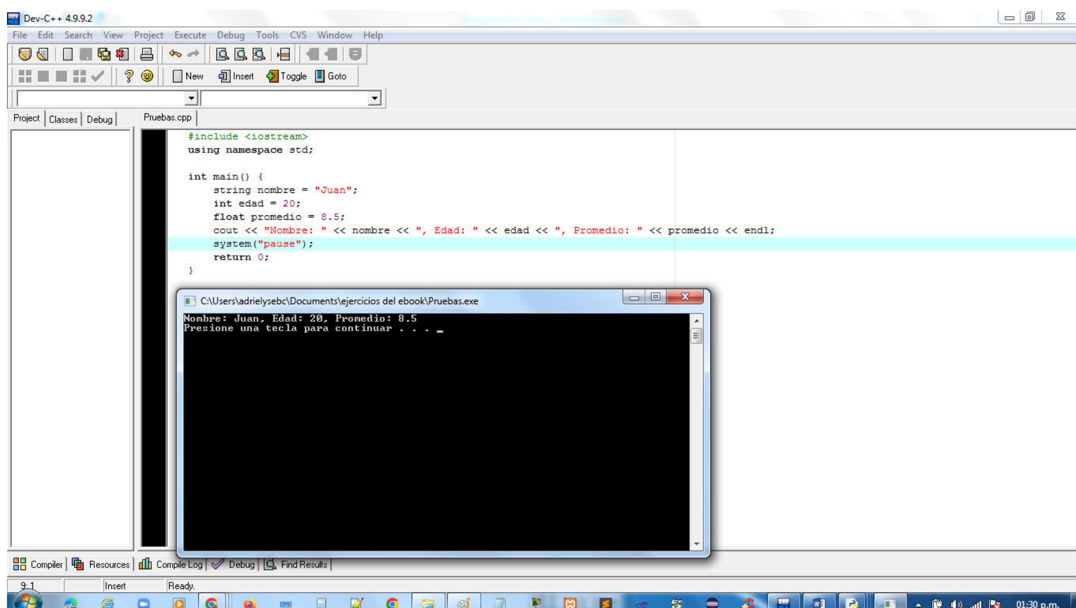
Fuente: *https://www.perplexity.ai*

La compilación y el enlazado son procesos esenciales en C y C++, que permiten convertir el código fuente en ejecutables. En contraste, Python utiliza un enfoque interpretado que permite una ejecución más flexible y dinámica, aunque con un rendimiento a veces inferior en comparación con los lenguajes compilados. Cada enfoque tiene sus ventajas y desventajas, y la elección entre ellos depende de las necesidades específicas del proyecto y del entorno de desarrollo.

4. Programa que declare variables:

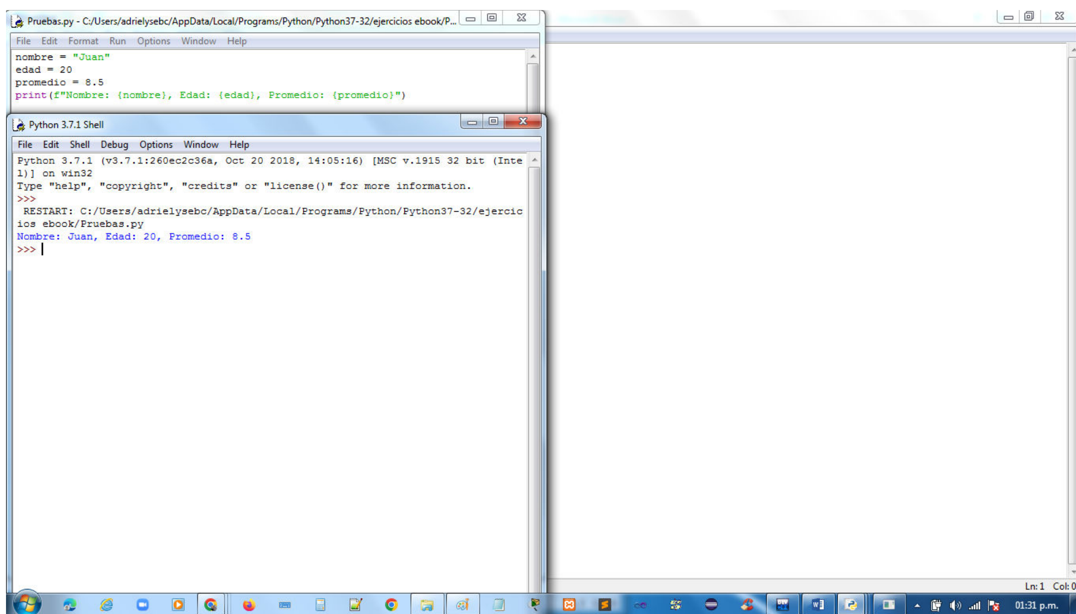
C++:

```cpp
#include <iostream>

using namespace std;

int main() {

    string nombre = "Juan";

    int edad = 20;

    float promedio = 8.5;

    cout << "Nombre: " << nombre << ", Edad: " << edad << ", Promedio: " <<
promedio << endl;

    return 0;

}
```



Python:

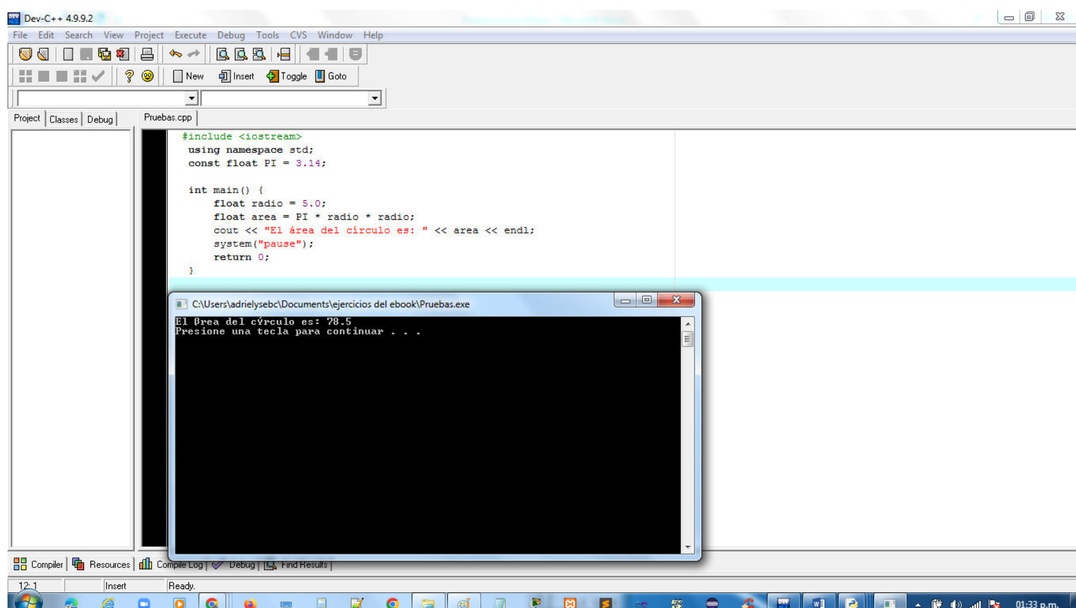
```python
nombre = "Juan"

edad = 20

promedio = 8.5

print(f"Nombre: {nombre}, Edad: {edad}, Promedio: {promedio}")
```



5. Cálculo del área de un círculo:

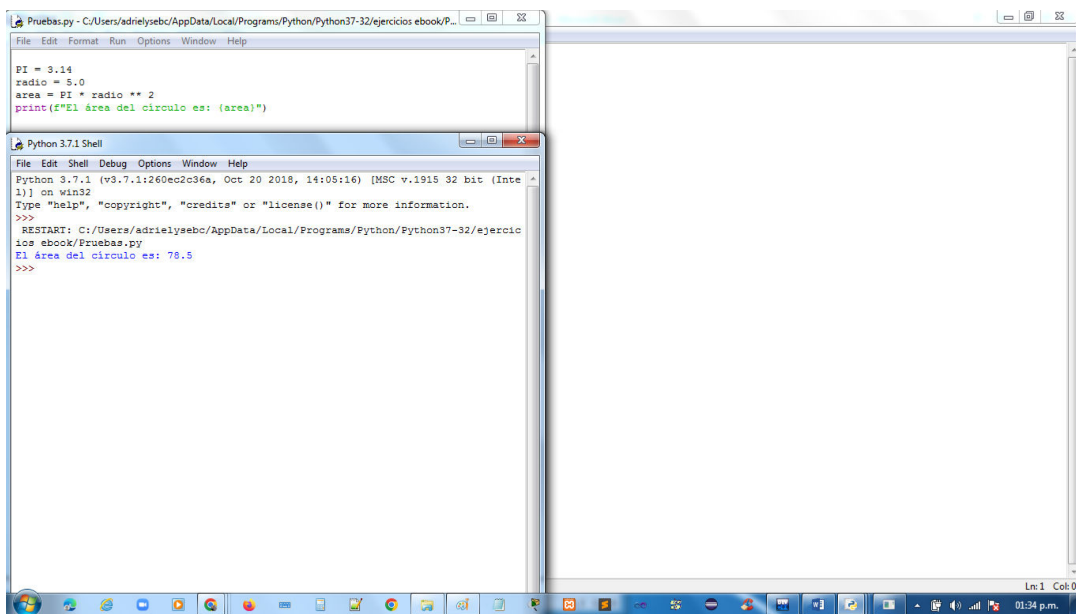
C++:

```cpp
#include <iostream>

using namespace std;

const float PI = 3.14;


int main() {

    float radio = 5.0;

    float area = PI * radio * radio;

    cout << "El área del círculo es: " << area << endl;

    return 0;
```



Python:

```python
PI = 3.14

radio = 5.0

area = PI * radio ** 2

print(f"El área del círculo es: {area}")
```



Unidad II

Funciones y Expresiones en C++ y Python

En esta unidad se presentaran una serie de ejercicios que muestran cómo aplicar los conceptos de funciones, constantes, operadores y recursividad en C++ y Python para resolver problemas específicos. Se recomienda proponer más ejercicios y desafíos para practicar y afianzar estos conocimientos tanto en el aula de clases como en el hogar.

Funciones de Control: Constantes, Operadores y Expresiones

Las funciones de control en C++ y Python permiten ejecutar bloques de código bajo ciertas condiciones. Algunos elementos clave son:

Constantes:

Valores fijos que no pueden ser modificados durante la ejecución del programa.

- **En C++:** se declaran con `const`: `const float PI = 3.14;`

```
const float PI = 3.14;`
```

- **En Python:** se asignan valores directamente: `PI = 3.14`

```
PI = 3.14
```

Operadores:

Símbolos que realizan operaciones aritméticas, lógicas y de asignación.

Ejemplos: `+`, `-`, `*`, `/`, `=`, `<`, `>`, `and`, `or`, `not`

Expresiones:

Combinación de variables, constantes, operadores y funciones que se evalúan a un valor.

Ejemplo: `area = PI * radio * radio`

Función "Main"

En Python:

No se requiere una función **`main()`** específica, el programa se ejecuta secuencialmente desde arriba hacia abajo.

En C++:

La función **`main()`** es el punto de entrada de un programa en C++. Es obligatoria y debe retornar un valor entero.

```cpp
int main() {
    // Código del programa
    return 0; // Indica ejecución exitosa
}
```

```
int main() {
    // Código del programa
    return 0; // Indica ejecución exitosa
  }
```

"C" como Bloque de Funciones:

Funciones como Procedimiento

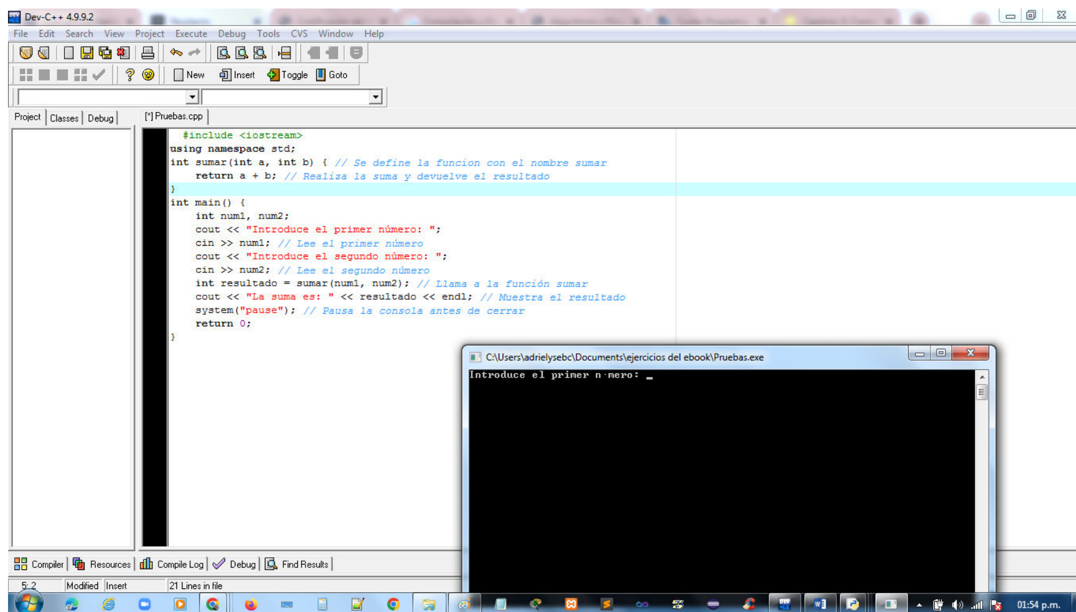
En C++, las funciones se definen con un tipo de retorno, un nombre y parámetros opcionales.

```
#include <iostream>

using namespace std;

int sumar(int a, int b) {

   return a + b; // Realiza la suma y devuelve el resultado

}

int main() {

   int num1, num2;

   cout << "Introduce el primer número: ";

   cin >> num1; // Lee el primer número

   cout << "Introduce el segundo número: ";

   cin >> num2; // Lee el segundo número

   int resultado = sumar(num1, num2); // Llama a la función sumar

   cout << "La suma es: " << resultado << endl; // Muestra el resultado
```

system("pause"); // Pausa la consola antes de cerrar

return 0;

}



En Python, la sintaxis es similar, pero no se especifica el tipo de retorno.

```Python

def sumar(a, b):

return a + b

 # Solicitar al usuario que ingrese dos números

num1 = float(input("Introduce el primer número: "))

num2 = float(input("Introduce el segundo número: "))

```
# Llamar a la función sumar y almacenar el resultado

resultado = sumar(num1, num2)

# Mostrar el resultado

print("La suma es:", resultado)
```

Nota: Las funciones pueden ser llamadas desde cualquier parte del programa.

Tabla 4

Comparación de Funciones

Caracteristica	C++	Python
Declaración	Tipo de retorno, nombre y parámetros	`def` seguido del nombre
Tipo de Retorno	Debe ser explícito	Puede ser implícito
Llamada	`nombre_funcion(argumentos)`	`nombre_funcion(argumentos)`
Recursividad	Soportada	Soportada
Funciones Anónimas	No tiene soporte nativo	Soportadas mediante `lambda`

Fuente: *https://www.perplexity.ai*

Recursividad en las Funciones

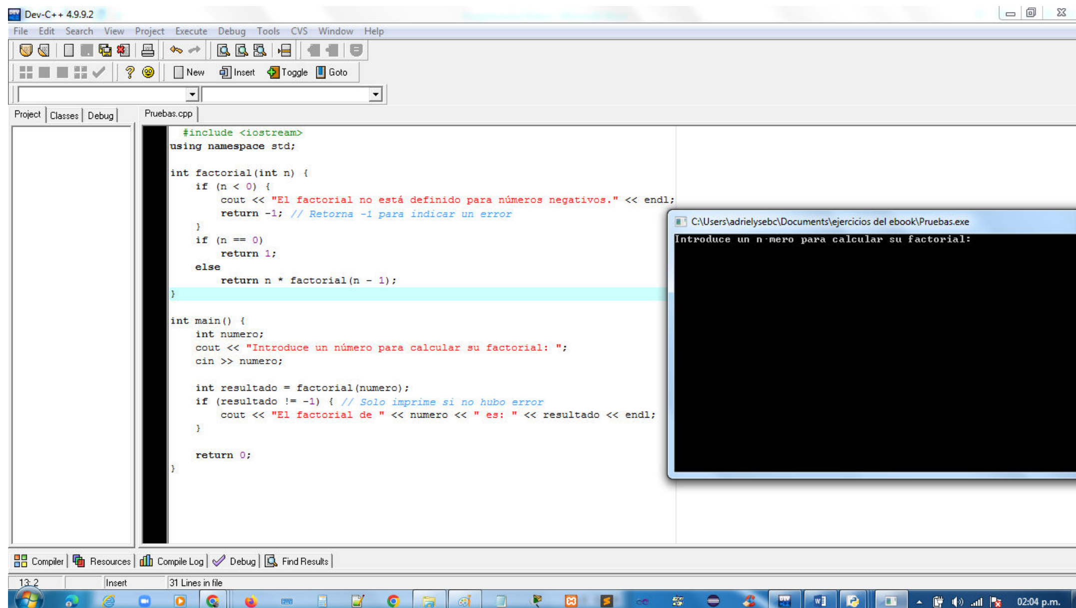
La recursividad es cuando una función se llama a sí misma. Útil para resolver problemas complejos de manera elegante.

Ejemplo de factorial recursivo

en C++:

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


int factorial(int n) {
    if (n < 0) {
```

```
        cout << "El factorial no está definido para números negativos." << endl;
        return -1; // Retorna -1 para indicar un error
    }
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
int main() {
    int numero;
    cout << "Introduce un número para calcular su factorial: ";
    cin >> numero;
    int resultado = factorial(numero);
    if (resultado != -1) { // Solo imprime si no hubo error
        cout << "El factorial de " << numero << " es: " << resultado << endl;
    }
    return 0;
}
```



Explicación del código:

1. `#include <iostream>`: Esto es necesario para usar `std::cout` y `std::cin`.

2. Validación de Entrada: Se agregó una verificación para asegurarse de que el número no sea negativo. Si es negativo, se imprime un mensaje de error.

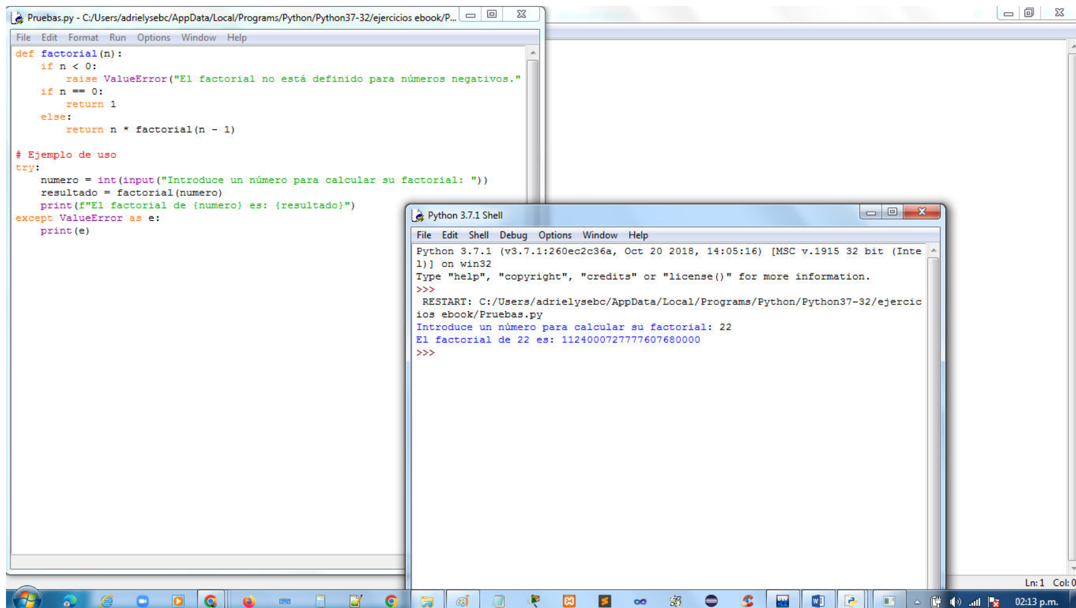
3. Función `main()`: Se añadió la función `main()` para que el programa tenga un punto de entrada. En esta función, se solicita al usuario que introduzca un número, se calcula el factorial y se imprime el resultado.

4. Manejo de Errores: Se utiliza un valor de retorno de `-1` para indicar un error en el cálculo del factorial, lo que permite al programa manejar la situación adecuadamente.

Este código está completo y listo para ejecutarse, proporcionando un manejo adecuado de la entrada y salida.

Ejemplo en Python:

```python
def factorial(n):
    if n < 0:
        raise ValueError("El factorial no está definido para números negativos.")
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
# Ejemplo de uso
try:
    numero = int(input("Introduce un número para calcular su factorial: "))
    resultado = factorial(numero)
    print(f"El factorial de {numero} es: {resultado}")
except ValueError as e:
print(e)
```



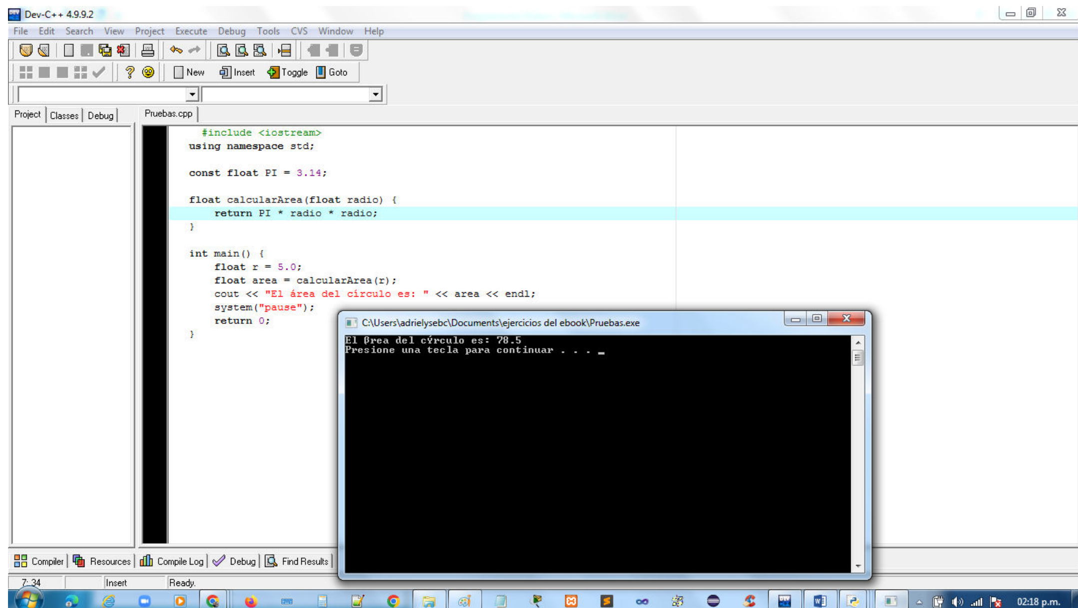
Pruebas.py - C:/Users/adrielysebc/AppData/Local/Programs/Python/Python37-32/ejercicios ebook/P...
File Edit Format Run Options Window Help
def factorial(n):
    if n < 0:
        raise ValueError("El factorial no está definido para números negativos.")
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
# Ejemplo de uso
try:
    numero = int(input("Introduce un número para calcular su factorial: "))
    resultado = factorial(numero)
    print(f"El factorial de {numero} es: {resultado}")
except ValueError as e:
    print(e)
Python 3.7.1 Shell
File Edit Shell Debug Options Window Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/adrielysebc/AppData/Local/Programs/Python/Python37-32/ejercicios ebook/Pruebas.py
Introduce un número para calcular su factorial: 22
El factorial de 22 es: 1124000727777607680000
>>>
Ln: 1 Col: 0
02:13 p.m.

Ejercicios Propuestos y Soluciones

1. Escribir una función que calcule el área de un círculo dado su radio.

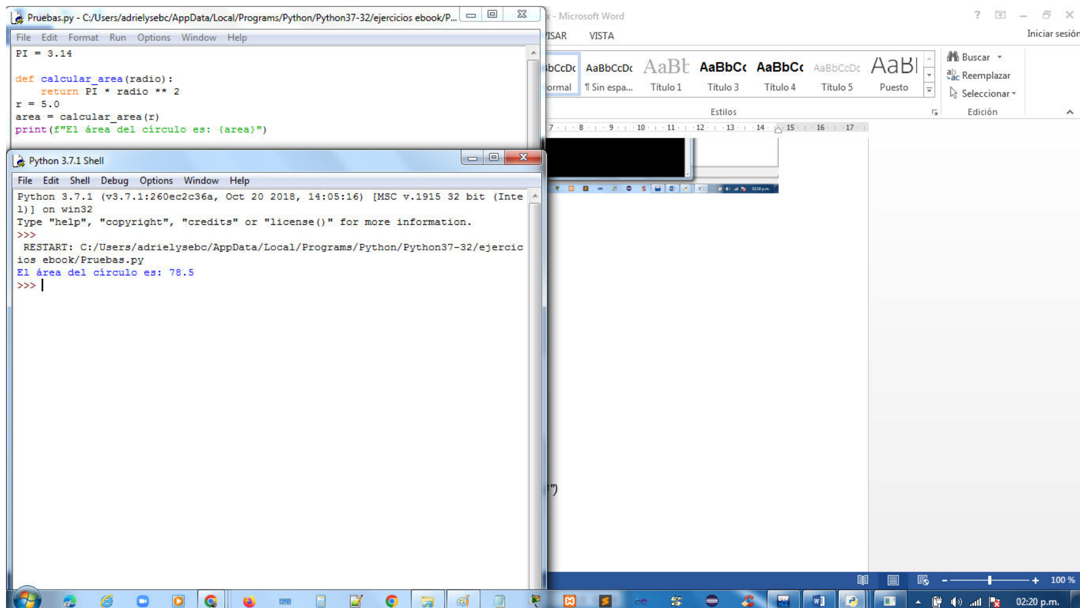
Solución en C++:

```cpp
#include <iostream>
using namespace std;
const float PI = 3.14;
float calcularArea(float radio) {
    return PI * radio * radio;
}
int main() {
    float r = 5.0;
    float area = calcularArea(r);
    cout << "El área del círculo es: " << area << endl;
    return 0;
}
```



Solución en Python:

```python
PI = 3.14

def calcular_area(radio):

    return PI * radio ** 2

r = 5.0

area = calcular_area(r)

print(f"El área del círculo es: {area}")
```



2. Crear una función recursiva que calcule la suma de los primeros n números naturales.

Solución en C++:

```cpp
#include <iostream>

using namespace std;

int sumaRecursiva(int n) {

    if (n == 1)

        return 1;

    else

        return n + sumaRecursiva(n-1);

}
```

int main() {

int n = 10;

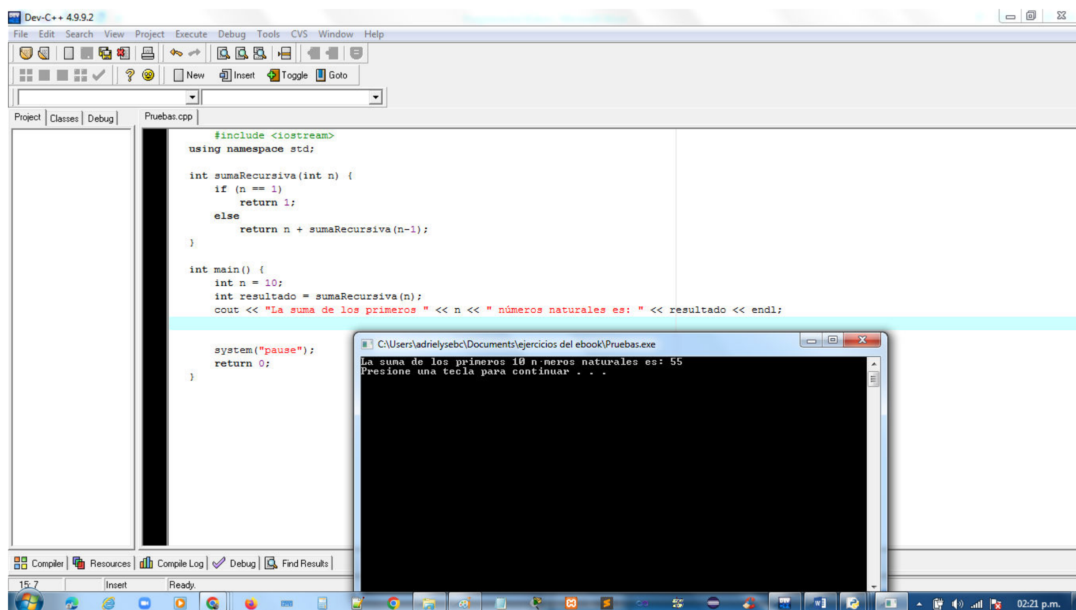
int resultado = sumaRecursiva(n);

cout << "La suma de los primeros " << n << " números naturales es: " << resultado << endl;

return 0;

}

```

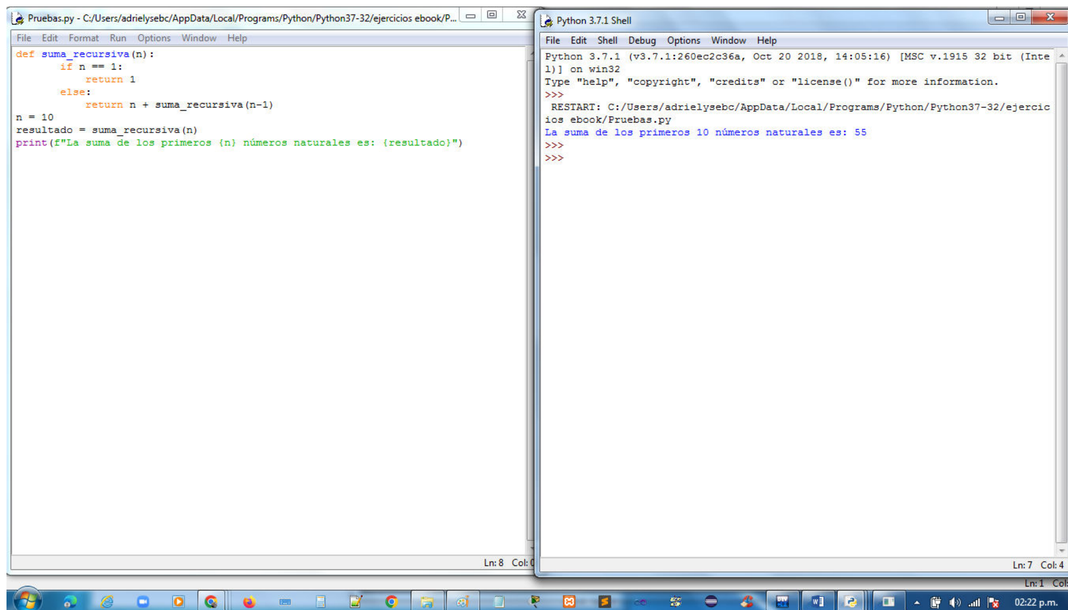



**Solución en Python:**

```python

def suma_recursiva(n):

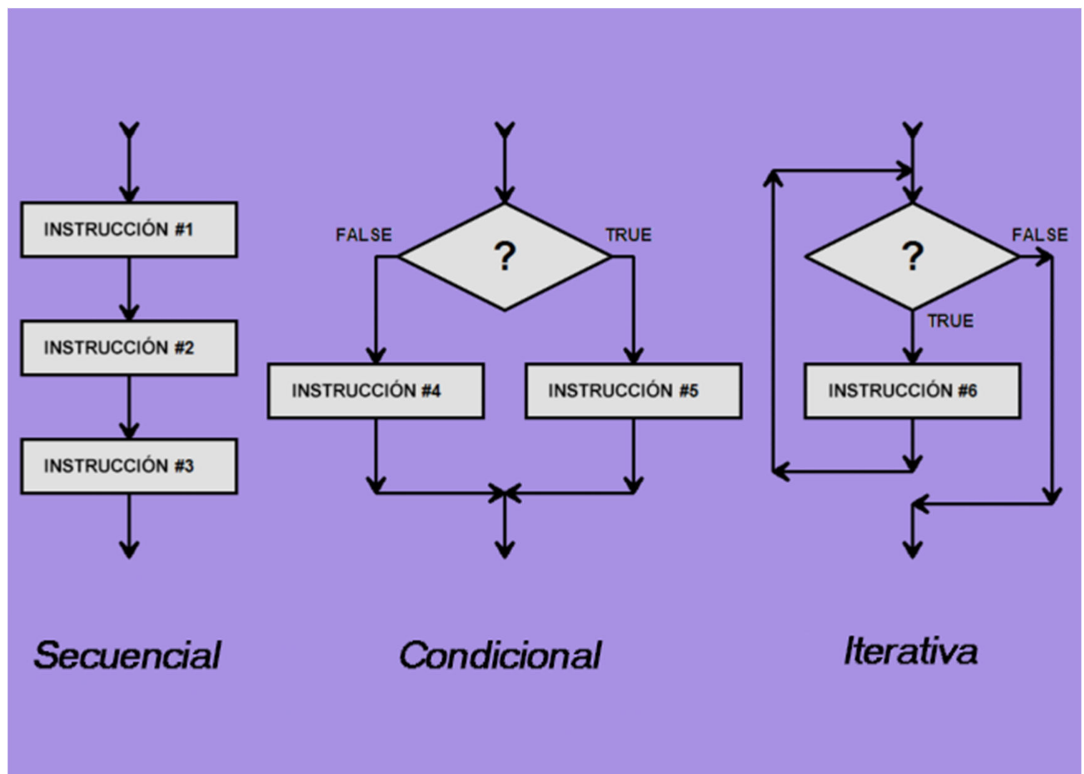
```
    if n == 1:
        return 1
    else:
        return n + suma_recursiva(n-1)
n = 10
resultado = suma_recursiva(n)
print(f"La suma de los primeros {n} números naturales es: {resultado}")
```



UNIDAD III

CONTROL DE FLUJO EN C++ Y PYTHON

Este capítulo proporcionara un marco teórico y práctico sobre las estructuras de control en C++ y Python, junto con ejemplos y ejercicios que permiten a los estudiantes aplicar lo aprendido.



Fuente: https://procomsys.wordpress.com/2018/05/27/estructuras-de-control-ejemplos-practicos-en-c/

Partes de una Estructura de Control

Las estructuras de control permiten dirigir el flujo de ejecución de un programa. Las partes principales de una estructura de control incluyen:

Condición: Expresión que se evalúa como verdadera o falsa.

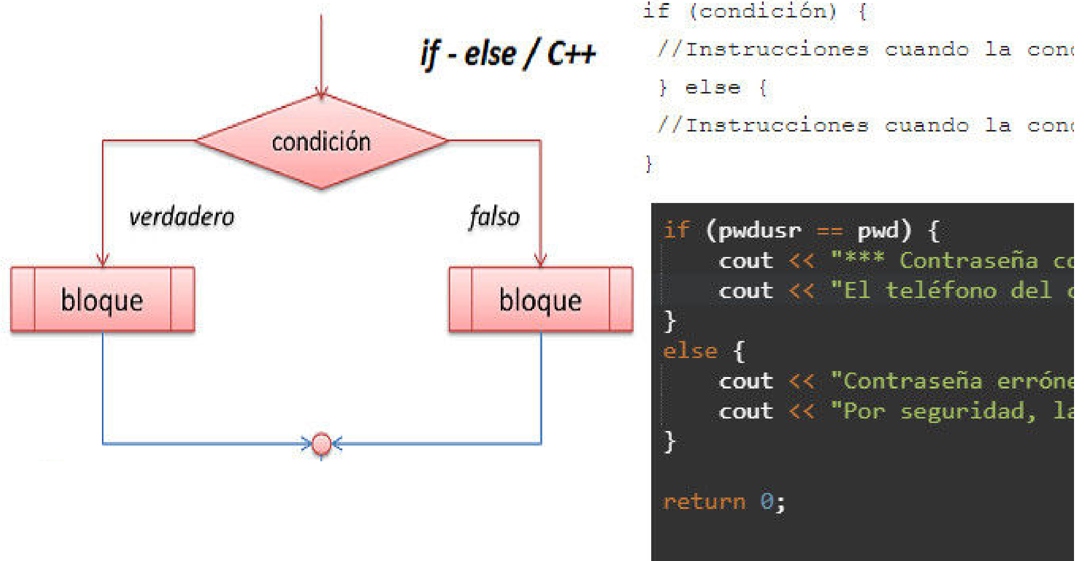
Bloque de Código: Conjunto de instrucciones que se ejecutan si la condición es verdadera.

Alternativa: Instrucciones que se ejecutan si la condición es falsa (en estructuras como `if/else`).

Estructuras de Selección: `if` / `else`

Las estructuras de selección permiten ejecutar diferentes bloques de código según el resultado de una condición.

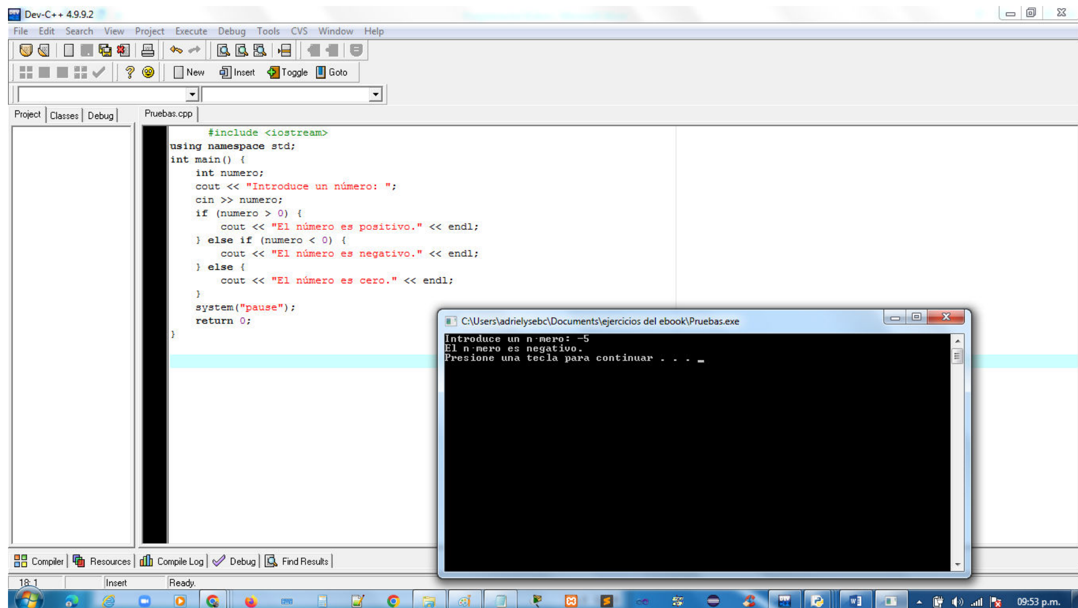
La sentencia if / else controla las 2 posibilidades de una decisión, if ejecuta las instrucciones cuando la condición es verdadera, y else ejecuta las instrucciones para el caso en que la condición es falsa.



Fuente:https://es.linkedin.com/pulse/estructura-else-en-c-roberto-c-onz

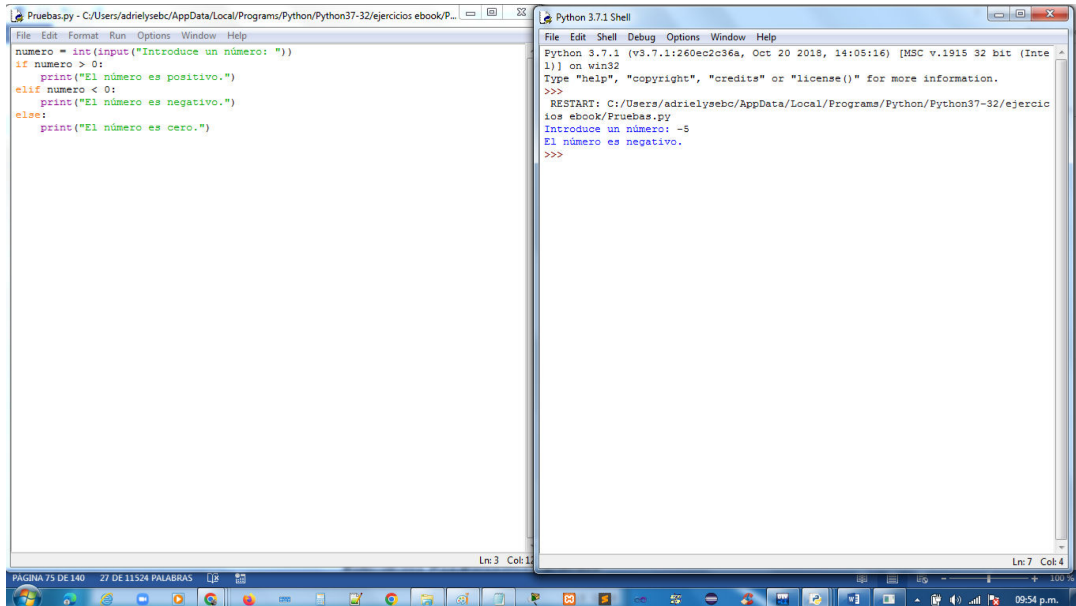
Ejemplo en C++:

```cpp
#include <iostream>
using namespace std;
int main() {
    int numero;
    cout << "Introduce un número: ";
    cin >> numero;
    if (numero > 0) {
        cout << "El número es positivo." << endl;
    } else if (numero < 0) {
        cout << "El número es negativo." << endl;
    } else {
        cout << "El número es cero." << endl;
    }
    return 0;
}
```



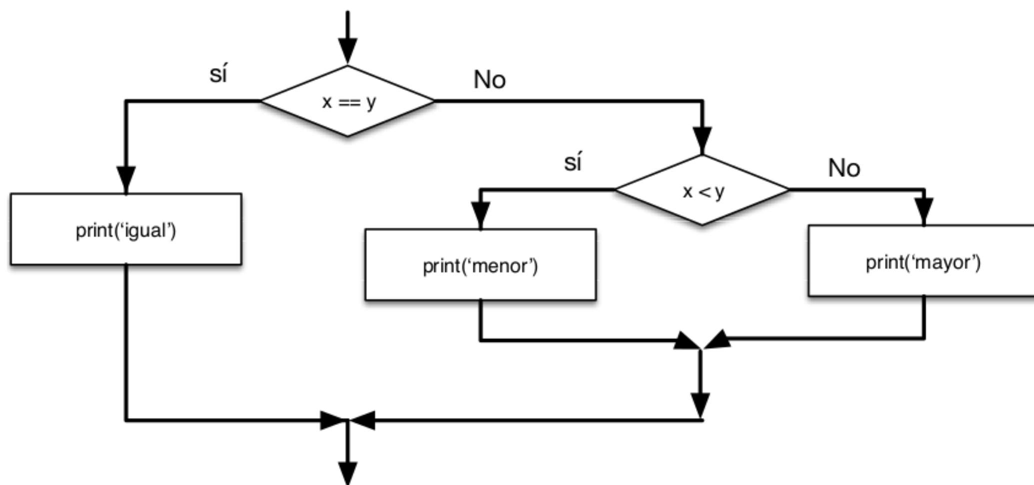
Ejemplo en Python:

```Python
numero = int(input("Introduce un número: "))
if numero > 0:
    print("El número es positivo.")
elif numero < 0:
    print("El número es negativo.")
else:
    print("El número es cero.")
```



Estructuras Condicionales Anidadas

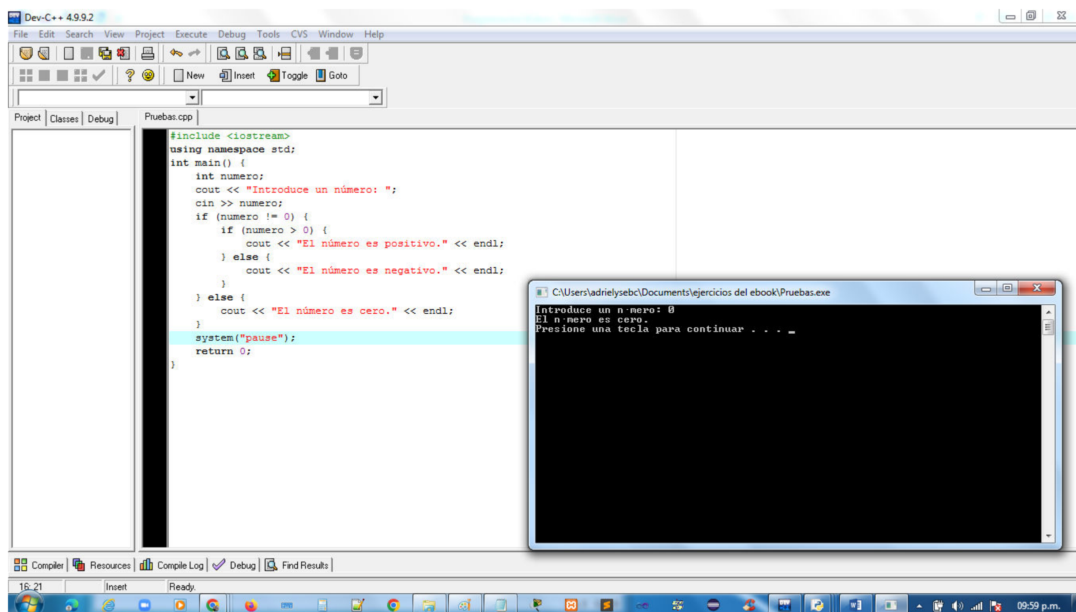
Las estructuras condicionales anidadas permiten incluir una estructura de control dentro de otra.



Fuente: https://www.tutorialesprogramacionya.com

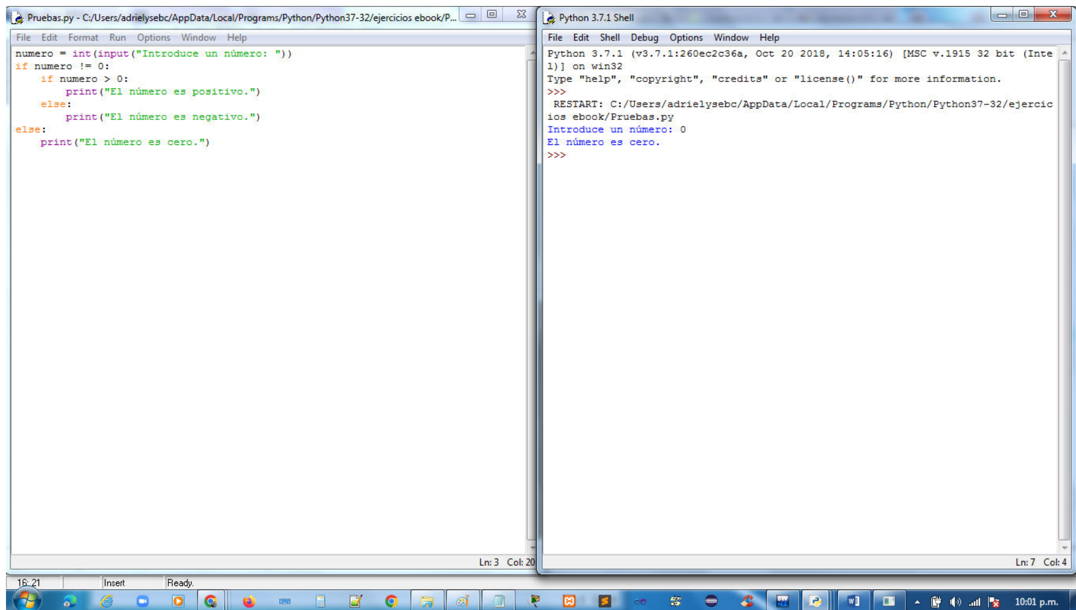
Ejemplo en C++:

```cpp
#include <iostream>
using namespace std;
int main() {
    int numero;
    cout << "Introduce un número: ";
    cin >> numero;
    if (numero != 0) {
        if (numero > 0) {
            cout << "El número es positivo." << endl;
        } else {
            cout << "El número es negativo." << endl;
        }
    } else {
        cout << "El número es cero." << endl;
    }
    return 0;
}
```



Ejemplo en Python:

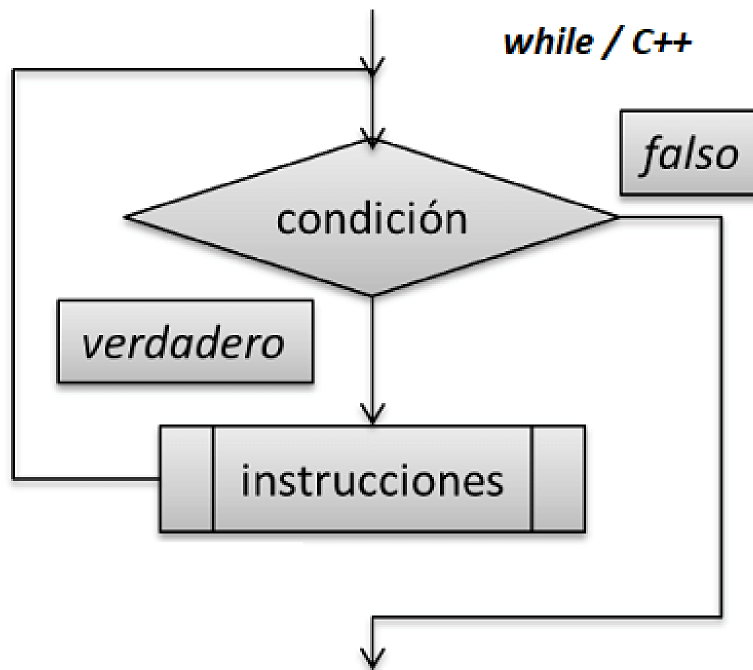
```Python
numero = int(input("Introduce un número: "))
if numero != 0:
    if numero > 0:
        print("El número es positivo.")
    else:
        print("El número es negativo.")
else:
    print("El número es cero.")
```



Estructura `While`, `For` y `Do While`

Estructura `While`

La estructura `while` ejecuta un bloque de código mientras la condición sea ve rdadera.

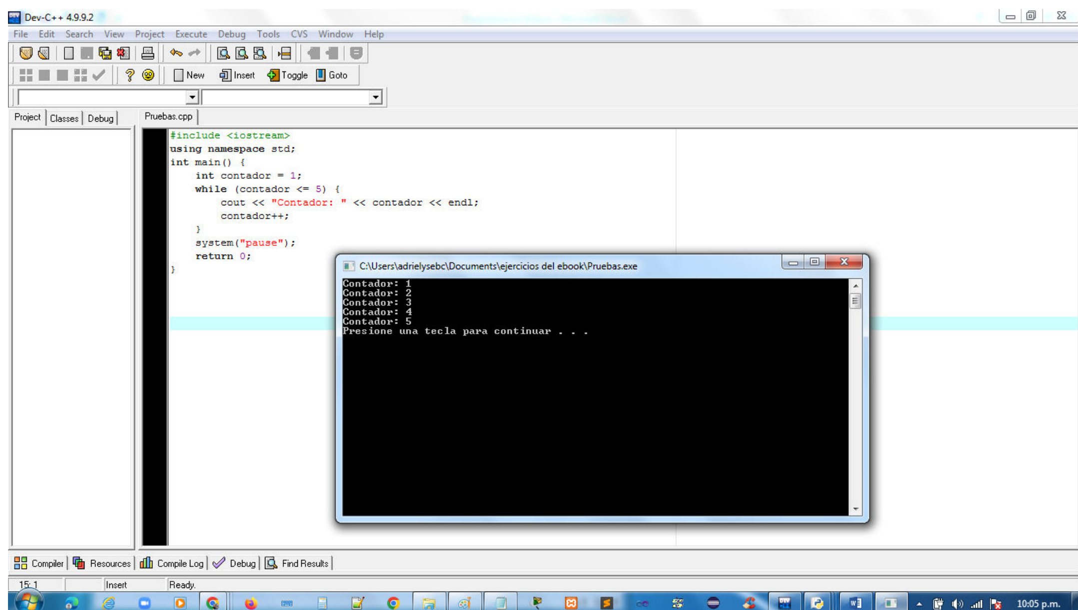


Fuente:*https://es.linkedin.com/pulse/estructura-else-en-c-roberto-c-onz*

Ejemplo en C++:

```cpp
#include <iostream>

using namespace std;

int main() {

   int contador = 1;

   while (contador <= 5) {
```

```
        cout << "Contador: " << contador << endl;

        contador++;

    }

    return 0;

}
```



Ejemplo en Python:

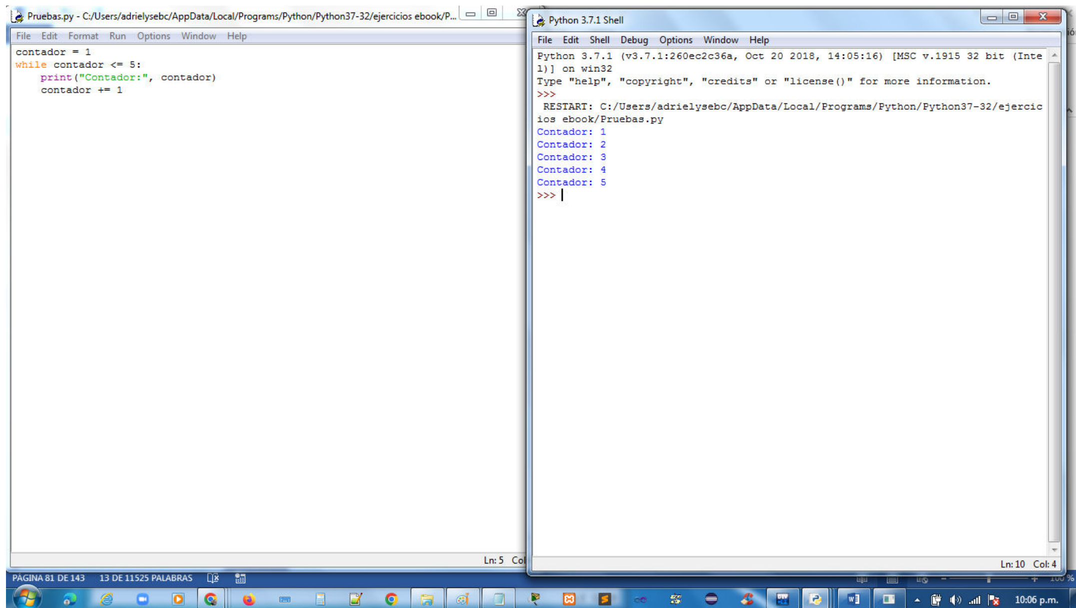
```python
contador = 1

while contador <= 5:

    print("Contador:", contador)
```

contador += 1

````
```
````



Estructura `For`

La estructura `for` se utiliza para iterar sobre un rango de valores.

Estructura for

Sirve *para* que cierto bloque de instrucciones se ejecute una y otra vez, siempre y cuando la variable a evaluar en la condición se cumpla. Para que no ocurra un bucle infinito se debe tener en cuenta un límite a declarar.

```
>> for (inicio; límite; salto)
   { Acción; }
```

Salto de la variable

Acción

Cumple

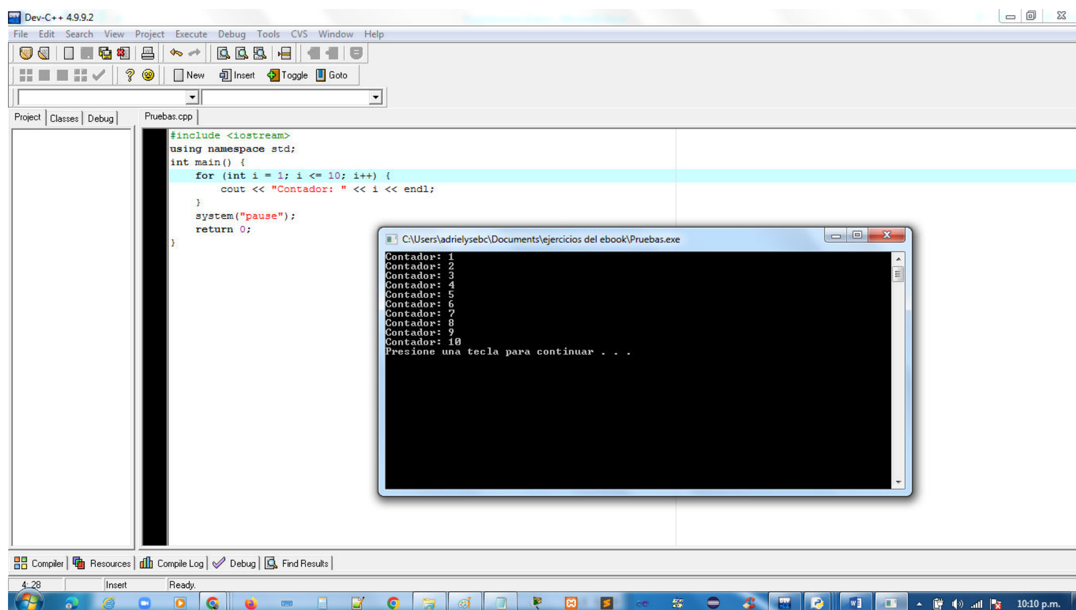
Evaluación de variable

No cumple
```

Fuente: https://www.youtube.com/watch?v=sc_C8Oo9mTE

Ejemplo en C++:

```cpp
#include <iostream>

using namespace std;

int main() {

    for (int i = 1; i <= 5; i++) {

        cout << "Contador: " << i << endl;

    }

    return 0;

}
```



Ejemplo en Python:

```Python
for i in range(1, 6):
    print("Contador:", i)
```

```
for i in range(1, 10):
    print("Contador:", i)
```

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5
Contador: 6
Contador: 7
Contador: 8
Contador: 9
```

Estructura `Do While`

La estructura `do while` ejecuta el bloque de código al menos una vez antes de evaluar la condición.

Estructura do while

Realiza el mismo papel que un *while*. La única diferencia es que *do while* primero *realiza* una acción y después evalúa una condición. Ésta acción se sigue ejecutando mientras la condición se cumpla.

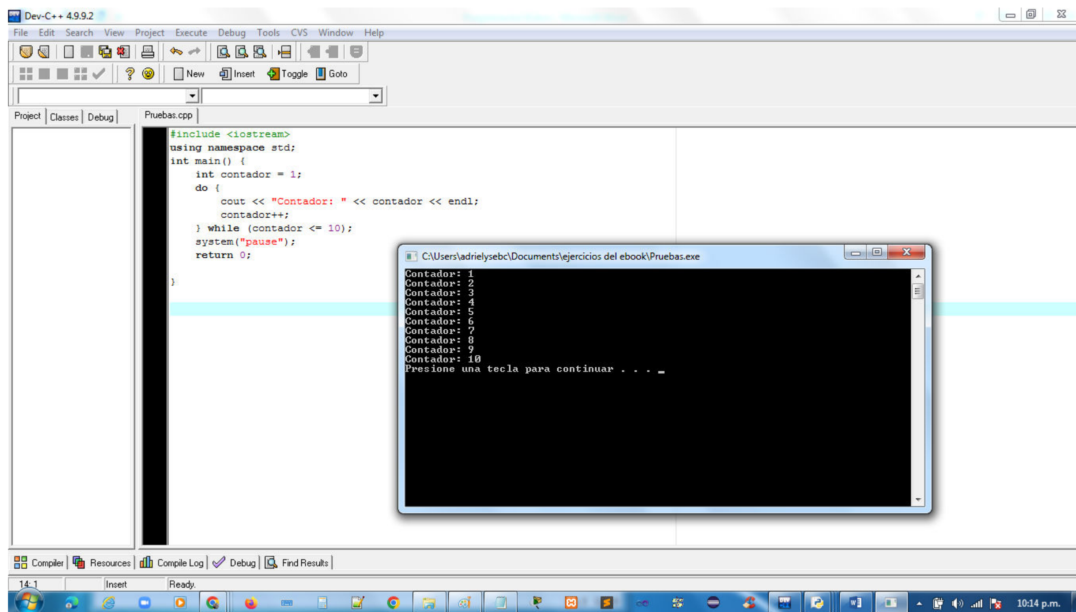
```
>> do { Acción; }
   while (condición);
```

Fuente: https://www.youtube.com/watch?v=ybNRakBzTAc

Ejemplo en C++:

```cpp
#include <iostream>

using namespace std;

int main() {

    int contador = 1;

    do {

        cout << "Contador: " << contador << endl;

        contador++;

    } while (contador <= 5);

    return 0;

}
```

```
```

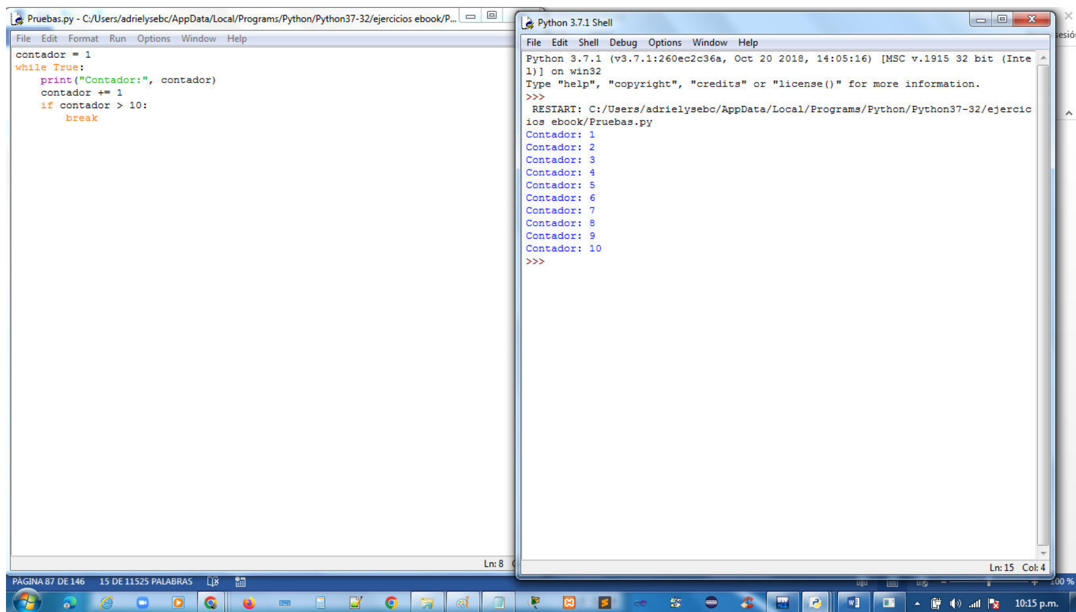


Ejemplo en Python:

Python no tiene una estructura `do while` nativa, pero se puede simular con un bucle `while`:

```Python
contador = 1
while True:
    print("Contador:", contador)
    contador += 1
    if contador > 5:
        break
```

El uso del **break()** en Python nos permite terminar prematuramente la ejecución de un bucle for o while.



Resumen sobre las estructuras condicionales en C++:

Estructuras de Selección en C++

C++ utiliza las siguientes estructuras de selección:

if: Evalúa una condición y ejecuta un bloque de código si es verdadera.

```cpp
if (condicion) {
    // Código a ejecutar si la condición es verdadera
}
```

if-else: *Evalúa una condición y ejecuta un bloque de código si es verdadera, de lo contrario ejecuta otro bloque.*

```cpp
if (condicion) {
    // Código a ejecutar si la condición es verdadera
} else {
    // Código a ejecutar si la condición es falsa
}
```

if-else if-else: Permite evaluar múltiples condiciones de forma secuencial.

```cpp
if (condicion1) {
    // Código a ejecutar si condicion1 es verdadera
} else if (condicion2) {
    // Código a ejecutar si condicion1 es falsa y condicion2 es verdadera
} else {
    // Código a ejecutar si todas las condiciones son falsas
}
```

Estructuras Condicionales Anidadas

Las estructuras condicionales pueden anidarse para crear condiciones más complejas:

```cpp
if (condicion1) {
    if (condicion2) {
        // Código a ejecutar si condicion1 y condicion2 son verdaderas
    } else {
        // Código a ejecutar si condicion1 es verdadera y condicion2 es falsa
    }
} else {
    // Código a ejecutar si condicion1 es falsa
}
```

Operadores de Comparación

C++ utiliza los siguientes operadores de comparación en las estructuras condicionales:

- **`==`** *(igual a)*
- **`!=`** *(distinto de)*
- **`>`** *(mayor que)*
- **`<`** *(menor que)*
- **`>=`** *(mayor o igual que)*
- **`<=`** *(menor o igual que)*

Operadores Lógicos

También se pueden utilizar operadores lógicos para combinar condiciones:

- **`&&`** *(and)*
- **`||`** *(or)*
- **`!`** *(not)*

Ejemplo:

```cpp
if (edad >= 18 && edad <= 65) {
   // Código a ejecutar si la edad está entre 18 y 65 años
}
```

Las estructuras condicionales en C++ permiten ejecutar diferentes bloques de código según el resultado de una o más condiciones, utilizando operadores de comparación y lógicos. Estas estructuras son fundamentales para el control de flujo en los programas.

Estructuras condicionales en Python:

Estructuras de Selección en Python

Python utiliza las siguientes estructuras de selección:

if: Evalúa una condición y ejecuta un bloque de código si es verdadera.

```python
if condicion:
```

```
    # Código a ejecutar si la condición es verdadera
```

if-else: Evalúa una condición y ejecuta un bloque de código si es verdadera, de lo contrario ejecuta otro bloque.

```python
if condicion:
    # Código a ejecutar si la condición es verdadera
else:
    # Código a ejecutar si la condición es falsa
```

if-elif-else: Permite evaluar múltiples condiciones de forma secuencial.

```python
if condicion1:
    # Código a ejecutar si condicion1 es verdadera
elif condicion2:
    # Código a ejecutar si condicion1 es falsa y condicion2 es verdadera
else:
    # Código a ejecutar si todas las condiciones son falsas
```

Estructuras Condicionales Anidadas

Las estructuras condicionales pueden anidarse para crear condiciones más complejas:

```python
if condicion1:
    if condicion2:
        # Código a ejecutar si condicion1 y condicion2 son verdaderas
    else:
        # Código a ejecutar si condicion1 es verdadera y condicion2 es falsa
else:
    # Código a ejecutar si condicion1 es falsa
```

Operadores de Comparación

Python utiliza los siguientes operadores de comparación en las estructuras condicionales:

- `==` (igual a)
- `!=` (distinto de)
- `>` (mayor que)
- `<` (menor que)
- `>=` (mayor o igual que)
- `<=` (menor o igual que)

Operadores Lógicos

También se pueden utilizar operadores lógicos para combinar condiciones:

- `and`: Verdadero si ambas condiciones son verdaderas.
- `or`: Verdadero si al menos una condición es verdadera.
- `not`: Niega el valor de una condición.

Ejemplo:

```python
if edad >= 18 and edad <= 65:
   # Código a ejecutar si la edad está entre 18 y 65 años
```

Las estructuras condicionales en Python permiten ejecutar diferentes bloques de código según el resultado de una o más condiciones, utilizando operadores de comparación y lógicos. Estas estructuras son fundamentales para el control de flujo en los programas.

EJERCICIOS PROPUESTOS

1. Ejercicio de Selección: Escribe un programa que pida un número y determine si es par o impar.

2. Ejercicio de Condicional Anidado: Escribe un programa que pida la calificación de un estudiante y muestre si ha aprobado (mayor o igual a 60) o no, y si está en rango de 0 a 100.

3. Ejercicio con `While`: Escribe un programa que imprima los números del 1 al 10 utilizando un bucle `while`.

4. Ejercicio con `For`: Escribe un programa que imprima la tabla de multiplicar del 5 (del 5 x 1 al 5 x 10) utilizando un bucle `for`.

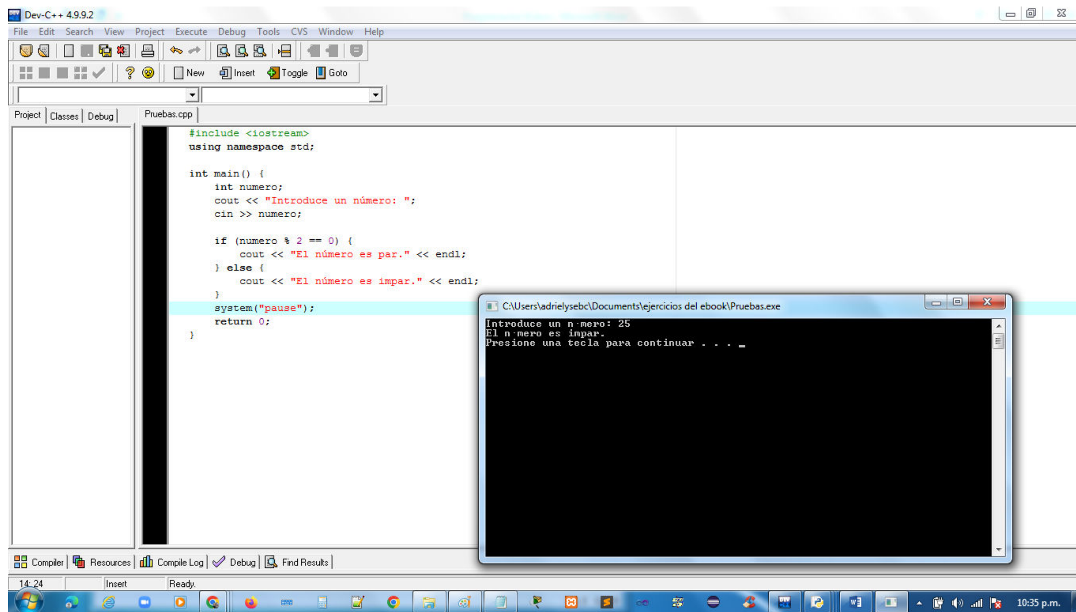
5. Ejercicio con `Do While`: Escribe un programa que pida al usuario que ingrese un número positivo y siga pidiendo hasta que se ingrese un número válido.

Soluciones a los Ejercicios Propuestos

1. Solución de Par o Impar:

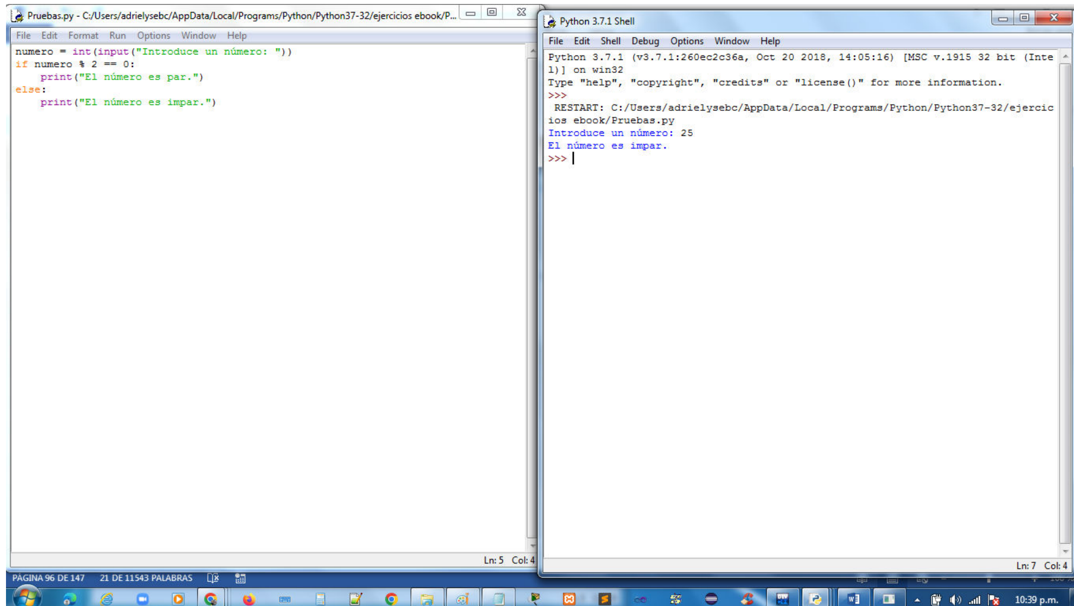
C++:

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

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

    if (numero % 2 == 0) {
        cout << "El número es par." << endl;
    } else {
        cout << "El número es impar." << endl;
    }
    return 0;
}
```



Python:

```python
numero = int(input("Introduce un número: "))
if numero % 2 == 0:
    print("El número es par.")
else:
    print("El número es impar.")
```

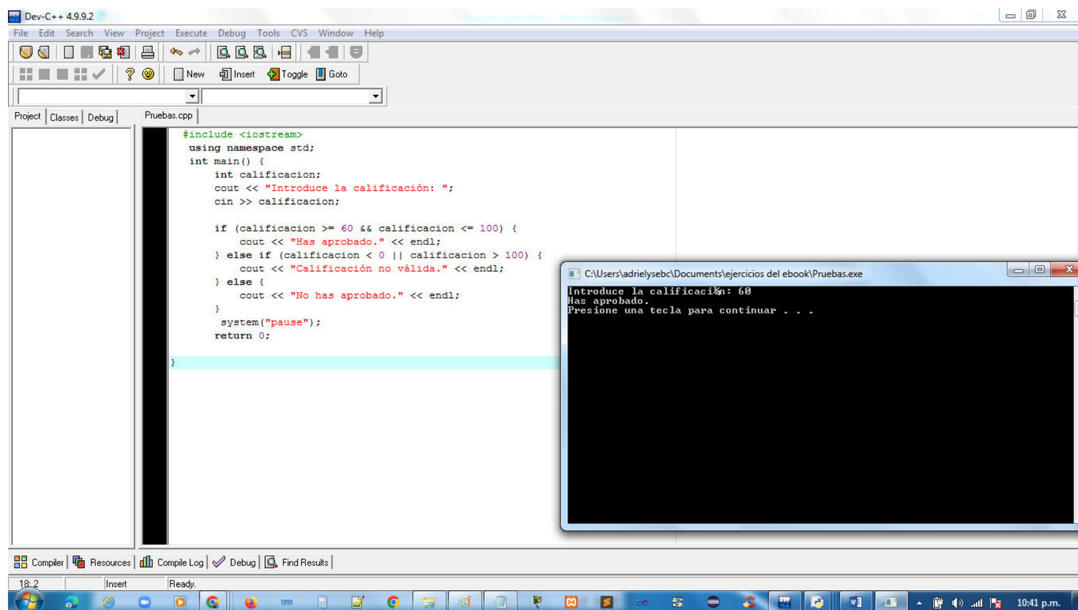


2. Solución de Calificación:

C++:

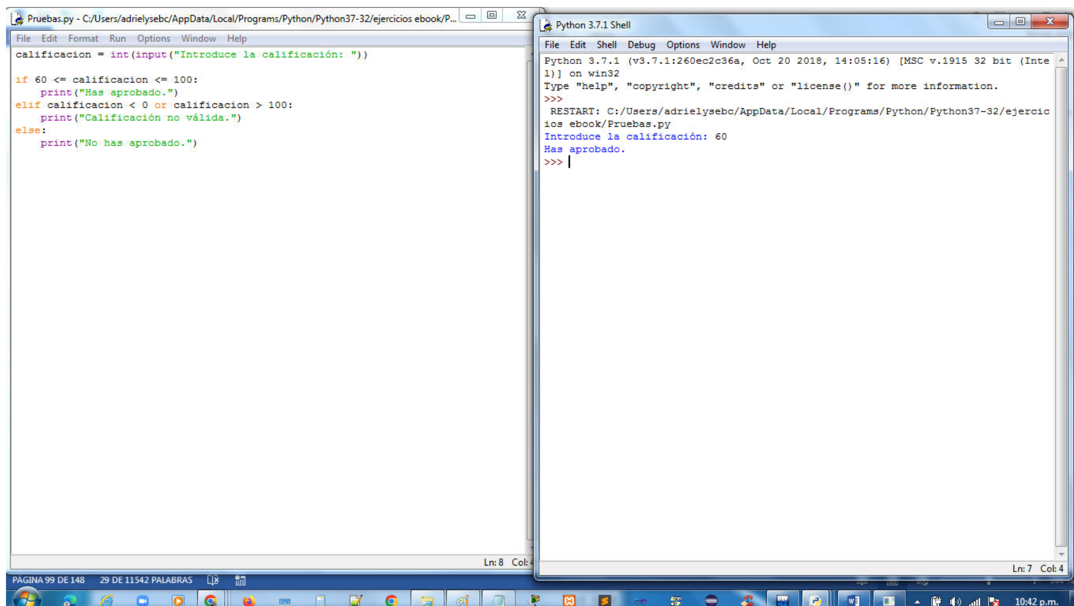
```cpp


}#include <iostream>

using namespace std;

int main() {

    int calificacion;

    cout << "Introduce la calificación: ";

    cin >> calificacion;


    if (calificacion >= 60 && calificacion <= 100) {
```

```
    cout << "Has aprobado." << endl;
  } else if (calificacion < 0 || calificacion > 100) {
    cout << "Calificación no válida." << endl;
  } else {
    cout << "No has aprobado." << endl;
  }
  return 0;
```



Python:

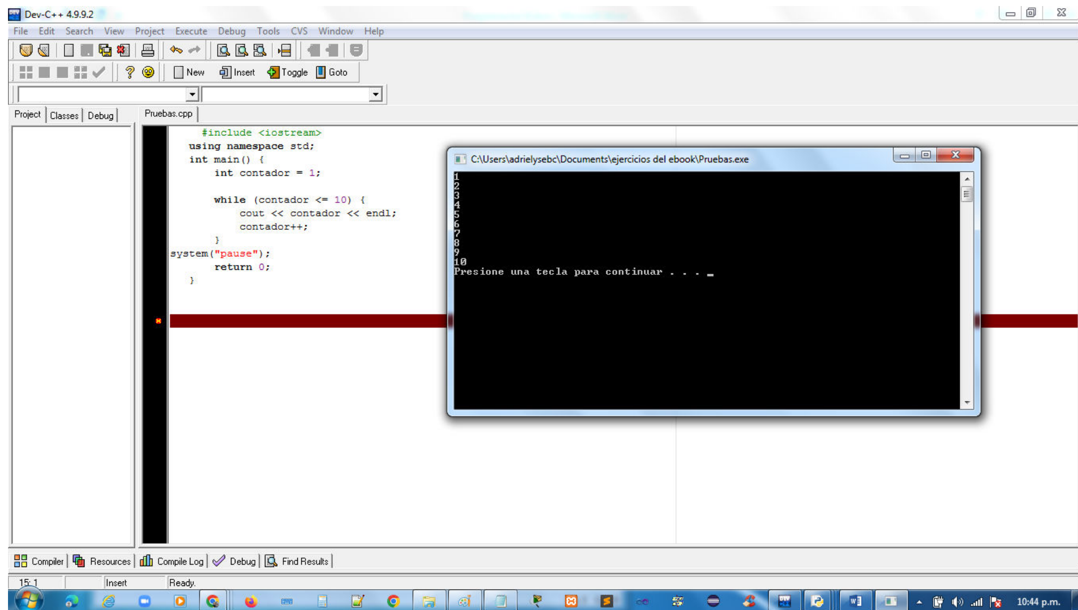
```python
calificacion = int(input("Introduce la calificación: "))


if 60 <= calificacion <= 100:
    print("Has aprobado.")
elif calificacion < 0 or calificacion > 100:
    print("Calificación no válida.")
else:
    print("No has aprobado.")
```



3. Solución con `While`:

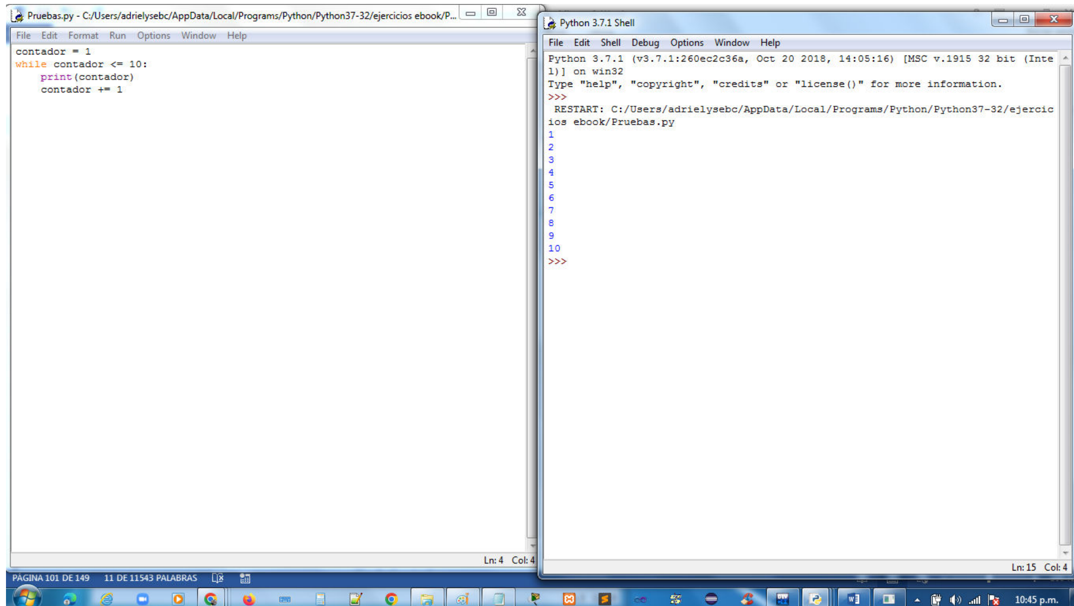
C++:

```cpp
#include <iostream>
using namespace std;
int main() {
    int contador = 1;

    while (contador <= 10) {
        cout << contador << endl;
        contador++;
    }
system(“pause”);
    return 0;
}
```



Python:

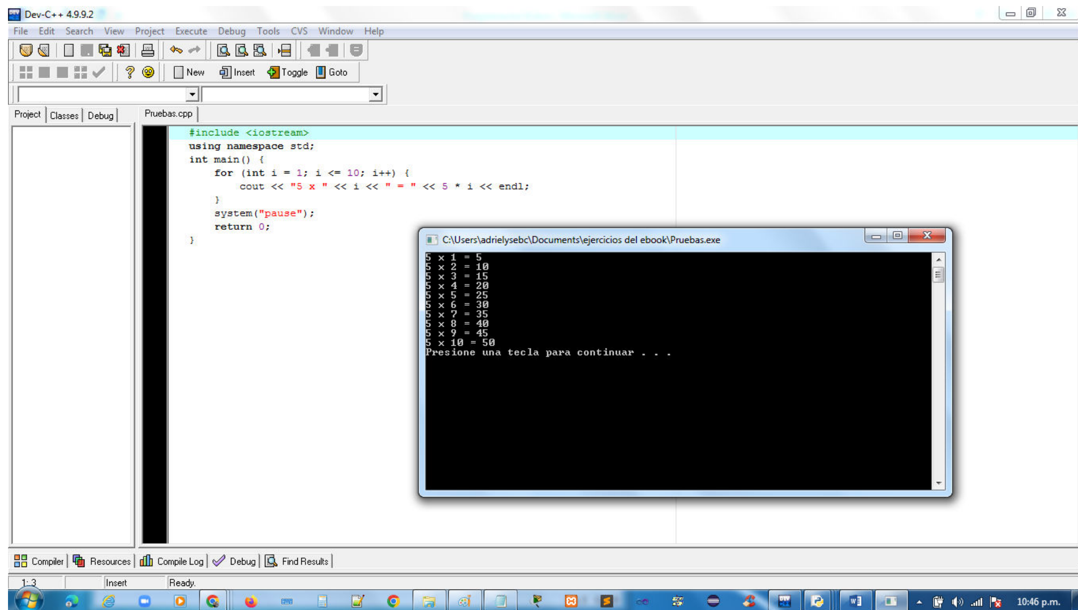
```python
contador = 1

while contador <= 10:
    print(contador)
    contador += 1
```



4. Solución con `For`:

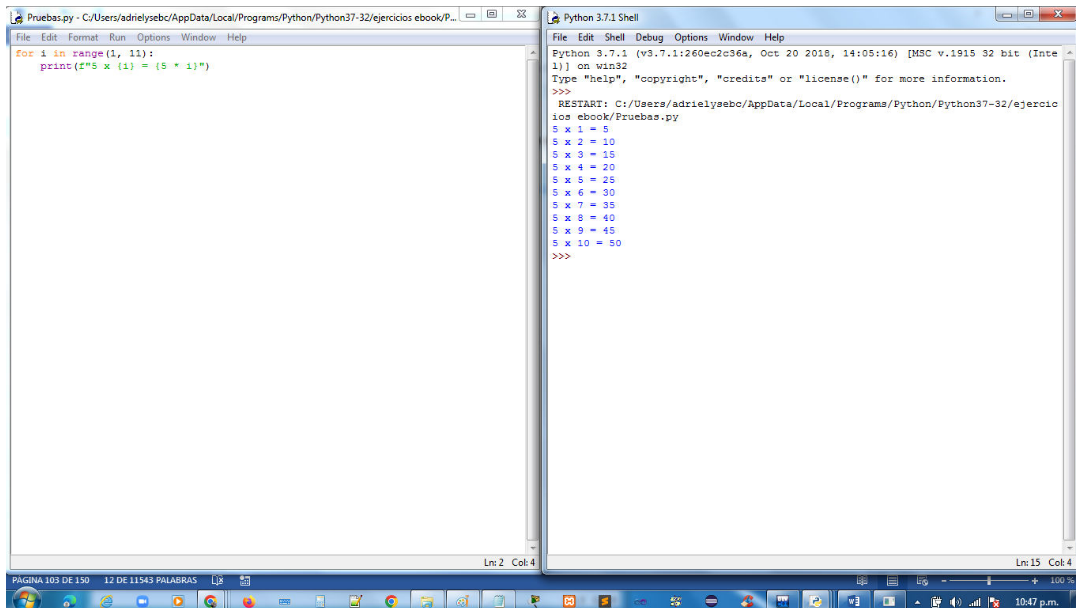
C++:

```cpp
#include <iostream>

using namespace std;

int main() {

    for (int i = 1; i <= 10; i++) {

        cout << "5 x " << i << " = " << 5 * i << endl;

    }

    return 0;

}
```



Python:

```python
for i in range(1, 11):
    print(f"5 x {i} = {5 * i}")
```



5. Solución con `Do While`:

C++:

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

int main() {
    int numero;

    do {
        cout << "Introduce un número positivo: ";
        cin >> numero;
```

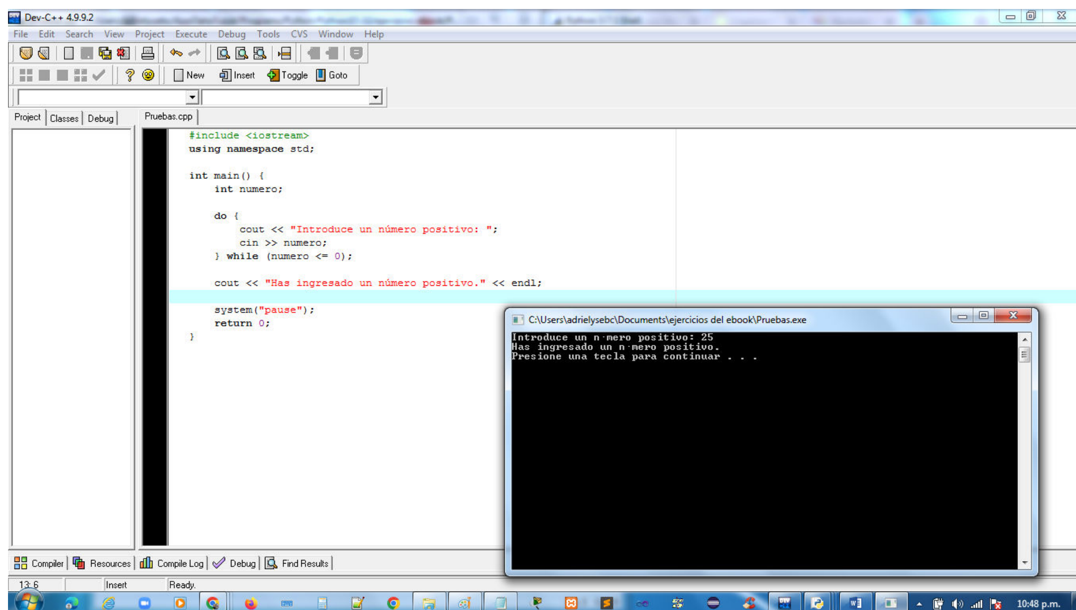
} while (numero <= 0);

cout << "Has ingresado un número positivo." << endl;

return 0;

}

```




**Python:**

```python

while True:

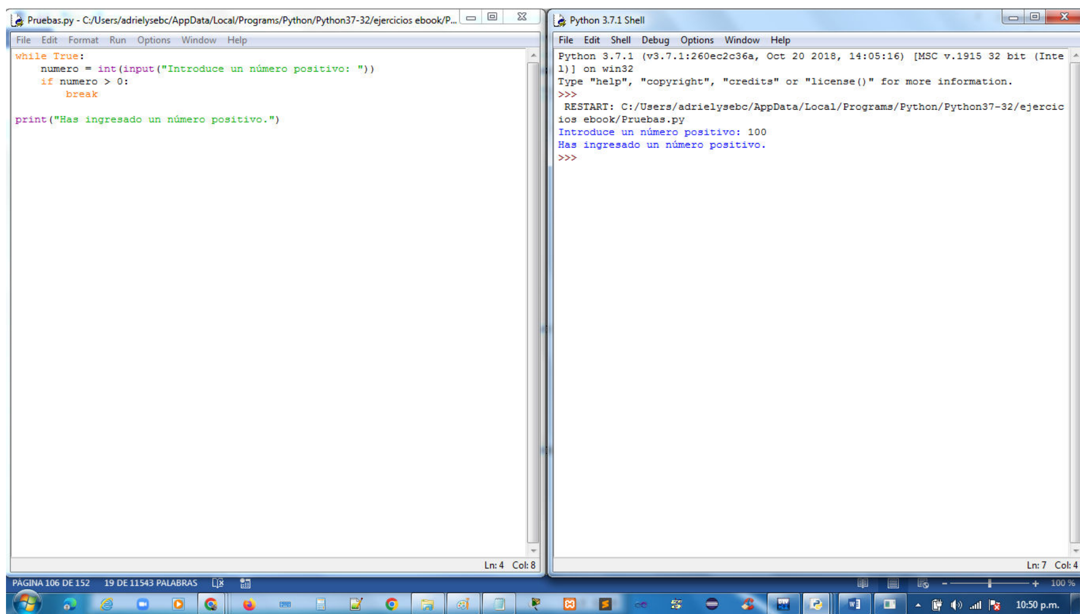
numero = int(input("Introduce un número positivo: "))

if numero > 0:

break

*print("Has ingresado un número positivo.")*

```


```

UNIDAD IV

ARREGLOS Y PUNTEROS EN C++ Y PYTHON

En esta unidad se proporciona un marco teórico y práctico sobre arreglos y punteros en C++ y Python, junto con ejemplos y ejercicios que permiten a los estudiantes aplicar lo aprendido

Arreglos Bidimensionales

Los arreglos bidimensionales son estructuras de datos que permiten almacenar datos en una tabla, es decir, en filas y columnas.

Ejemplo en C++*

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


int main() {
    int matriz[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };


    // Imprimir la matriz
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matriz[i][j] << " ";
        }
```

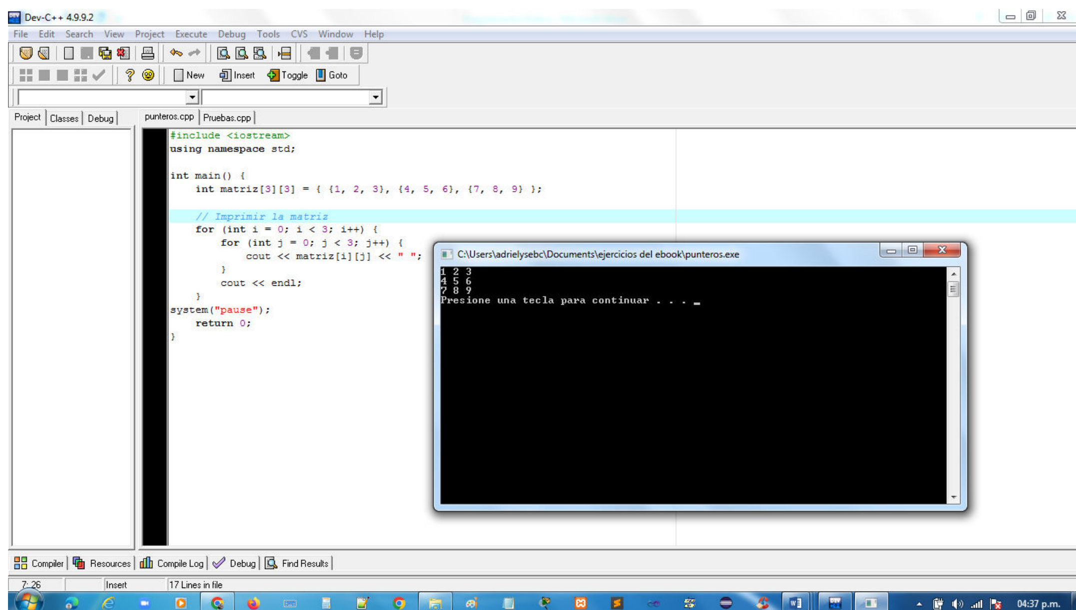
cout << endl;

}

return 0;

}

```
```



Ejemplo en Python:

```python

*matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]*

*# Imprimir la matriz*

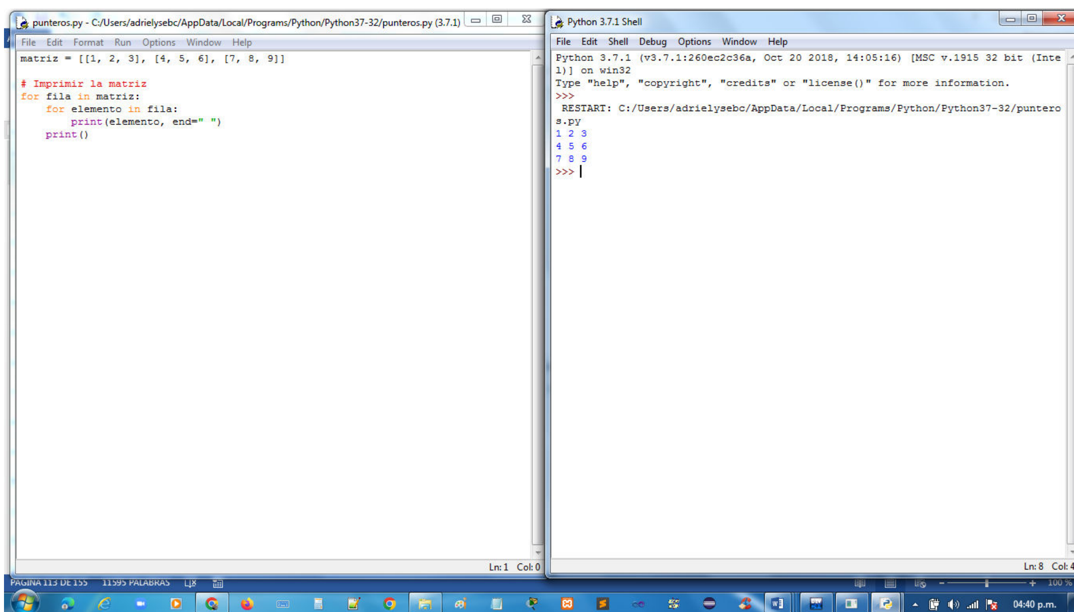
*for fila in matriz:*
```

for elemento in fila:

print(elemento, end=" ")

print()

```



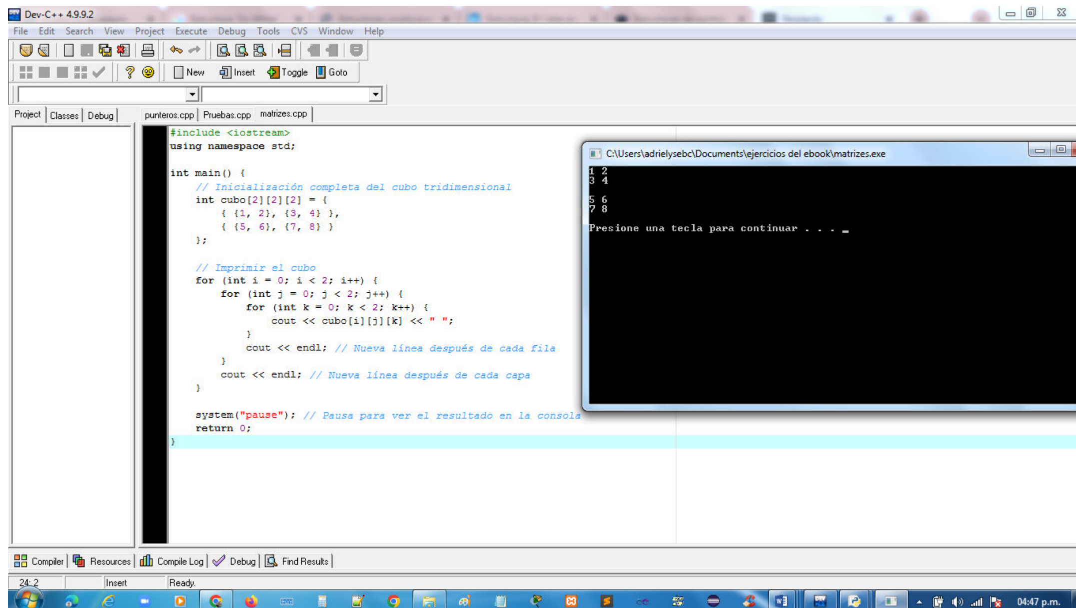

**Arreglos Multidimensionales**

Los arreglos multidimensionales son extensiones de los arreglos bidimensionales y pueden tener más de dos dimensiones.

**Ejemplo en C++:**

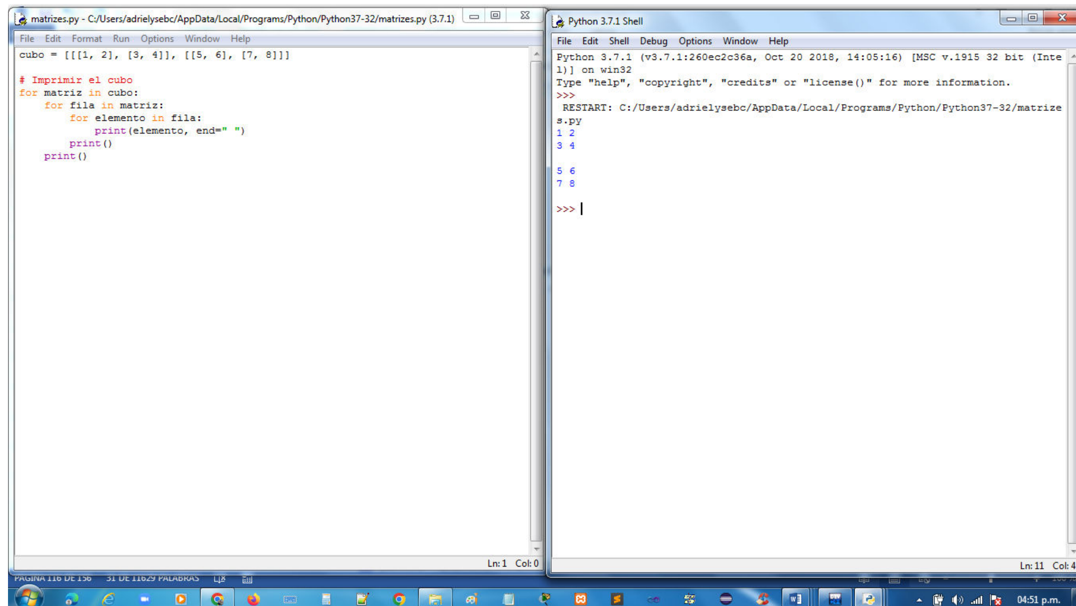
*```cpp*

*#include <iostream>*

*using namespace std;*
```

```
int main() {
    // Inicialización completa del cubo tridimensional
    int cubo[2][2][2] = {
        { {1, 2}, {3, 4} },
        { {5, 6}, {7, 8} }
    };
    // Imprimir el cubo
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                cout << cubo[i][j][k] << " ";
            }
            cout << endl; // Nueva línea después de cada fila
        }
        cout << endl; // Nueva línea después de cada capa
    }
    system("pause"); // Pausa para ver el resultado en la consola
    return 0;
}
```



Ejemplo en Python:

```python
cubo = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]

# Imprimir el cubo

for matriz in cubo:

    for fila in matriz:

        for elemento in fila:

            print(elemento, end=" ")

        print()

    print()
```



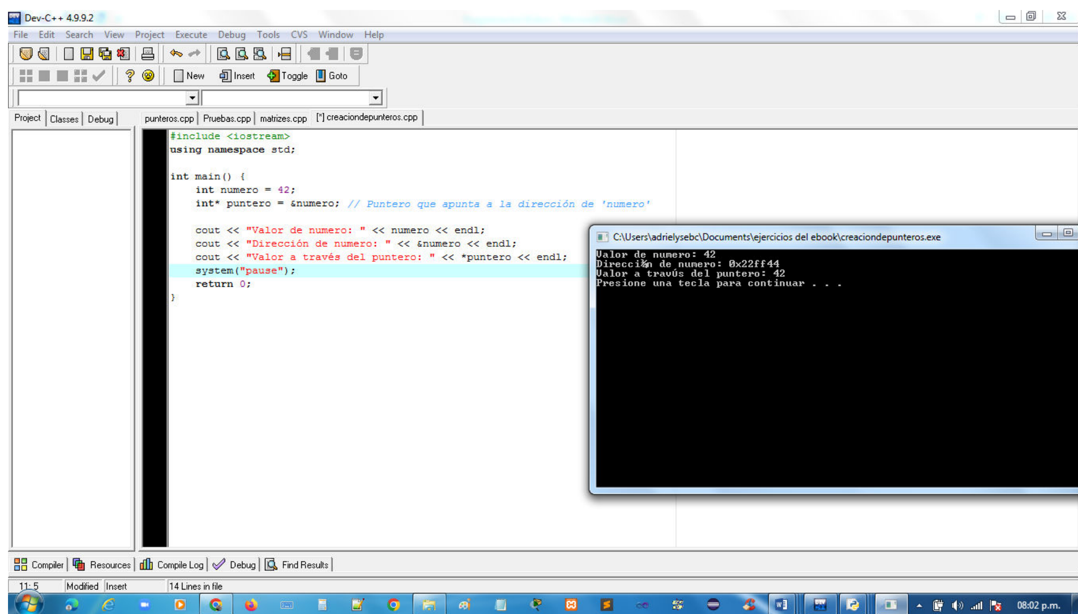
Creación de Punteros

Los punteros son variables que almacenan la dirección de otra variable. En C++, se utilizan con el operador `*` para declarar un puntero.

Ejemplo en C++:

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

int main() {
    int numero = 42;
    int* puntero = &numero; // Puntero que apunta a la dirección de 'numero'
```

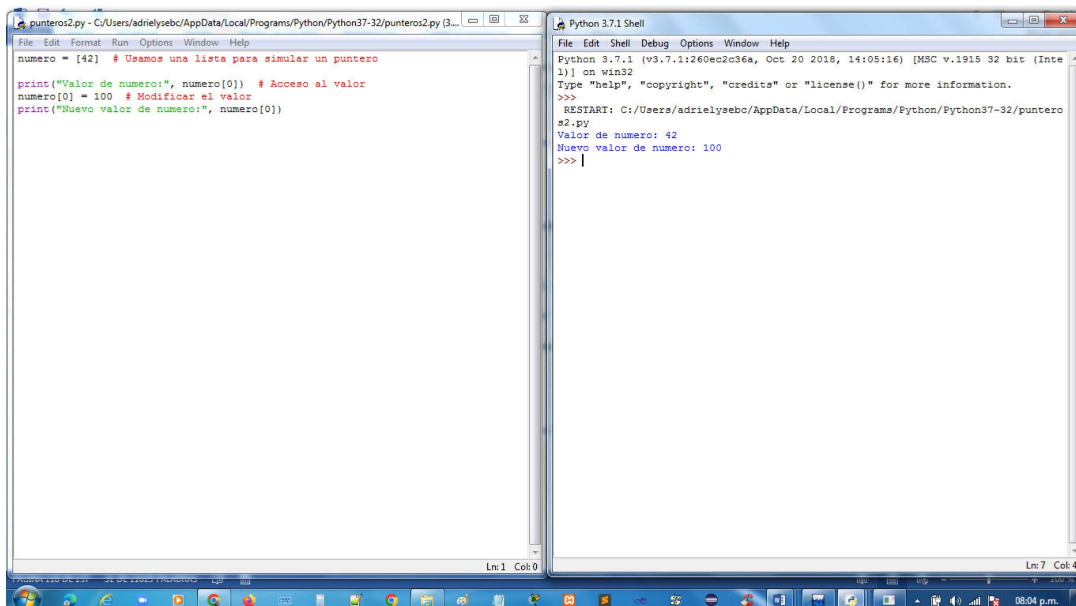
```
    cout << "Valor de numero: " << numero << endl;

    cout << "Dirección de numero: " << &numero << endl;

    cout << "Valor a través del puntero: " << *puntero << endl;


    return 0;

}
```



Ejemplo en Python:

Python no tiene punteros como C++, pero se pueden simular utilizando listas o diccionarios.

```python
numero = [42]  # Usamos una lista para simular un puntero


print("Valor de numero:", numero[0])  # Acceso al valor
numero[0] = 100  # Modificar el valor
print("Nuevo valor de numero:", numero[0])
```

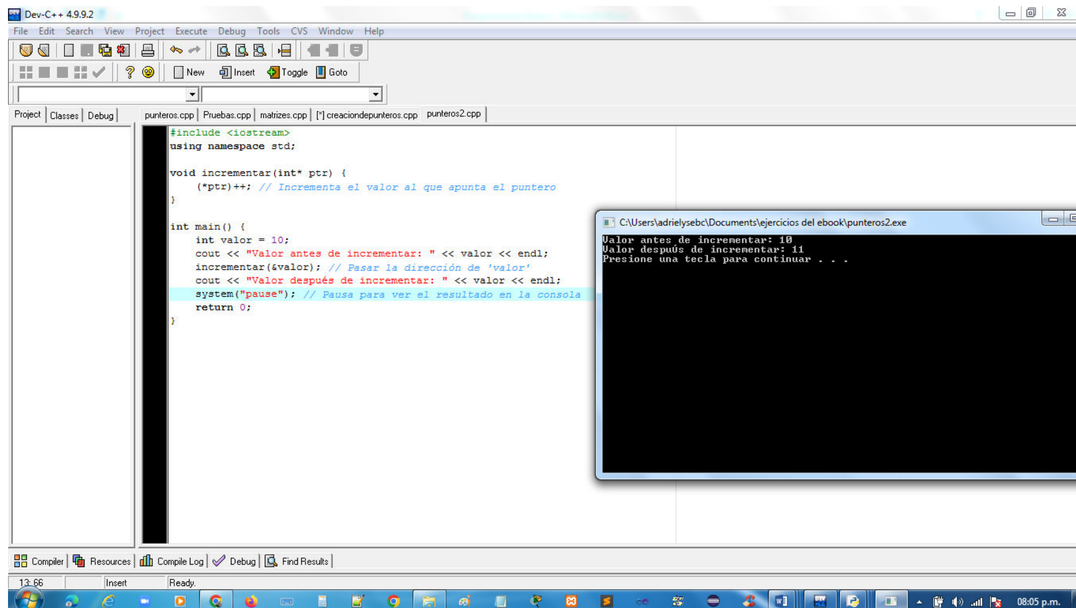


Administración de Punteros

La administración de punteros implica el uso de punteros para acceder y manipular datos en memoria.

Ejemplo en C++:

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

void incrementar(int* ptr) {
    (*ptr)++; // Incrementa el valor al que apunta el puntero
}
int main() {
    int valor = 10;
    cout << "Valor antes de incrementar: " << valor << endl;
    incrementar(&valor); // Pasar la dirección de 'valor'
    cout << "Valor después de incrementar: " << valor << endl;
    return 0;
}
```



Ejemplo en Python:

Aunque Python no tiene punteros, se puede lograr un efecto similar utilizando listas.

```python
def incrementar(lista):
    lista[0] += 1  # Incrementa el primer elemento de la lista

valor = [10]
print("Valor antes de incrementar:", valor[0])
incrementar(valor)  # Pasar la lista
print("Valor después de incrementar:", valor[0])
```

```

punteros3.py - C:/Users/adrielysebc/AppData/Local/Programs/Python/Python37-32/punteros3.py (3....

File Edit Format Run Options Window Help

~~~python
def incrementar(lista):
    lista[0] += 1  # Incrementa el primer elemento de la lista

valor = [10]
print("Valor antes de incrementar:", valor[0])
incrementar(valor)  # Pasar la lista
print("Valor después de incrementar:", valor[0])
~~~

Ln: 3 Col: 0

Python 3.7.1 Shell

File Edit Shell Debug Options Window Help

~~~
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Inte
l)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/adrielysebc/AppData/Local/Programs/Python/Python37-32/puntero
s3.py
Valor antes de incrementar: 10
Valor después de incrementar: 11
>>> 
~~~

Ln: 7 Col: 4

PÁGINA 121 DE 158  31 DE 11629 PALABRAS

08:07 p.m.
```

Ejercicios Propuestos

1. Ejercicio de Arreglos Bidimensionales: Crea un programa que inicialice una matriz de 3x3 con números del 1 al 9 y los imprima.

2. Ejercicio de Arreglos Multidimensionales: Crea un programa que inicialice un cubo de 2x2x2 con números del 1 al 8 y los imprima.

3. Ejercicio de Punteros: Escribe un programa que declare un puntero a un entero, le asigne un valor y lo imprima.

4. Ejercicio de Administración de Punteros: Crea una función que reciba un puntero y lo incremente en 5. Imprime el valor antes y después de la llamada a la función.

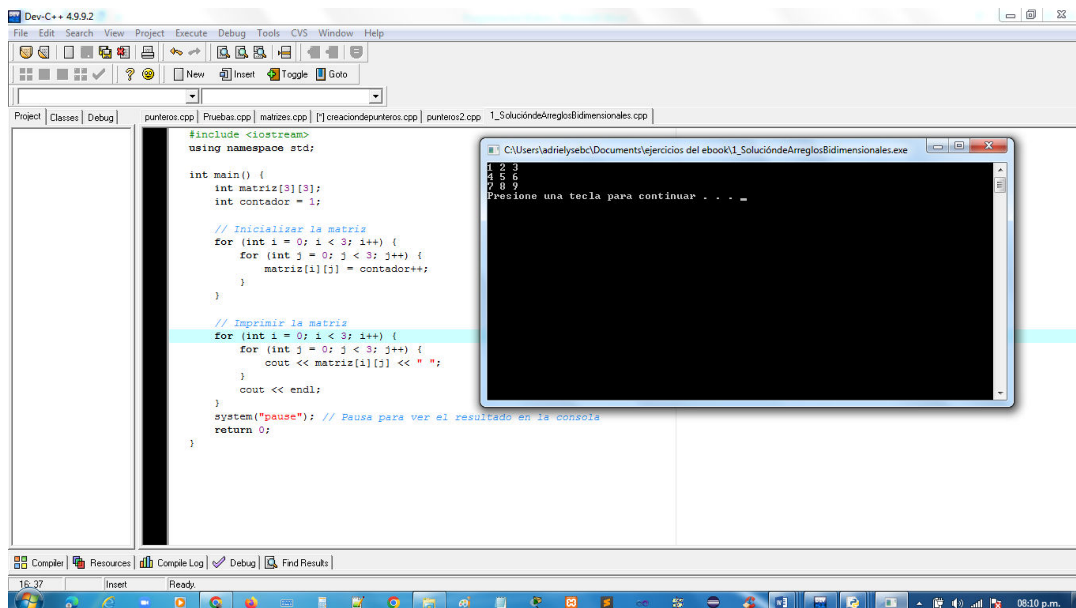
Soluciones a los Ejercicios Propuestos

1. Solución de Arreglos Bidimensionales:

C++:

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

int main() {
    int matriz[3][3];
    int contador = 1;

    // Inicializar la matriz
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            matriz[i][j] = contador++;
        }
    }

    // Imprimir la matriz
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
```

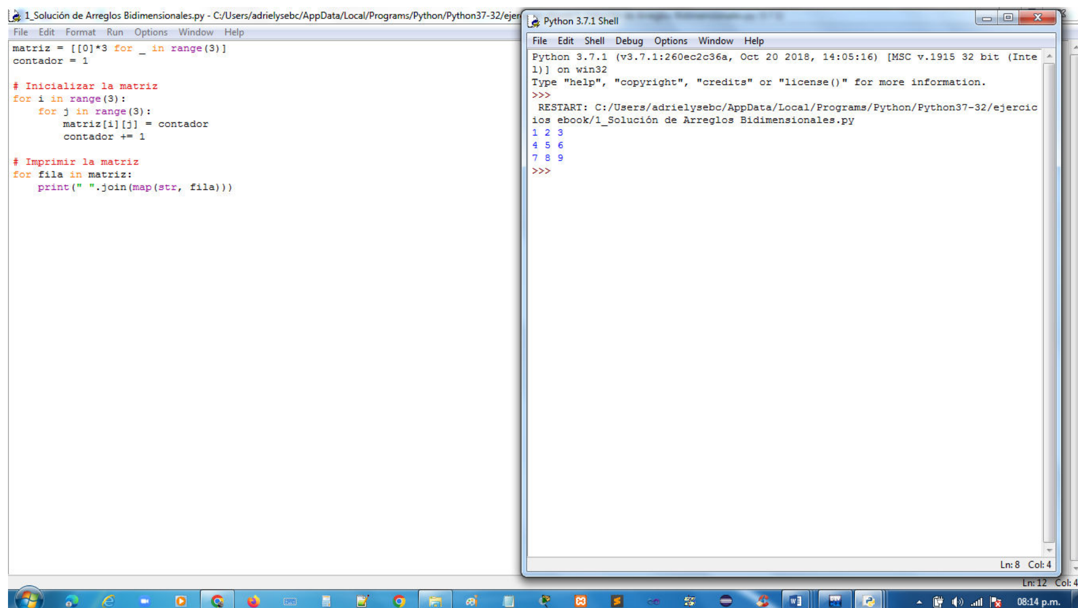
```
            cout << matriz[i][j] << " ";
        }
        cout << endl;
    }
    system("pause"); // Pausa para ver el resultado en la consola
    return 0;
}
```



Python:

```python
matriz = [[0]*3 for _ in range(3)]

contador = 1
```

```
# Inicializar la matriz
for i in range(3):
    for j in range(3):
        matriz[i][j] = contador
        contador += 1

# Imprimir la matriz
for fila in matriz:
    print(" ".join(map(str, fila)))
```

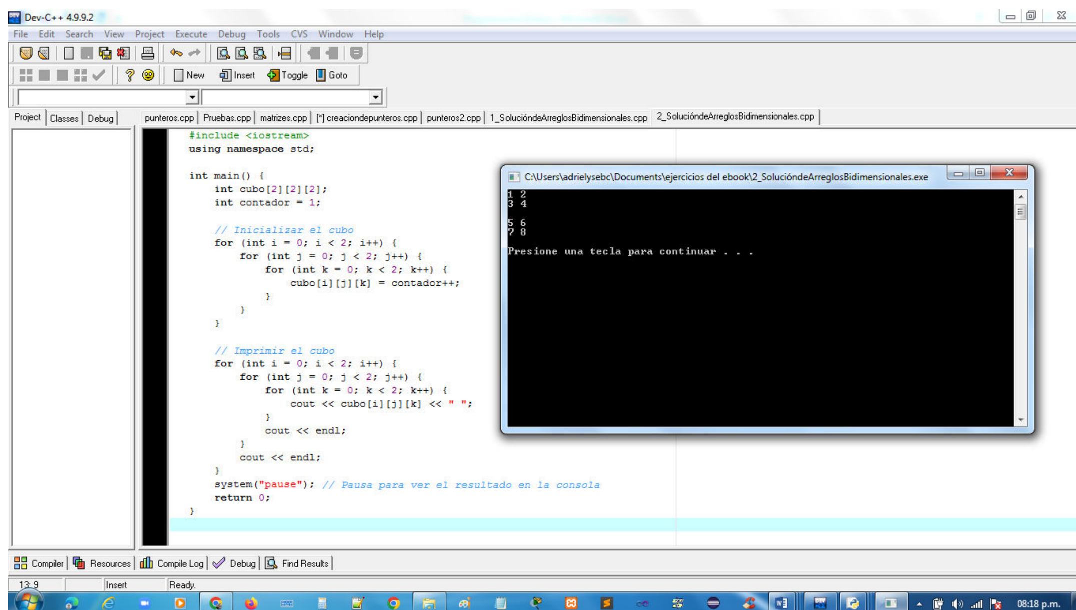


2. Solución de Arreglos Multidimensionales:

C++:

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

int main() {
    int cubo[2][2][2];
    int contador = 1;

    // Inicializar el cubo
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                cubo[i][j][k] = contador++;
            }
        }
    }

    // Imprimir el cubo
    for (int i = 0; i < 2; i++) {
```

```
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                cout << cubo[i][j][k] << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    return 0;
}
```



Python:

```python
cubo = [[[0]*2 for _ in range(2)] for _ in range(2)]
contador = 1

# Inicializar el cubo
for i in range(2):
    for j in range(2):
        for k in range(2):
            cubo[i][j][k] = contador
            contador += 1

# Imprimir el cubo
for matriz in cubo:
    for fila in matriz:
        print(" ".join(map(str, fila)))
    print()
```

```
2_Solución de Arreglos Bidimensionales.py - C:/Users/adrielysebc/AppData/Local/Programs/Python/Python37-32/ejercicios ebook/2_Solución de Arreglos Bidimensionales.py (3.7.1)
File  Edit  Format  Run  Options  Window  Help
cubo = [[[0]*2 for _ in range(2)] for _ in range(2)]
contador = 1

# Inicializar el cubo
for i in range(2):
    for j in range(2):
        for k in range(2):
            cubo[i][j][k] = contador
            contador += 1

# Imprimir el cubo
for matriz in cubo:
    for fila in matriz:
        print(" ".join(map(str, fila)))
    print()
```

```
Python 3.7.1 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Inte
l)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/adrielysebc/AppData/Local/Programs/Python/Python37-32/ejercic
ios ebook/2_Solución de Arreglos Bidimensionales.py
1 2

3 4

5 6

7 8

>>> |
```

3. Solución de Punteros:

C++:

```cpp

#include <iostream>

using namespace std;

int main() {

int numero = 42;

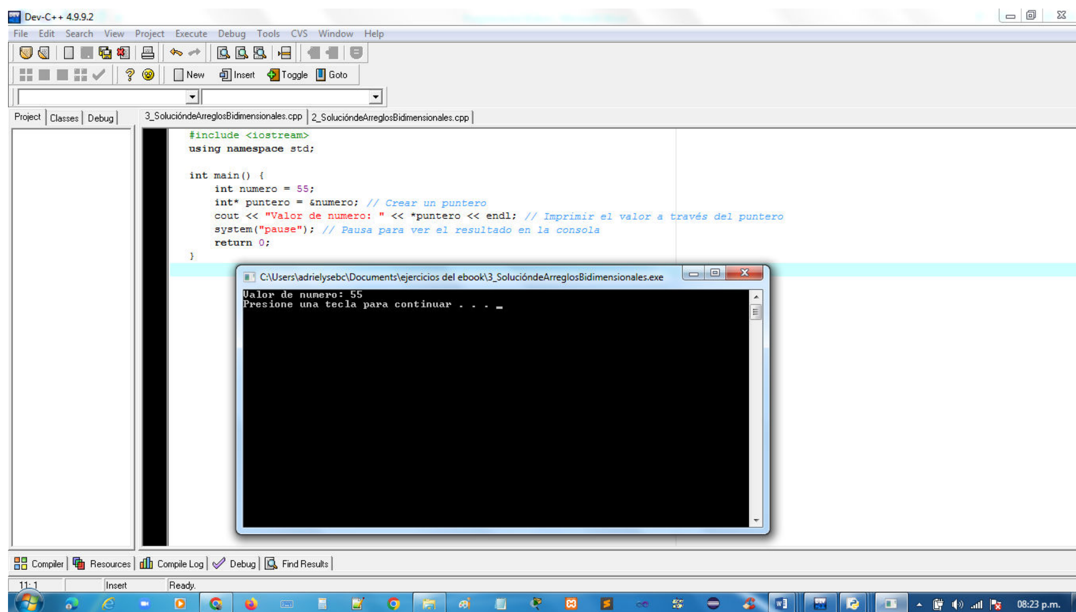
int puntero = № // Crear un puntero*

*cout << "Valor de numero: " << *puntero << endl; // Imprimir el valor a través del puntero*

return 0;

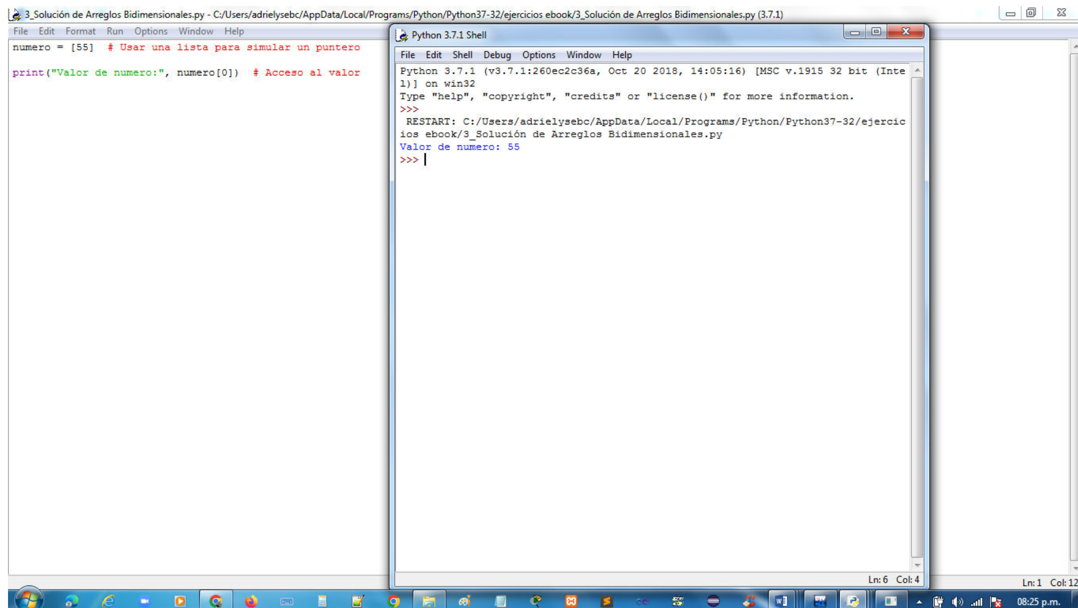
}

```
```



Python:

```python
numero = [55]  # Usar una lista para simular un puntero


print("Valor de numero:", numero[0])  # Acceso al valor
```

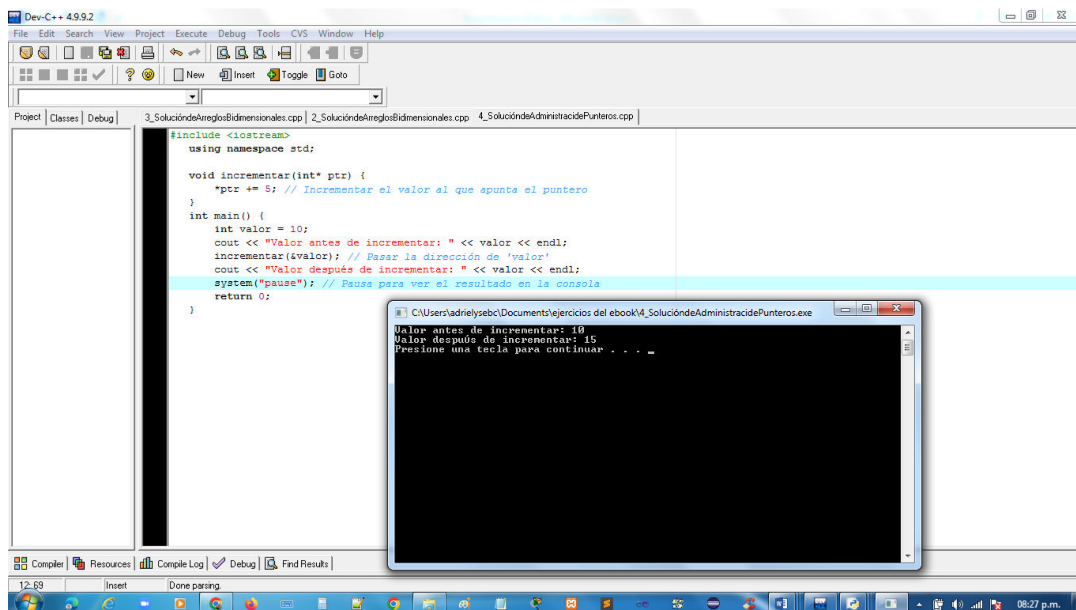


4. Solución de Administración de Punteros:

C++:

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


void incrementar(int* ptr) {
    *ptr += 5; // Incrementar el valor al que apunta el puntero
}
int main() {
    int valor = 10;
    cout << "Valor antes de incrementar: " << valor << endl;
```

```
    incrementar(&valor); // Pasar la dirección de 'valor'

    cout << "Valor después de incrementar: " << valor << endl;

    return 0;

}
```



Python:

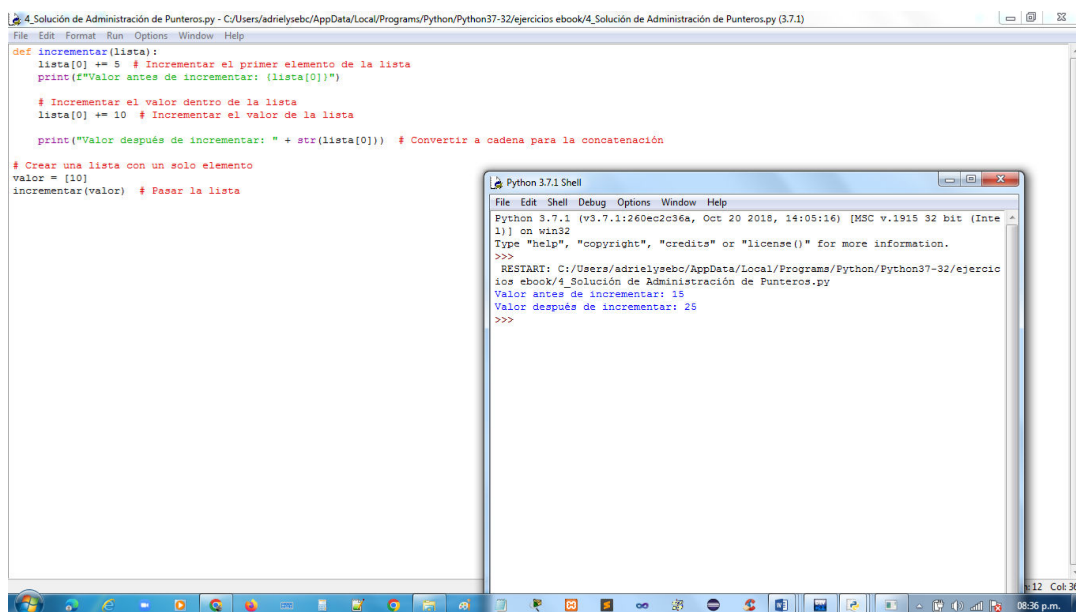
```python

def incrementar(lista):

 lista[0] += 5  # Incrementar el primer elemento de la lista

 print(f"Valor antes de incrementar: {lista[0]}")

     # Incrementar el valor dentro de la lista

 lista[0] += 10  # Incrementar el valor de la lista
```

print("Valor después de incrementar: " + str(lista[0])) # Convertir a cadena para la concatenación

 # Crear una lista con un solo elemento

valor = [10]

incrementar(valor) # Pasar la lista

```




*Se usa `**str(lista)**` para convertir el entero a cadena antes de concatenarlo con otra cadena.*
```

Técnicas De Administración De Memoria

Las técnicas de administración de memoria que se utilizan con punteros en C++ son fundamentales para el manejo eficiente de recursos.

A continuación, se describen las principales técnicas:

Asignación Dinámica de Memoria

La asignación dinámica permite reservar memoria en tiempo de ejecución, lo que es útil para estructuras de datos cuyo tamaño no se conoce de antemano. En C++, se utilizan los operadores `new` y `delete`.

Ejemplo de Asignación Dinámica:

```cpp
int* puntero = new int; // Asignación de memoria para un entero
*puntero = 42; // Asignar valor
delete puntero; // Liberar memoria
```

Arreglos Dinámicos:

```cpp
int* arreglo = new int[10]; // Asignar memoria para un arreglo de 10 enteros
delete[] arreglo; // Liberar memoria del arreglo
```

Gestión de Recursos

Es importante liberar la memoria que ya no se necesita para evitar fugas de memoria. Se utilizan `delete` y `delete[]` para liberar memoria asignada con `new` y `new[]`, respectivamente.

Ejemplo:

```cpp
int* puntero = new int[5]; // Asignación de un arreglo dinámico
// Uso del puntero
delete[] puntero; // Liberar memoria
```

Punteros a Punteros

Los punteros pueden apuntar a otros punteros, lo que permite crear estructuras de datos más complejas, como matrices dinámicas.

Ejemplo:

```cpp
int** punteroDoble = new int*[3]; // Crear un puntero a puntero para una matriz 3x3
for (int i = 0; i < 3; i++) {
    punteroDoble[i] = new int[3]; // Asignar memoria para cada fila
}
// Liberar memoria
for (int i = 0; i < 3; i++) {
```

```cpp
    delete[] punteroDoble[i];
}
delete[] punteroDoble;
```

Uso de Smart Pointers (C++11 en adelante)

Los smart pointers son una forma moderna de manejar la memoria en C++. Estos punteros gestionan automáticamente la memoria, liberándola cuando ya no son necesarios.

Ejemplo con `std::unique_ptr`:

```cpp
#include <memory>


int main() {
    std::unique_ptr<int> puntero(new int(42)); // Asignación automática
    // No es necesario liberar la memoria, se hace automáticamente
    return 0;
}
```

Evitar Fugas de Memoria

Es crucial seguir buenas prácticas para evitar fugas de memoria. Esto incluye:

- Siempre liberar la memoria que se ha asignado dinámicamente.

- Usar herramientas de análisis de memoria (como Valgrind) para detectar fugas.
- Preferir el uso de contenedores estándar de C++ (como `std::vector`) que manejan la memoria automáticamente.

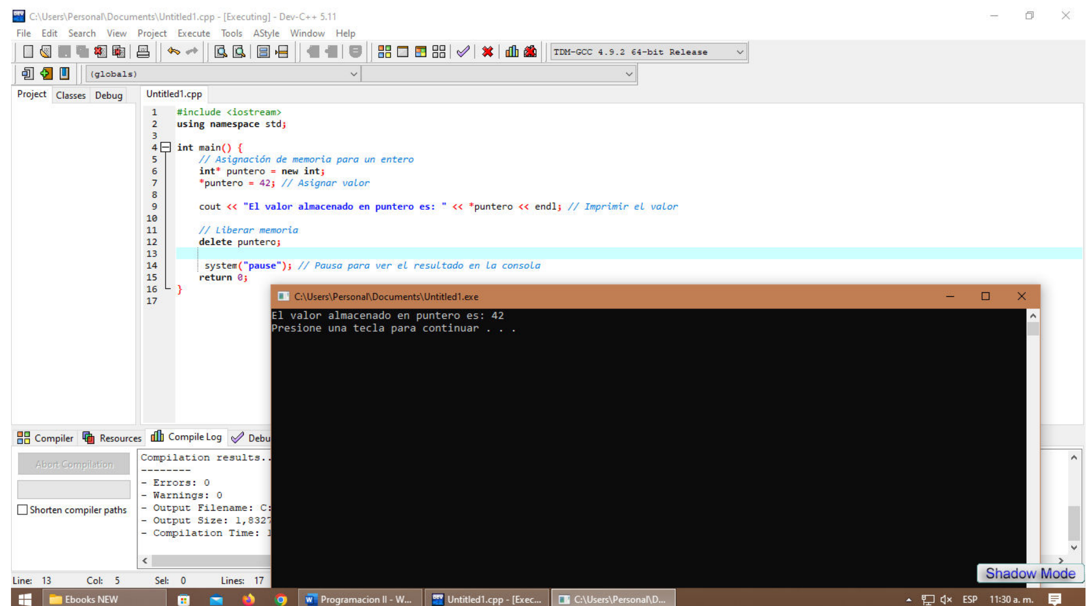
La administración de memoria con punteros en C++ es una habilidad esencial para el desarrollo de software eficiente. Las técnicas de asignación dinámica, gestión de recursos, y el uso de punteros inteligentes son fundamentales para evitar fugas de memoria y asegurar un uso eficiente de los recursos del sistema.

Te mostraremos un código que proporcionara la asignación y liberación de memoria dinámica en C++ es de manera correcta. Sin embargo, es importante asegurarse de que el uso de punteros y la gestión de memoria se realicen de manera segura y eficiente. Aquí mostramos un ejemplo corregido y mejorado, junto con algunas recomendaciones:

Código de ejemplo

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

int main() {
    // Asignación de memoria para un entero
    int* puntero = new int;
    *puntero = 42; // Asignar valor
```

```
        cout << "El valor almacenado en puntero es: " << *puntero << endl; //
Imprimir el valor

        // Liberar memoria
        delete puntero;
        puntero = nullptr; // Opcional: evitar punteros colgantes

        return 0;
    }
```



Explicación del código

1. `#include <iostream>`: Se añade para la inclusión de la biblioteca `<iostream>` para permitir el uso de `cout`.

2. Uso de `cout`: Se agregó una línea para imprimir el valor almacenado en el puntero, lo que proporciona retroalimentación sobre la operación.

3. Inicialización del Puntero a `nullptr` Después de liberar la memoria con `delete`, se asigna `nullptr` al puntero. Esto es una buena práctica para evitar punteros colgantes, que pueden causar errores si se intenta acceder a la memoria después de haberla liberado.

Consideraciones Adicionales

Manejo de Errores: En un programa más complejo, considera agregar manejo de errores para verificar si la asignación de memoria fue exitosa.

Uso de `std::unique_ptr` o `std::shared_ptr`: En C++, es recomendable usar punteros inteligentes como `std::unique_ptr` o `std::shared_ptr` para gestionar la memoria automáticamente y evitar fugas de memoria. Esto simplifica el manejo de la memoria y hace que el código sea más seguro.

Ejemplo con `std::unique_ptr`

Aquí tienes un ejemplo utilizando `std::unique_ptr`:

```cpp
#include <iostream>
#include <memory> // Para std::unique_ptr
using namespace std;
```

```
int main() {
    // Usar un puntero inteligente para gestionar la memoria
    unique_ptr<int> puntero = make_unique<int>(42); // Asignar y inicializar
    cout << "El valor almacenado en puntero es: " << *puntero << endl; // Imprimir el valor

    // No es necesario llamar a delete, se libera automáticamente al salir del alcance
    return 0;
}
```

El uso de punteros y la gestión de memoria son fundamentales en C++. Asegúrate de seguir buenas prácticas para evitar fugas de memoria y errores en tu programa.

UNIDAD V

DESARROLLO DE PROGRAMAS CON FUNCIONES EN C++ Y PYTHON

Esta unidad proporciona un marco teórico y práctico sobre el desarrollo de programas con funciones en C++ y Python, junto con ejemplos y ejercicios que permiten a los estudiantes aplicar lo aprendido.

Concepto, Elementos y Uso de Funciones

Las funciones son bloques de código reutilizables que realizan tareas específicas. Tienen elementos como:

- Nombre de la función

- Parámetros (opcionales)

- Cuerpo de la función (instrucciones)

- Valor de retorno (opcional)

Pasos para Declarar y Definir Funciones en C++

Declaración de la Función:

- Se especifica el tipo de retorno de la función.
- Se proporciona el nombre de la función.
- Se incluyen los parámetros entre paréntesis (si los hay).
- Se finaliza con un punto y coma (`;`) si es solo una declaración.

Ejemplo:

```cpp
int sumar(int a, int b); // Declaración de la función
```

```
1    int sumar(int a, int b); // Declaración de la función
```

Definición de la Función:

- Se repite la declaración con el cuerpo de la función, que contiene las instrucciones que se ejecutarán.
- Se utiliza la palabra clave `return` para devolver un valor (si es necesario).

Ejemplo:

```cpp
int sumar(int a, int b) {
    return a + b; // Cuerpo de la función
}
```

```cpp
int sumar(int a, int b) {
    return a + b; // Cuerpo de la función
}
```

Llamada a la Función:

- Para utilizar la función, se llama por su nombre y se pasan los argumentos necesarios entre paréntesis.

Ejemplo:

```cpp

int resultado = sumar(10, 20); // Llamada a la función

```

```cpp
int resultado = sumar(10, 20); // Llamada a la función
```

Uso de la Función `main`:

- La función `main` es el punto de entrada del programa y se utiliza para llamar a otras funciones.

Ejemplo Completo:

```cpp

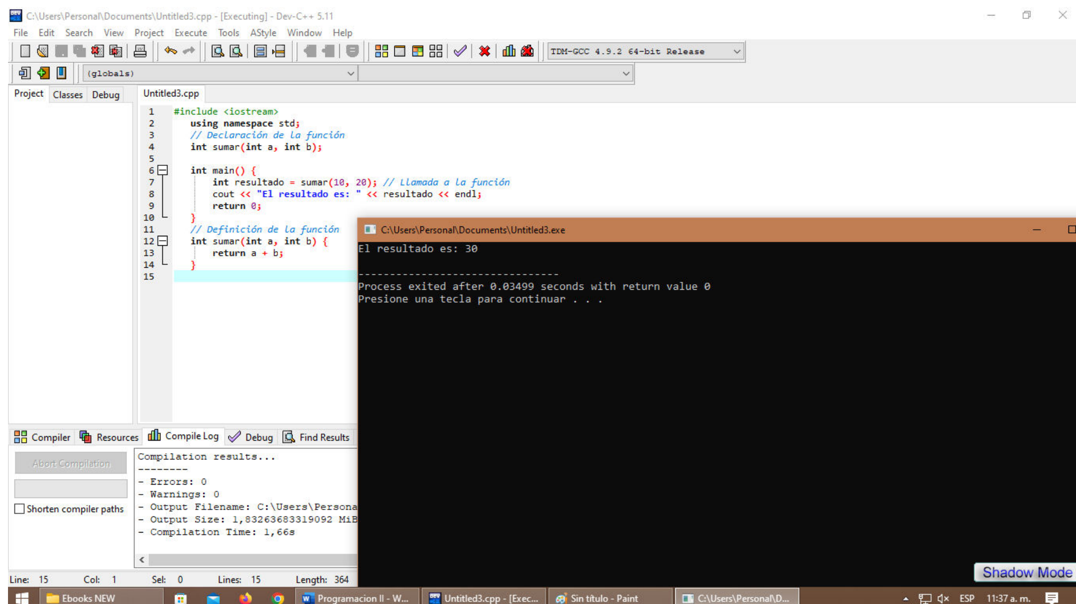
#include <iostream>

using namespace std;

// Declaración de la función

int sumar(int a, int b);

```
int main() {
    int resultado = sumar(10, 20); // Llamada a la función
    cout << "El resultado es: " << resultado << endl;
    return 0;
}
// Definición de la función
int sumar(int a, int b) {
    return a + b;
}
```



Resumen de los Pasos

1. Declarar la función especificando el tipo de retorno, nombre y parámetros.
2. Definir la función con su cuerpo y lógica.
3. Llamar a la función desde `main` o desde otra función, pasando los argumentos necesarios.

Estos pasos permiten estructurar y organizar el código en C++, facilitando la reutilización y la claridad del mismo.

Para declarar una función en C++ con parámetros, se siguen estos pasos

1. Especificar el Tipo de Retorno: Indica el tipo de dato que la función devolverá. Si no devuelve nada, se utiliza `void`.

2. Nombre de la Función: Debe ser descriptivo y seguir las reglas de nomenclatura de C++.

3. Parámetros: Se definen entre paréntesis. Cada parámetro debe tener un tipo de dato seguido de un nombre. Se pueden especificar múltiples parámetros separados por comas.

4. Cuerpo de la Función: Se encierra entre llaves `**{}**` y contiene las instrucciones que se ejecutarán cuando se llame a la función.

Ejemplo de Declaración y Definición

```cpp
#include <iostream>

using namespace std;
```

```
// Declaración de la función
int sumar(int a, int b);
int main() {
    int resultado = sumar(10, 20); // Llamada a la función
    cout << "El resultado es: " << resultado << endl;
    return 0;
}
// Definición de la función
int sumar(int a, int b) {
    return a + b; // Cuerpo de la función
}
```

Desglose del Ejemplo

Declaración: `***int sumar(int a, int b);***` indica que `sumar` es una función que toma dos enteros como parámetros y devuelve un entero.

Llamada: `***sumar(10, 20);***` llama a la función con `10` y `20` como argumentos.

Definición: `***int sumar(int a, int b) { return a + b; }***` define lo que hace la función: suma los dos parámetros y devuelve el resultado.

Por lo tanto para declarar una función en C++ con parámetros, se especifica el tipo de retorno, el nombre de la función, los parámetros entre paréntesis y se define el cuerpo de la función.

Para definir y utilizar funciones en Python, se siguen estos pasos:

Ejemplo en Python:

```python

def sumar(a, b):

return a + b

```

```python
def sumar(a, b):
    return a + b
```

1. Definición de la función:

- Se utiliza la palabra clave `def` seguida del nombre de la función.
- Se especifican los parámetros de entrada entre paréntesis (si los hay).
```

- El bloque de código de la función se indenta.
- Se utiliza la palabra clave `return` para devolver un valor (opcional).

```python

def nombre_funcion(parametro1, parametro2):

  # Bloque de código de la función

  resultado = operacion(parametro1, parametro2)

  return resultado

```

```python
def nombre_funcion(parametro1, parametro2):
    # Bloque de código de la función
    resultado = operacion(parametro1, parametro2)
    return resultado
```

2. Invocación de la función:

- Se llama a la función por su nombre.
- Se pasan los argumentos necesarios entre paréntesis.
- La función se ejecuta y devuelve un valor (si corresponde).

```python

valor_retornado = nombre_funcion(argumento1, argumento2)

```

**3. Funciones como módulos:**

- Las funciones pueden organizarse en módulos (archivos .py).
- Para usar las funciones de un módulo, se importa el módulo.

- Se accede a las funciones del módulo usando el nombre del módulo seguido del nombre de la función.

*```python*

*import mi_modulo*

*resultado = mi_modulo.nombre_funcion(argumento)*

```

```python
import mi_modulo
resultado = mi_modulo.nombre_funcion(argumento)
```

4. Funciones anónimas (lambda):

- Son funciones simples que no tienen nombre.
- Se definen con la palabra clave `lambda` seguida de los parámetros y un `:`
- El cuerpo de la función se especifica después del `:`.

```python

funcion_anonima = lambda parametro: operacion(parametro)

resultado = funcion_anonima(argumento)

```

```python
funcion_anonima = lambda parametro: operacion(parametro)
resultado = funcion_anonima(argumento)
```

Las funciones en Python permiten modularizar el código, reutilizar bloques de código, mejorar la legibilidad y facilitar el mantenimiento de los programas.

Las principales diferencias entre las funciones en Python y C++ son:

Declaración y Definición

Python: Las funciones se definen usando la palabra clave `def` seguida del nombre de la función y los parámetros entre paréntesis. No se especifica el tipo de retorno.

```python
def sumar(a, b):
    return a + b
```


```
def sumar(a, b):
    return a + b
```


C++: Las funciones se declaran con un tipo de retorno, el nombre de la función y los parámetros entre paréntesis. El tipo de retorno se especifica antes del nombre de la función.

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


```
1 int sumar(int a, int b) {
2     return a + b;
3 }
4
```

Qué son los tipos de datos en Python

Python es un lenguaje de tipado dinámico, por lo que no es necesario declarar el tipo de datos de las variables. El tipo se infiere en tiempo de ejecución.

Un tipo de dato se refiere a la clasificación de los valores que pueden ser almacenados en una variable. Cada tipo de dato tiene características y propiedades específicas que lo hacen adecuado para ciertos usos y operaciones.

Los tipos de datos también se utilizan para determinar cómo se almacenan los valores en la memoria de la computadora y cómo se pueden manipular y operar los valores. Por ejemplo, los números enteros se almacenan como secuencias de bits en la memoria, y se pueden realizar operaciones aritméticas como suma, resta, multiplicación y división.

1. Numéricos
2. De texto
3. Booleanos
4. De secuencias
5. De mapeo
6. Conjuntos
7. De bytes

Tipos de datos numéricos

int

Los datos tipo int (abreviatura de «entero»), en Python, son aquellos que representan números enteros. Estos son un tipo de número que no tiene parte decimal, es decir, son completos sin fracciones. En Python, los datos tipo int se representan simplemente escribiendo un número sin parte decimal. Por ejemplo:

1, 2, 3, 4, 5

Los datos tipo int se utilizan para realizar operaciones matemáticas, como sumas, restas, multiplicaciones y divisiones, así como para realizar comparaciones entre números enteros. Al trabajar con datos tipo int en Python, es importante tener en cuenta que el resultado de algunas operaciones puede

ser de tipo flotante si se utiliza la división. Por ejemplo, si se divide 7 entre 2 en Python, el resultado es 3.5, que es un número tipo float. Si se quiere obtener solo la parte entera del resultado, se puede utilizar la función «división entera» (//), que devuelve el resultado entero de la división. Por ejemplo, 7 // 2 devuelve 3 (que es el resultado entero de la división).

float

Los datos tipo float (abreviatura de «coma flotante»), en Python, son aquellos que representan números con decimales. En otras palabras, los datos tipo float son números reales que pueden tener una parte entera y una parte decimal. En Python, los datos tipo float se representan escribiendo un número con un punto decimal. Por ejemplo:

1.23, 3.14, 45.5, -12.34

Los datos tipo float se utilizan para realizar operaciones matemáticas que involucran números con decimales. Al trabajar con estos, es importante tener en cuenta que algunos cálculos pueden arrojar una precisión limitada, debido a la forma en que se almacenan los números en la memoria de la computadora. En general, los números tipo float tienen una precisión de alrededor de 15-17 dígitos decimales. Además, al realizar cálculos con números tipo float, es posible que se produzcan errores de redondeo o de exactitud que pueden afectar los resultados de la operación.

complex

Los datos tipo complex (abreviatura de «números complejos»), en Python, **son aquellos que representan números que tienen una parte real y una parte imaginaria**. En matemáticas, los números complejos se definen como la suma de una parte real y una parte imaginaria multiplicada por la unidad imaginaria (que se denota como «i» o «j»).

Los datos tipo complex se representan escribiendo un número real seguido por una «j» para indicar la parte imaginaria.

Los siguientes, son ejemplos de datos tipo complex en Python:

3 + 2j, -7.5 - 1j, 2j

Se utilizan para realizar operaciones matemáticas que involucran números complejos. Python tiene funciones incorporadas para trabajar con

números complejos, como la función **abs()** para calcular el valor absoluto de un número complejo y la función **conjugate()** para calcular el conjugado de un número complejo.

Es importante tener en cuenta que Python, al igual que la mayoría de los lenguajes de programación, utiliza una notación diferente para la unidad imaginaria en comparación con la notación matemática tradicional. En lugar de **i**, Python utiliza **j**, para denotar la unidad imaginaria.

Tipos de texto

str

Los datos tipo str (abreviatura de «cadena de caracteres» o «string»), en Python, son aquellos que representan una secuencia de caracteres. En otras palabras, los datos tipo string son texto que se puede representar mediante una serie de caracteres (letras, números, símbolos, etcétera). Los datos tipo string se representan entre comillas simples ('') o dobles ("").

Los siguientes son ejemplos de datos tipo str en Python:

"Hola mundo", 'Python es genial', "1234".

Los datos tipo string se utilizan para almacenar texto y manipular cadenas de caracteres. Python tiene muchas funciones incorporadas a fin de trabajar con cadenas de caracteres, como la función **len()** para obtener la longitud de una cadena y la función **split()** para dividir una cadena en una lista de subcadenas separadas por un carácter específico.

Es importante tener en cuenta que en Python, las cadenas de caracteres son inmutables; es decir, **no se pueden modificar una vez creadas**. Si se necesita manipular una cadena de caracteres, se hace una nueva con los cambios deseados.

Tipos booleanos

bool

Los datos tipo bool (abreviatura de «booleanos»), en Python, son aquellos que representan valores de verdad. En otras palabras, son valores verdaderos o falsos. En Python, los datos tipo bool se representan como **True o False** (con la primera letra en mayúscula).

Por ejemplo:

True, False

Se utilizan principalmente en expresiones lógicas y de control de flujo, como en declaraciones **if** y **while**. Las expresiones lógicas en Python pueden evaluar a un valor verdadero o falso, y se utilizan para tomar decisiones en el código.

Además de los valores **True y False,** Python tiene algunas funciones incorporadas que devuelven valores booleanos, como la función «bool()», que devuelve **True** si el argumento que se le pasa es verdadero, y **False** si el argumento es falaz.

Tipos de secuencias

list

Los datos tipo list (abreviatura de «listas»), en Python, son aquellos que representan una colección ordenada de elementos. En otras palabras, las listas son una estructura de datos que permite almacenar múltiples valores en un solo objeto y acceder a ellos por su posición en la lista.

En Python, los datos tipo lista se representan mediante corchetes «[ ]» que contienen los elementos de la lista separados por comas. Los elementos pueden ser de cualquier tipo de datos, incluyendo otros objetos, como otras listas.

Por ejemplo:

[1, 2, 3, 4, 5], ['a', 'b', 'c'], [1, 'a', True, [2, 3, 4]]

Los datos tipo lista se utilizan para almacenar y manipular colecciones de datos. Python tiene muchas funciones y métodos incorporados a fin de trabajar con listas, como la función **len()** para obtener la longitud de una lista y los métodos **append()** y **extend()** para agregar elementos a una lista.

Es importante tener en cuenta que las listas en Python son mutables, lo que significa que **se pueden modificar una vez que se han creado**. Esto permite agregar, eliminar o modificar elementos en una lista.

tuple

Los datos tipo tuple (abreviatura de «tuplas»), en Python, son similares a las listas, pero a diferencia de las listas, son inmutables. En otras palabras, **las tuplas son una estructura de datos que permite almacenar múltiples valores en un solo objeto** y acceder a ellos por su posición en la tupla, pero una vez creada la tupla, los elementos no pueden modificarse.

Los datos tipo tupla se representan mediante paréntesis «( )» que contienen los elementos de la tupla separados por comas. Los elementos pueden ser de cualquier tipo de datos, incluyendo otros objetos, como otras tuplas.

Por ejemplo:

(1, 2, 3, 4, 5), ('a', 'b', 'c'), (1, 'a', True, (2, 3, 4))

Los datos tipo tupla se utilizan principalmente para almacenar datos que no deben ser modificados una vez creados, como una fecha o coordenadas geográficas. Python tiene algunas funciones incorporadas a fin de trabajar con tuplas, como la función **len()** para obtener la longitud de una tupla y la función **tuple()** para crear una tupla a partir de una lista u otro iterable.

Es importante tener en cuenta que las tuplas en Python son inmutables; es decir, que no se pueden modificar una vez que se han creado. Si se necesita manipular una colección de datos, se debe usar una lista en su lugar.

range

Los datos tipo range, en Python, representan una secuencia inmutable de números enteros. Un objeto de tipo range se utiliza comúnmente para generar una secuencia de números enteros para su uso en un bucle for.

La función **range()** devuelve un objeto de tipo range. El objeto range toma tres argumentos: el valor inicial, el valor final (no incluido en la secuencia) y el tamaño del paso. Por defecto, el valor inicial es 0 y el tamaño del paso es 1.

Por ejemplo:

range(0, 10, 1), range(0, 10), range(1, 11, 2)

El primer ejemplo crea un objeto range que representa la secuencia de números enteros del 0 al 9 (inclusivo) con un tamaño de paso de 1. El segundo

ejemplo es equivalente al primer ejemplo, ya que el valor inicial y el tamaño del paso se asumen como 0 y 1, respectivamente, por defecto. El tercer ejemplo crea un objeto range que representa la secuencia de números enteros del 1 al 9 (inclusivo) con un tamaño de paso de 2.

Tipos de mapeo

dict

Los datos tipo dict (abreviatura de «diccionario»), en Python, son una estructura de datos que permite almacenar un conjunto de datos como pares clave-valor, donde cada valor es accesible a través de una clave única. En otras palabras, los diccionarios permiten asociar valores con claves.

Los datos tipo dict se representan mediante llaves «{ }» que contienen una serie de pares clave-valor separados por comas y cada par se separa por dos puntos «:». Las claves en un diccionario son únicas y pueden ser de cualquier tipo inmutable, como una cadena, un número entero o una tupla. Los valores también pueden ser de cualquier tipo de datos.

Por ejemplo:

{'nombre': 'Juan', 'edad': 25, 'ciudad': 'Madrid'}, {1: 'uno', 2: 'dos', 3: 'tres'}, {(1, 2): 'valor de la tupla', 'clave': [1, 2, 3]}

En Python, los diccionarios son útiles para almacenar y acceder a datos de manera eficiente. Los datos pueden ser accedidos en un diccionario utilizando su clave, como en el siguiente ejemplo:

```
D = {'NOMBRE': 'JUAN', 'EDAD': 25, 'CIUDAD': 'MADRID'}

PRINT(D['NOMBRE']) # SALIDA: 'JUAN'
```

También es posible modificar los valores en un diccionario, agregar nuevos pares clave-valor y eliminar pares clave-valor existentes. Los diccionarios son muy versátiles y se utilizan ampliamente en programación.

Tipos de conjuntos

set

Los datos tipo set, en Python, son una colección de elementos únicos e inmutables; es decir, no debe haber duplicados en un conjunto y no se pueden cambiar después de ser creados. Un conjunto se crea utilizando llaves «{ }» o la función **set()** y los elementos se separan por comas.

Por ejemplo:

```
{1, 2, 3, 4, 5}{

'HOLA', 'MUNDO', 'PYTHON'}

SET([1, 2, 3, 4])
```

En el primer ejemplo se crea un conjunto con los elementos 1, 2, 3, 4 y 5; en el segundo, un conjunto con las cadenas ‘hola’, ‘mundo’ y ‘python’; y en el tercero, un conjunto a partir de una lista con los elementos 1, 2, 3 y 4.

Los conjuntos en Python son útiles para realizar operaciones de conjuntos como la unión, la intersección, la diferencia y la comprobación de pertenencia. También se pueden usar para eliminar duplicados de una lista o para encontrar elementos únicos en una secuencia.

Por ejemplo, la siguiente secuencia de código utiliza conjuntos para eliminar duplicados de una lista:

```
LISTA = [1, 2, 3, 4, 4, 3, 5, 6, 6]

CONJUNTO = SET(LISTA)

LISTA_SIN_DUPLICADOS = LIST(CONJUNTO)

PRINT(LISTA_SIN_DUPLICADOS) # SALIDA: [1, 2, 3, 4, 5, 6]
```

frozenset

Los datos tipo frozenset son similares a los conjuntos (sets), pero son inmutables; es decir, una vez creados no se pueden modificar. Se crean utilizando la función **frozenset()** y se usan para almacenar elementos únicos, como los conjuntos.

A diferencia de los conjuntos, los datos tipo frozenset se pueden utilizar como elementos de diccionarios, ya que son inmutables y, por lo tanto, hashables. Además, los datos tipo frozenset es factible usarlos como elementos de otros conjuntos, ya que también son inmutables.

Aquí hay algunos ejemplos de datos tipo frozenset en Python:

```
FSET = FROZENSET([1, 2, 3, 4])

PRINT(FSET) # SALIDA: FROZENSET({1, 2, 3, 4})

FSET2 = FROZENSET(('HOLA', 'MUNDO', 'PYTHON'))

PRINT(FSET2) # SALIDA: FROZENSET({'HOLA', 'MUNDO', 'PYTHON'})
```

En el primer ejemplo, se crea un frozenset con los elementos 1, 2, 3 y 4; y en el segundo, un frozenset con las cadenas 'hola', 'mundo' y 'python'. Al ser inmutables, no se pueden agregar o quitar elementos de un frozenset después de que se ha creado.

Tipos de bytes

bytes

Los datos tipo bytes, en Python, son una secuencia inmutable de bytes. Representan una cadena de bytes en bruto y se utilizan principalmente para trabajar con datos binarios como archivos, sockets de red, criptografía, entre otros.

Los datos tipo bytes se pueden crear utilizando una sintaxis de prefijo **b** antes de la cadena de bytes o utilizando la función **bytes()**. Los elementos de los datos tipo bytes son enteros en el rango de 0 a 255.

Aquí hay algunos ejemplos de datos tipo bytes en Python:

```
B = B'HOLA'

PRINT(B) # SALIDA: B'HOLA'

B2 = BYTES([0X68, 0X6F, 0X6C, 0X61])

PRINT(B2) # SALIDA: B'HOLA'
```

En el primer ejemplo se crea un objeto bytes utilizando la sintaxis de prefijo **b**; y en el segundo, un objeto bytes utilizando la función **bytes()** y una lista de enteros que representan los códigos ASCII de los caracteres 'h', 'o', 'l' y 'a'.

Los datos tipo bytes se pueden utilizar para codificar y decodificar datos binarios, y también a fin de realizar operaciones binarias, como **XOR** y desplazamiento de bits. Además, los datos tipo bytes es factible convertirlos a objetos bytearray, que son mutables.

bytearray

Los datos tipo bytearray, en Python, son similares a los datos tipo bytes, pero son mutables; es decir, una vez creados se pueden modificar. Los datos tipo bytearray se utilizan para representar secuencias de bytes modificables, como archivos binarios, buffers de red, entre otros.

Los datos tipo bytearray se pueden crear utilizando la función **bytearray()**. Al igual que los datos tipo bytes, los elementos de los datos tipo bytearray son enteros en el rango de 0 a 255.

Aquí hay algunos ejemplos de datos tipo bytearray en Python:

```
B_ARR = BYTEARRAY(B'HOLA')

PRINT(B_ARR) # SALIDA: BYTEARRAY(B'HOLA')

B_ARR[1] = 111 # CAMBIA EL SEGUNDO ELEMENTO A 'O'

PRINT(B_ARR) # SALIDA: BYTEARRAY(B'OOLA')

B_ARR2 = BYTEARRAY([104, 111, 108, 97])

PRINT(B_ARR2) # SALIDA: BYTEARRAY(B'HOLA')
```

En el primer ejemplo se crea un objeto bytearray utilizando la función **bytearray()**. En el segundo, se modifica el siguiente elemento del objeto bytearray a 'o'. En el tercero, se crea un objeto bytearray utilizando una lista de enteros que representan los códigos ASCII de los caracteres 'h', 'o', 'l' y 'a'.

Los datos tipo bytearray se pueden utilizar para realizar operaciones binarias, como XOR y desplazamiento de bits, y también convertir a objetos bytes, que son inmutables.

Tipos de datos principales en C++

C++ es un lenguaje de tipado estático, por lo que se debe declarar el tipo de datos de las variables y parámetros. El tipo se verifica en tiempo de compilación.

4 Son los tipos de datos principales en C++ con los que podremos hacer prácticamente todo, a menos que necesitemos un tipo de dato que tenga mayor capacidad en memoria, y esto se debe a que al declarar una variable esta reserva un espacio en la memoria del dispositivo llamémosle memoria RAM en las computadoras, cada tipo de dato tiene un tamaño de memoria que reserva, suelen ser muy grandes pero por si acaso el tipo de dato no es lo suficientemente grande entonces existen otras palabras clave que nos permitirán incrementar el tamaño del tipo de dato.

Entero: recordemos que un número entero simplemente es cualquier número(natural) sin punto por ejemplo 0 es de tipo entero, 846582 también lo es pero 1.2 no porque tiene un punto y por eso no podríamos almacenar este valor en una variable de tipo entero, para declarar una variable de tipo entero utilizamos la palabra reservada int que significa integer en inglés y como ya era de suponer se traduce como entero al español, posteriormente añadimos el nombre de la variable veamos ahora ejemplos de cómo se usa:

```
int casas = 5;
```

```
int objetos;
```

En ese ejemplo tenemos nuestra variable casas de tipo int con el número 5, también tenemos otra variable llamada objetos que todavía no contiene ningún valor y a futuro se le puede asignar el valor.

Punto flotante: este tipo de datos es la solución para números que tienen decimales como es el caso de 98.3 como vimos anteriormente no podemos incluirlo en variables de tipo entero, para declararlos se utiliza la palabra reservada float, ejemplos de uso son:

```
float metros=8.53;
```

```
float dólares;
```

Texto: en este tipo de datos además de texto también pueden incluirse números pero no pueden hacerse operaciones como suma, resta u otras para lo cual se necesitaría int o float porque el lenguaje los interpreta como texto, para declarar una variable con este tipo se emplea la palabra char que significa character y se traduce como carácter, pero en este tipo de datos además tenemos que indicar la cantidad de caracteres máxima que se incluirán + 1 así es como "Estetexto" contiene 9 caracteres y reservamos un total de 10 como mínimo, se reserva entre paréntesis cuadrado, veamos cómo.

```
char carta[8] = "manzana" ;
```

```
char hoja[800] = "hola" ;
```

```
char ultimo[4000];
```

No pasara mucho tiempo antes de que te des cuenta que esto puede ser un problema, porque no siempre sabemos que tamaño tendrá nuestra cadena de caracteres pero esto es así porque C++ necesita reservar la memoria para esa cadena de caracteres con anticipación, así que podríamos calcular diciendo si mi texto puede tener 500 palabras reservare 4000 o incluso 2000 y así nunca se produciría algún error en la aplicación, pero en verdad

esto sería reservar memoria innecesaria por tanto para esto se ha creado varias soluciones una de ellas que conviene aprender ya es el uso del tipo de datos string, que en verdad es una clase pero su implementación es igual a la de cualquier otro tipo de datos.

```
string libro = "Aquí un pequeño ejemplo";
```

```
string nombre;
```

Booleano: Una variable de tipo booleano solo acepta dos tipos de valores estos son true que significa verdadero o false que significa falso, se utiliza si se quiere guardar el estado de un resultado por ejemplo si preguntamos 1=5 el resultado será false porque uno no es igual a cinco y podemos almacenarlo en una variable de tipo booleano, para declarar una variable booleana utilizamos la palabra reservada bool, ejemplos serian:

```
bool reunir=false;
```

```
bool bicicleta;
```

Nota: En algunos libros o cursos se suele mencionar la palabra reservada void como un tipo de datos, pero en verdad no es un tipo, void se usa para indicar que algo no tiene valor, en variables puede utilizarse como un puntero pero eso va más adelante en el capítulo de punteros y no en este de tipos de variables.

Si se necesita incrementar el tamaño de una variable de texto más allá de 1 byte puede utilizarse el tipo wchar_t, o si se necesita incrementar el tamaño de una variable de tipo float de 4 a 8 bytes se utiliza el tipo double que

es básicamente un float de doble tamaño, aunque en verdad con estos tamaños estándar suele realizarse la mayoría de programas básicos pero a futuro puede llegar a necesitar usarse los otros.

Paso de Argumentos

Python: Los argumentos se pasan por valor o por referencia según el tipo de dato. Las listas y diccionarios se pasan por referencia.

C++: Los argumentos se pasan por valor de forma predeterminada. Se pueden pasar por referencia usando `&` o por puntero.

Retorno de Valores

Python: Una función puede retornar múltiples valores separados por comas. Se pueden retornar diferentes tipos de datos.

C++: Una función solo puede retornar un valor. El tipo de retorno debe ser compatible con el tipo de dato declarado en la función.

Funciones Anónimas

Python: Soporta funciones anónimas (lambda) que permiten definir funciones simples en una sola línea.

```python

sumar = *lambda a, b: a* + *b*

```

```python
sumar = lambda a, b: a + b
```

C++: No tiene una sintaxis específica para funciones anónimas, pero se pueden simular con punteros a funciones.
```

Python tiene una sintaxis más concisa y flexible para definir funciones, mientras que C++ requiere una declaración más explícita de tipos y retornos. Ambos lenguajes permiten crear y utilizar funciones, pero con diferencias en la sintaxis y el manejo de tipos de datos.

Importación y Llamado de Módulos

Los módulos son archivos que contienen funciones y variables. Se importan para usar su contenido.

Ejemplo en C++:

```cpp

#include "mi_modulo.h"

```

```cpp
1    #include "mi_modulo.h"
```

Ejemplo en Python:

```python

import mi_modulo

```

```python
import mi_modulo
```

Invocación de Funciones

Para ejecutar una función, se la llama por su nombre y se pasan los argumentos necesarios.

Ejemplo en C++:

```cpp

int resultado = sumar(10, 20);

```

```cpp
1   int resultado = sumar(10, 20);
```

Ejemplo en Python:

```python

resultado = sumar(10, 20)

```

```python
resultado = sumar(10, 20)
```

Funciones como Módulos

En Python, las funciones pueden organizarse en módulos para una mejor modularidad.

Ejemplo en Python:

```python

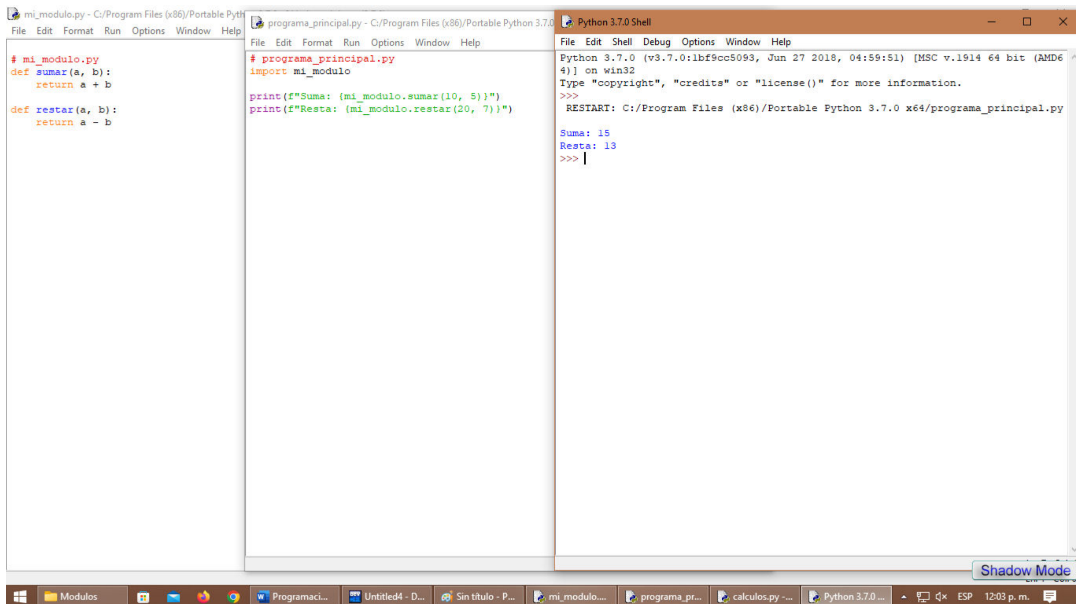
 # mi_modulo.py

def sumar(a, b):

return a + b

```python
def restar(a, b):
    return a - b
```

```python
# programa_principal.py
import mi_modulo

print(f"Suma: {mi_modulo.sumar(10, 5)}")
print(f"Resta: {mi_modulo.restar(20, 7)}")
```



Ejercicios Propuestos

1. Crear una función que calcule el área de un círculo dado su radio.

2. Escribir un programa que importe un módulo con funciones para operaciones matemáticas básicas (suma, resta, multiplicación, división) y las utilice.

3. Desarrollar un programa que defina funciones para calcular el factorial y la potencia de un número.

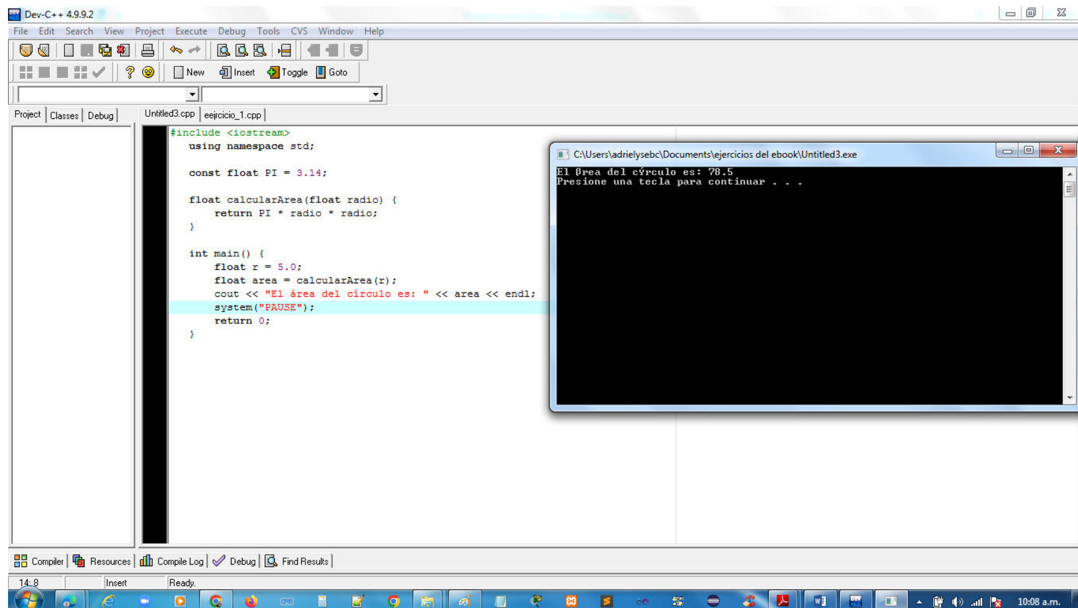
4. Crear un módulo con funciones para convertir entre grados Celsius, Fahrenheit y Kelvin. Importar el módulo y probar las funciones.

Soluciones a los Ejercicios Propuestos

1. Solución para calcular el área de un círculo:

C++:

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

const float PI = 3.14;

float calcularArea(float radio) {
    return PI * radio * radio;
}

int main() {
    float r = 5.0;
    float area = calcularArea(r);
    cout << "El área del círculo es: " << area << endl;
    return 0;
}
```

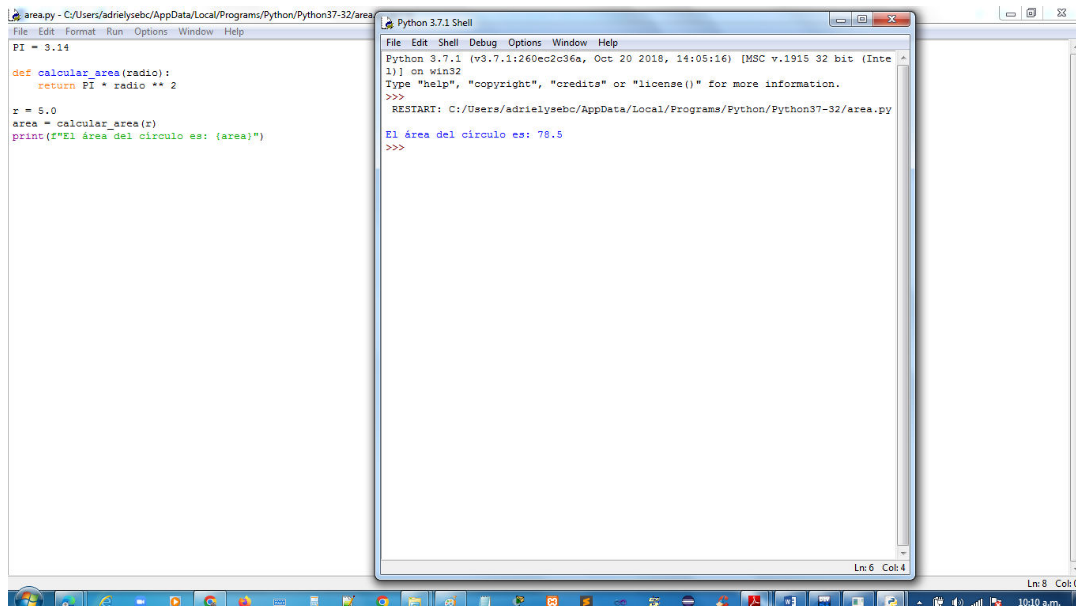


Python:

```python
PI = 3.14

def calcular_area(radio):
    return PI * radio ** 2

r = 5.0
area = calcular_area(r)
print(f"El área del círculo es: {area}")
```



2. Solución para operaciones matemáticas básicas:

C++:

```
```cpp

#include <iostream>

#include "operaciones.h" // Asegúrate de que este archivo contiene las declaraciones de las funciones

using namespace std;

int main() {

    int a = 10, b = 5;

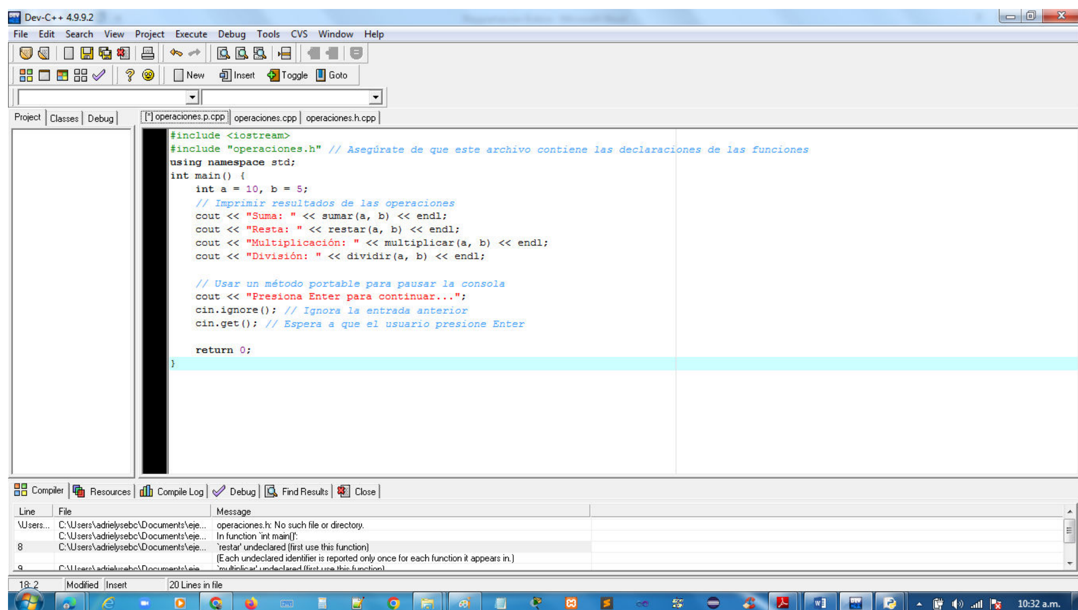
    // Imprimir resultados de las operaciones

    cout << "Suma: " << sumar(a, b) << endl;
```
```

```
    cout << "Resta: " << restar(a, b) << endl;
    cout << "Multiplicación: " << multiplicar(a, b) << endl;
    cout << "División: " << dividir(a, b) << endl;

    // Usar un método portable para pausar la consola
    cout << "Presiona Enter para continuar...";
    cin.ignore(); // Ignora la entrada anterior
    cin.get(); // Espera a que el usuario presione Enter

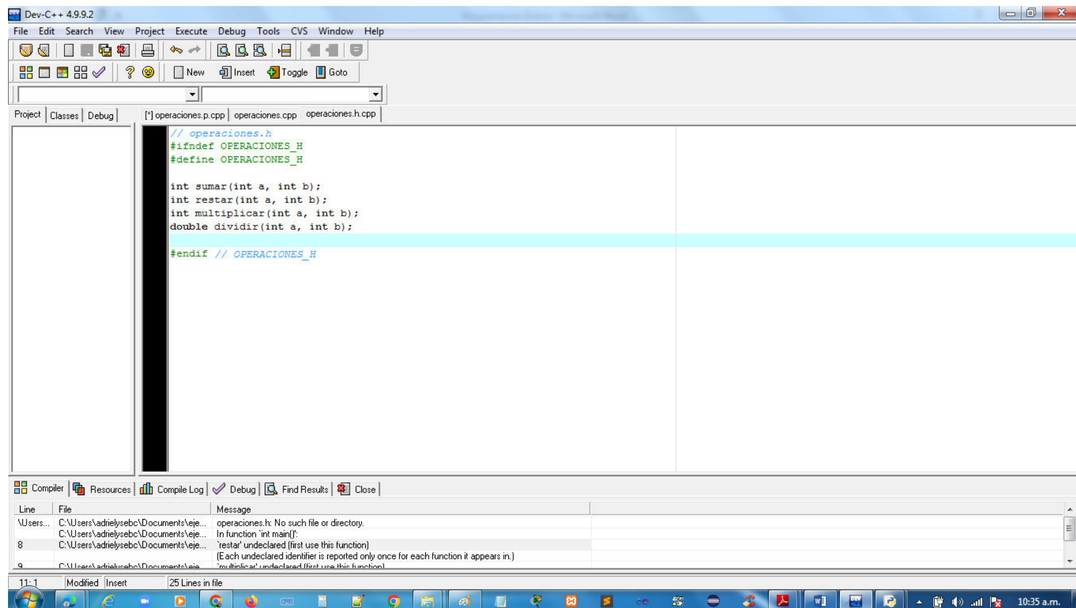
    return 0;
}
```



Nota. No se manejó el `system("PAUSE")`: Este comando es específico de Windows y no es portable. En su lugar, se utiliza `cin.ignore()` y `cin.get()` para esperar a que el usuario presione Enter antes de cerrar la consola.

El Archivo "operaciones.h": Asegúrate de que el archivo `operaciones.h` contenga las declaraciones de las funciones `sumar`, `restar`, `multiplicar` y `dividir`. Aquí tienes un ejemplo de cómo podría verse:

```cpp
// operaciones.h
#ifndef OPERACIONES_H
#define OPERACIONES_H

int sumar(int a, int b);
int restar(int a, int b);
int multiplicar(int a, int b);
double dividir(int a, int b);

#endif // OPERACIONES_H
```



Implementación de las funciones: Asegúrate de que las funciones estén implementadas en un archivo de código fuente correspondiente, por ejemplo, `operaciones.cpp`.

```cpp
// operaciones.cpp
#include "operaciones.h"
int sumar(int a, int b) {
    return a + b;
}
int restar(int a, int b) {
    return a - b;
}
```

```
int multiplicar(int a, int b) {
    return a * b;
}
double dividir(int a, int b) {
    if (b != 0) {
        return static_cast<double>(a) / b; // Conversión a double para evitar la división entera
    } else {
        throw std::invalid_argument("División por cero"); // Manejo de error
    }
}
```

Python:

En el presente código Python que hemos proporcionado, es importante asegurarse de que el módulo `operaciones` esté correctamente definido y que contenga las funciones necesarias (`sumar`, `restar`, `multiplicar`, `dividir`). A continuación, presentamos una version mejorada del código, junto con ejemplos de cómo podrían definirse esas funciones en el módulo `operaciones`.

```python
import operaciones

a = 10

b = 5

# Imprimir los resultados de las operaciones

print("Suma:", operaciones.sumar(a, b))

print("Resta:", operaciones.restar(a, b))

print("Multiplicación:", operaciones.multiplicar(a, b))


# Manejo de división por cero

try:

    print("División:", operaciones.dividir(a, b))

except ZeroDivisionError as e:

    print("Error en la división:", e)
```

Definición del Módulo `operaciones`

Asegúrate de que el archivo `operaciones.py` contenga las siguientes definiciones de funciones:

```python
# operaciones.py
def sumar(a, b):
    return a + b


def restar(a, b):
    return a - b


def multiplicar(a, b):
    return a * b


def dividir(a, b):
    if b == 0:
        raise ZeroDivisionError("No se puede dividir entre cero.")
    return a / b
```

Explicación del código:

1. Manejo de la División por Cero: Se ha añadido un bloque `try-except` para manejar el caso en que se intente dividir entre cero, lo cual es una buena práctica para evitar que el programa falle.

2. Impresión de Resultados: Se han mantenido las impresiones de los resultados de las operaciones, asegurando que el código sea claro y fácil de seguir.

3. Funciones en el Módulo: Se ha proporcionado un ejemplo de cómo definir las funciones en el archivo `operaciones.py`, asegurando que cada función realice la operación correspondiente.

Consideraciones Adicionales

- Estructura de Archivos: Asegúrate de que ambos archivos (`main.py` y `operaciones.py`) estén en el mismo directorio o que el módulo `operaciones` esté en el `PYTHONPATH` para que se pueda importar correctamente.
- Pruebas: Considera agregar pruebas unitarias para cada función en `operaciones.py` para verificar que funcionan correctamente en diferentes *escenarios.*

Realizando los pasos antes mencionados el código debería funcionar correctamente y manejar adecuadamente los errores en la división.

3. Solución para factorial y potencia:

C++:

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

int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}

int potencia(int base, int exponente) {
    if (exponente == 0)
        return 1;
    else
        return base * potencia(base, exponente-1);
}
```

```cpp
int main() {
    cout << "Factorial de 5: " << factorial(5) << endl;
    cout << "2 elevado a 4: " << potencia(2, 4) << endl;
    return 0;
}
```

Python:

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)


def potencia(base, exponente):
    if exponente == 0:
        return 1
    else:
        return base * potencia(base, exponente-1)
print("Factorial de 5:", factorial(5))
print("2 elevado a 4:", potencia(2, 4))
```

~~~
```
~~~

4. Solución para conversión de temperaturas:

C++:

```cpp
#include <iostream>

#include "conversion_temperatura.h"

using namespace std;


int main() {

    float celsius = 25.0;

    float fahrenheit = convertirCelsiusAFahrenheit(celsius);

    float kelvin = convertirCelsiusAKelvin(celsius);


    cout << celsius << " °C = " << fahrenheit << " °F" << endl;

    cout << celsius << " °C = " << kelvin << " K" << endl;

    return 0;

}
```

Python:

```python
import conversion_temperatura
```

```
celsius = 25.0
fahrenheit = conversion_temperatura.convertir_celsius_fahrenheit(celsius)
kelvin = conversion_temperatura.convertir_celsius_kelvin(celsius)

print(f"{celsius} °C = {fahrenheit} °F")
print(f"{celsius} °C = {kelvin} K")
```

Avances de C++ y Python, Proyecciones Futuras y la Importancia de su Uso.

Avances Recientes

C++:

Mejoras en la Lengua: Con la introducción de C++11, C++14 y C++17, se han añadido características como expresiones lambda, inicialización uniforme, y mejoras en la biblioteca estándar, que facilitan la programación y aumentan la eficiencia.

Programación Concurrente: Se han incorporado bibliotecas para facilitar la programación multihilo, lo que es crucial en aplicaciones modernas que requieren alta eficiencia y rendimiento.

Estándares Modernos: La adopción de estándares modernos ha permitido a los desarrolladores escribir código más limpio y seguro.

Python:

Popularidad y Comunidad: Python ha visto un crecimiento exponencial en popularidad, especialmente en áreas como ciencia de datos, aprendizaje automático y desarrollo web.

Bibliotecas y Frameworks: La expansión de bibliotecas como TensorFlow, Pandas y Flask ha permitido a los desarrolladores abordar una amplia variedad de problemas con facilidad.

Simplicidad y Legibilidad: Python sigue siendo valorado por su sintaxis sencilla y su enfoque en la legibilidad del código, lo que lo hace accesible para principiantes.

Proyecciones Futuras

C++: Se espera que continúe evolucionando con nuevas versiones que incorporen características que faciliten la programación segura y eficiente. La comunidad está trabajando en mejorar la interoperabilidad con otros lenguajes y plataformas.

Python: Se proyecta que Python seguirá siendo un lenguaje clave en el desarrollo de inteligencia artificial y análisis de datos. La comunidad está enfocada en mejorar el rendimiento y la eficiencia del lenguaje, así como en la creación de herramientas que faciliten el desarrollo de aplicaciones complejas.

Importancia de su Uso

C++:

Desarrollo de Software de Alto Rendimiento: C++ es fundamental en el desarrollo de sistemas operativos, motores de videojuegos y aplicaciones que requieren un control preciso sobre los recursos del sistema.

Programación Orientada a Objetos: Facilita la creación de software modular y reutilizable, lo que es esencial en proyectos grandes y complejos.

Python:

Versatilidad y Facilidad de Aprendizaje: Python es ampliamente utilizado en educación y en la industria debido a su facilidad de uso y su capacidad para manejar tareas complejas con menos código.

Comunidad Activa: La gran comunidad de Python contribuye continuamente al desarrollo de nuevas herramientas y bibliotecas, lo que amplía su aplicabilidad en diversas áreas.

En Conclusión tanto C++ como Python tienen un papel crucial en el desarrollo de software moderno. Sus avances recientes, proyecciones futuras

y la importancia de su uso en diversas aplicaciones los consolidan como lenguajes fundamentales en el campo de la programación. La elección entre C++ y Python dependerá de las necesidades específicas del proyecto, la eficiencia requerida y la experiencia del desarrollador.

RECURSOS INTERACTIVOS

- https://pseudocodigoejemplos.com/algoritmos-en-python/
- https://www.tutorialesprogramacionya.com/pythonya/
- https://www.w3schools.com/python/default.asp
- https://paiza.io/projects/jOvAZWuJEVXUbrMBtD4sMw?language=java
- https://www.programarya.com/Cursos/C++/Entornos
- https://www.tutorialesprogramacionya.com/pythonya/detalleconcepto.php?punto=2&codigo=2&inicio=0
- https://digitalnestweb.com/10-ejercicios-resueltos-de-python-listas/
- https://www.jetbrains.com/es-es/pycharm/download/?section=windows
- https://digitalnestweb.com/10-ejercicios-resueltos-de-python-listas/
- https://cienciaeducacion502.blogspot.com/2013/12/antecedentes-del-lenguaje-c.html
- https://beastieux.com/2012/01/02/programacion-en-c-declaracion-de-variables-y-tipos-de-datos/
- https://www.programacionfacil.org/cursos/c++/index.html
- https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/5733859/b10a159f-1167-4512-aeb9-ba8667a7287d/paste.txt
- https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/5733859/4c5670d0-f9b0-49a5-9651-3b5465a1755e/paste-2.txt
- https://ellibrodepython.com/
- http://slent.iespana.es/programacion/index.html

REFERENCIAS CONSULTADAS

- **Como programar en C/C++.** H.M Deitel, P.J Deitel. Prentice Hall, 2ª ed. 1995.
- **Programación orientada a objeto con C++.** Daniel Ceballos. Ed. rama 1993.
- **Programación en C++.** Enríquez Hernández / José Hernández. Ed. Paraninfos 1993.
- **Documentación de Python en Español.** Guido van Rossum and the Python Development Team. Versión 3.8.3rc1. 2013.
- **Informática aplicada prácticas para aprender a programar en lenguaje C.** Universidad Politécnica de Cartagena. Pedro María Alcover Garau. Ediciones UPCT. 2021.
- **Programación en C++ Un enfoque práctico.** Luis Joyanes Aguilar/Lucas Sánchez García. Serie Schaum. 2006.
- **Algoritmos Resueltos con Python.** Enrique Edgardo Cóndor Tinoco/Marco Antonio De La Cruz Rocca. Editorial Eidec
- **Fundamentos De Programación: Python.** Miguel Toro Bonilla. Editorial Universidad de Sevilla. 2022.