1. Tipos de dato

Índice

TARGET DECK: Materia:: Parcial I::

En C++, hay varios tipos de datos disponibles para almacenar diferentes tipos de valores. Aquí tienes algunos ejemplos de tipos de datos junto con su declaración:

1. **Enteros (int):** Utilizados para almacenar números enteros.

```
int numero = 10;
```

2. **Punto flotante (float):** Utilizado para almacenar números decimales de precisión simple.

```
float valor = 3.14;
```

3. **Doble precisión (double):** Utilizado para almacenar números decimales de doble precisión.

```
double pi = 3.14159;
```

4. **Carácter (char):** Utilizado para almacenar un único carácter.

```
char letra = 'A';
```

5. **Booleano (bool):** Utilizado para almacenar valores verdaderos o falsos.

```
bool esVerdadero = true;
```

6. **Cadena de caracteres (string):** Utilizado para almacenar secuencias de caracteres.

```
string mensaje = "Hola, mundo";
```

7. **Tipo de dato vacío (void):** Utilizado en funciones para indicar que no se devuelve ningún valor.

```
void funcionSinRetorno() {
    // Código de la función
}
```

8. **Punteros:** Utilizados para almacenar direcciones de memoria de otros tipos de datos.

```
int numero = 5;
int *punteroNumero = &numero;
```

9. **Arreglos:** Utilizados para almacenar colecciones de elementos del mismo tipo.

```
int numeros[5] = {1, 2, 3, 4, 5};
```

> **Importante**
>
> En C++, el llamar al nombre de un arreglo sin índice se convierte automáticamente en un puntero al primer elemento del arreglo.

> **Importante 2**
>
> No es posible hacer una asignación directa de un arreglo a otro arreglo en C++. La asignación de arreglos no funciona de esa manera en el lenguaje. Si intentas hacer algo como `int arregloDos[3] = arreglo;`, obtendrás un error de compilación. Para copiar el contenido de un arreglo en otro, necesitarás utilizar un bucle o alguna función que realice la copia elemento por elemento.

10. **Estructuras (struct):** Utilizadas para agrupar diferentes tipos de datos bajo una sola entidad.

```
struct Persona {
    string nombre;
    int edad;
};

Persona persona1 = {"Juan", 25};
```

11. **Enumeraciones (enum):** Utilizadas para crear un conjunto de constantes con nombre.

```
enum DiaSemana {
    Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo
};

DiaSemana dia = Miercoles;
```

Punteros

Los punteros son una característica fundamental en C++ y son una herramienta poderosa que te permite manipular directamente la memoria en el programa.

1. ¿Qué es un puntero?

Un puntero en C++ es una variable que almacena la dirección de memoria de otra variable. En lugar de contener directamente el valor, un puntero guarda la ubicación en la memoria donde se encuentra el valor. Esto permite acceder y manipular directamente la memoria, lo que puede ser útil en situaciones como la gestión dinámica de memoria y la eficiencia en el uso de recursos.

2. Declaración de punteros:

Para declarar un puntero en C++, utilizamos el tipo de dato seguido de un asterisco (*). Aquí tienes un ejemplo:

```
int numero = 42;  // Una variable entera
int *puntero;     // Declaración de un puntero a un entero
puntero = &numero;  // Asignar la dirección de 'numero' al puntero
```

3. Operador de dirección (&):

El operador de dirección (&) se utiliza para obtener la dirección de memoria de una variable. En el ejemplo anterior, `&numero` devuelve la dirección de memoria donde se almacena `numero`.

4. Acceso al valor a través de un puntero:

Podemos acceder al valor apuntado por un puntero utilizando el operador de desreferenciación (*). Esto nos permite obtener o modificar el valor almacenado en esa dirección de memoria.

```
int valor = *puntero;  // Obtiene el valor apuntado por 'puntero'
*puntero = 99;         // Modifica el valor almacenado en la dirección apuntada
```

5. Aritmética de punteros:

Los punteros pueden ser manipulados mediante operaciones matemáticas para apuntar a diferentes ubicaciones de memoria. Esto es especialmente útil al trabajar con arreglos.

```
int arreglo[3] = {10, 20, 30};
int *p = arreglo;  // 'p' apunta al primer elemento del arreglo
int segundoValor = *(p + 1);  // Acceso al segundo elemento: 20
```

> En C++, el nombre de un arreglo sin índice se convierte automáticamente en un puntero al primer elemento del arreglo.

```
int arreglo[3] = {10, 20, 30};
`int *puntero = &arreglo[0];  // Uso explícito de la dirección de memoria del primer elemento`
int segundoValor = *(p + 1);  // Acceso al segundo elemento: 20
```

Operaciones Aritméticas en Punteros y Tamaño del Tipo de Dato

En C++, los punteros pueden manipularse con operaciones aritméticas, como la suma y la resta. Esta característica es especialmente útil cuando trabajamos con arreglos y estructuras de datos. Cuando sumamos o restamos valores a un puntero, el compilador ajusta automáticamente la dirección en memoria basándose en el tamaño del tipo de dato al que apunta el puntero.

Ejemplo: Punteros y Tamaño de Tipo de Dato

Supongamos que tenemos un arreglo de enteros `int arreglo[3] = {10, 20, 30};`. Cada entero en este arreglo ocupa 4 bytes en memoria. Ahora, tenemos un puntero `int *p` que apunta al primer elemento del arreglo (`arreglo[0]`).

Si hacemos la operación `p + 1`, en lugar de avanzar solo un byte en memoria, el puntero avanzará 4 bytes, que es el tamaño de un entero.

```
int arreglo[3] = {10, 20, 30};
int *p = arreglo;      // 'p' apunta al primer elemento del arreglo
int *siguienteElemento = p + 1;  // Avanza 4 bytes (tamaño de un int)

// 'siguienteElemento' apunta ahora a 'arreglo[1]'
```

El resultado es que `siguienteElemento` ahora apunta al segundo elemento del arreglo (`arreglo[1]`), ya que el puntero avanzó un espacio de tamaño entero en memoria.

Conclusión

En resumen, puedes sumar o restar valores a un puntero en C++ porque el compilador realiza automáticamente los ajustes necesarios según el tamaño del tipo de dato al que apunta el puntero. Esto hace que sea más sencillo trabajar con arreglos y estructuras, permitiéndote navegar por la memoria de manera eficiente sin preocuparte por los detalles de la representación de la memoria.

---

6. Punteros y memoria dinámica:

Los punteros son esenciales para trabajar con memoria dinámica, como la que se asigna durante la ejecución de un programa. Usamos `new` para asignar memoria dinámica y obtener un puntero a esa memoria.

```
int *pValorNuevo = new int;  // Asigna memoria para un entero
*pValorNuevo = 75;           // Asigna un valor a la memoria
```

```
delete pValorNuevo;          // Libera la memoria cuando ya no es necesaria
```

7. Punteros y funciones:

Los punteros también se utilizan en las funciones para pasar valores por referencia y permitir que la función modifique directamente los valores de las variables externas.

```
void duplicar(int *num) {
    *num *= 2;  // Duplica el valor apuntado por 'num'
}

int numero = 5;
duplicar(&numero);  // 'numero' se convierte en 10
```

8. Punteros nulos y punteros salvajes:

Un puntero nulo es aquel que no apunta a ninguna dirección de memoria. Es importante inicializar los punteros antes de utilizarlos. Un puntero salvaje es aquel que no ha sido inicializado correctamente y apunta a una dirección de memoria desconocida.

```
int *punteroNulo = nullptr;  // Puntero nulo
int *punteroSalvaje;         // Puntero sin inicializar (salvaje)
```

9. Uso seguro de punteros:

El uso inadecuado de punteros puede llevar a errores de memoria y comportamientos no deseados. Es importante manejarlos con cuidado y utilizar herramientas como punteros inteligentes (`std::unique_ptr`, `std::shared_ptr`) para gestionar la memoria de manera más segura.

En resumen, los punteros en C++ te permiten manipular directamente la memoria y son esenciales para trabajar con estructuras de datos dinámicas y optimizar el uso de recursos. Sin embargo, requieren cuidado y comprensión para evitar errores. Con práctica y comprensión, puedes aprovechar su potencial para mejorar tus habilidades de programación.

2. Operadores

Índice

TARGET DECK: Materia:: Parcial I::

Operadores

Operadores Aritméticos:

- `+`: Suma dos valores.

```
int suma = 5 + 3;  // suma es 8
```

- `-`: Resta dos valores.

```
int resta = 10 - 4;  // resta es 6
```

- `*`: Multiplica dos valores.

```
int producto = 3 * 7;  // producto es 21
```

- `/`: Divide el primer valor por el segundo.

```
double division = 15.0 / 2.0;  // division es 7.5
```

- `%`: Calcula el residuo de la división entera.

```
int residuo = 17 % 5;  // residuo es 2
```

Operadores de Asignación:

- `=`: Asigna el valor de la derecha a la variable de la izquierda.

  ```
  int x = 10;
  ```

- `+=`, `-=`, `*=`, `/=`, `%=`: Realizan una operación y luego asignan el resultado a la variable de la izquierda.

  ```
  int a = 5;
  a += 3;  // a es ahora 8
  ```

Operadores de Comparación:

- `==`: Comprueba si dos valores son iguales.

  ```
  bool igual = (10 == 10);  // igual es true
  ```

- `!=`: Comprueba si dos valores son diferentes.

  ```
  bool diferente = (5 != 3);  // diferente es true
  ```

- `<`, `>`, `<=`, `>=`: Comprueban si un valor es menor, mayor, menor o igual, o mayor o igual que otro.

  ```
  bool menor = (7 < 10);  // menor es true
  ```

Operadores Lógicos:

- `&&`: Evalúa si ambas expresiones son verdaderas.

  ```
  bool yLogico = (true && false);  // yLogico es false
  ```

- `||`: Evalúa si al menos una de las expresiones es verdadera.

```
bool oLogico = (true || false);  // oLogico es true
```

- `!`: Niega el valor de una expresión.

```
bool negacion = !(5 < 3);  // negacion es true
```

Operadores de Incremento y Decremento:

- `++`: Incrementa el valor de una variable en 1.

```
int contador = 0;
contador++;  // contador es ahora 1
```

- `--`: Decrementa el valor de una variable en 1.

```
int valor = 10;
valor--;  // valor es ahora 9
```

Operadores de Miembro (Punteros):

- `->`: Accede a los miembros de una estructura o clase a través de un puntero.

```
struct Persona {
    int edad;
};
Persona persona;
Persona *pPersona = &persona;
pPersona->edad = 25;
```

Operadores de Ternario:

- `condición ? valor_verdadero : valor_falso`: Evalúa la condición y devuelve `valor_verdadero` si es verdadera, o `valor_falso` si es falsa.

```
int resultado = (5 > 3) ? 10 : 20;  // resultado es 10
```

Otros operadores

1. **Operador de Ámbito `::`:**

   El operador de ámbito `::` se utiliza para acceder a miembros de una clase o namespace. Se utiliza para especificar a qué clase o namespace pertenece un miembro particular.

```cpp
namespace MiNamespace {
    int numero = 42;
}

int main() {
    int numero = 10;
    std::cout << MiNamespace::numero << std::endl;  // Acceso al miembro 'numero' del namespace
    return 0;
}
```

2. **Operador de Dirección `&`:**

   El operador de dirección `&` se utiliza para obtener la dirección de memoria de una variable.

```cpp
int valor = 50;
int *puntero = &valor;  // 'puntero' almacena la dirección de 'valor'
```

3. **Operador de Desreferenciación `*`:**

   El operador de desreferenciación `*` se utiliza para acceder al valor al que apunta un puntero.

```cpp
int valor = 30;
int *puntero = &valor;
int valorPuntero = *puntero;  // 'valorPuntero' contiene 30
```

4. **Operador de Tamaño `sizeof`:**

   El operador de tamaño `sizeof` se utiliza para obtener el tamaño en bytes de un tipo de dato o una variable.

```cpp
int entero = 10;
std::cout << "Tamaño de int: " << sizeof(int) << " bytes" << std::endl;  // Tamaño de int: 4 bytes
```

5. **Operador de Casting (Conversiones):**

   Los operadores de casting permiten convertir un tipo de dato en otro.

   - Conversión de tipo estático:

```cpp
double numeroDouble = 3.14;
int numeroEntero = static_cast<int>(numeroDouble);  // Convierte 'numeroDouble' a int
```

   - Conversión dinámica (polimorfismo):

```
Base *ptrBase = new Derivada();
Derivada *ptrDerivada = dynamic_cast<Derivada*>(ptrBase);  // Convierte 'ptrBase' a
'ptrDerivada'
```

Operador de Inserción `<<`

El operador de inserción `<<` se utiliza para enviar datos a un flujo de salida, como la consola o un archivo. Es comúnmente usado con objetos de flujo de salida, como `std::cout`, para mostrar información en la pantalla.

```
#include <iostream>

int main() {
    int numero = 42;
    double decimal = 3.14;
    std::string mensaje = "Hola, mundo!";

    std::cout << "Número: " << numero << std::endl;  // Imprime "Número: 42"
    std::cout << "Decimal: " << decimal << std::endl;  // Imprime "Decimal: 3.14"
    std::cout << "Mensaje: " << mensaje << std::endl;  // Imprime "Mensaje: Hola, mundo!"

    return 0;
}
```

En este ejemplo, utilizamos el operador de inserción `<<` junto con el objeto de flujo de salida `std::cout` para enviar datos a la consola. La secuencia de caracteres y los valores de las variables se muestran en la pantalla.

> El operador de inserción `<<` en C++ puede entenderse como una forma de concatenar o combinar diferentes elementos en una cadena de salida. Aunque no es exactamente igual a la concatenación en el sentido tradicional, tiene un efecto similar cuando se trata de imprimir datos en la salida estándar.
> En lugar de unir cadenas de caracteres directamente, como se hace con la concatenación en otros lenguajes, el operador `<<` se utiliza para enviar diferentes tipos de datos (números, cadenas, etc.) a un flujo de salida para que se muestren en pantalla en el orden especificado.

Impresión en consola en c++

Flujo de Salida `std::cout` para Impresión en Consola:

`std::cout` es un objeto de flujo de salida estándar que se utiliza para mostrar información en la consola.

Sintaxis:

```
#include <iostream>

int main() {
    // Imprimir en la consola
    std::cout << "Texto a imprimir" << std::endl;
}
```

Explicación:

- `#include <iostream>`: Esto incluye la biblioteca estándar de entrada y salida (`iostream`) para usar las funciones de entrada y salida.
- `std::cout`: Es el objeto de flujo de salida estándar. Utilizamos `std::cout` seguido del operador `<<` para enviar datos al flujo de salida.
- `<<`: El operador de inserción se utiliza para enviar datos al flujo de salida. Puedes concatenar varios valores con este operador.
- `std::endl`: Este es el manipulador de flujo que realiza un salto de línea en la salida. También vacía el búfer, lo que significa que los datos se mostrarán inmediatamente en la consola en lugar de quedar en espera.

Ejemplo:

```
#include <iostream>

int main() {
    int edad = 25;
    double altura = 1.75;

    std::cout << "Mi edad es: " << edad << " años" << std::endl;
    // Imprime: Mi edad es: 25 años
    std::cout << "Mi altura es: " << altura << " metros" << std::endl;
        // Imprime: Mi altura es: 1.75
    return 0;
}
```

Entrada en consola en c++

La entrada de datos en C++ se realiza utilizando el flujo de entrada `std::cin`, que permite leer valores desde el teclado o desde otros flujos de entrada.

Sintaxis:

```
#include <iostream>

int main() {
```

```
        // Variable de entrada
    tipo_dato variable;
        // Mensaje al ususario
    std::cout << "Ingrese un valor: ";
    // Sentencia que asigna el valor recibido en la variable
    std::cin >> variable;
}
```

Explicación:

- `#include <iostream>`: Como en el caso de la salida, incluimos la biblioteca estándar de entrada y salida (`iostream`) para utilizar las funciones de entrada y salida.
- `std::cin`: Es el objeto de flujo de entrada estándar. Utilizamos `std::cin` seguido del operador `>>` para leer valores desde la entrada.
- `tipo_dato variable;`: Declaración de una variable para almacenar el valor que se leerá.
- `std::cout << "Ingrese un valor: ";`: Muestra un mensaje para indicar al usuario que debe ingresar un valor.
- `std::cin >> variable;`: Lee el valor ingresado por el usuario y lo almacena en la variable declarada.

Ejemplo:

```
#include <iostream>

int main() {
    int edad;

    std::cout << "Ingrese su edad: ";
    std::cin >> edad;

    std::cout << "Su edad es: " << edad << " años" << std::endl;

    return 0;
}
```

En este ejemplo, usamos `std::cin` para leer el valor de la edad ingresado por el usuario desde la consola y lo almacenamos en la variable `edad`. Luego, usamos `std::cout` para mostrar la edad ingresada.

Lectura de caracter

`std::cin.get()` es una función en C++ que se utiliza para leer un carácter individual desde la entrada estándar (generalmente el teclado) y devolver ese carácter como resultado. También es útil para pausar el programa y esperar a que el usuario presione la tecla Enter.

```cpp
#include <iostream>

int main() {
    std::cout << "Presiona Enter para continuar...";
    std::cin.get();  // Esperar a que el usuario presione Enter

    std::cout << "Continuando con el programa..." << std::endl;

    return 0;
}
```

3. Ciclos repetitivos

Índice

TARGET DECK: Materia:: Parcial I::

Ciclos repetitivos

---

1. Ciclo `for`:

El ciclo `for` se utiliza para repetir un bloque de código un número específico de veces.

Estructura y Sintaxis:

```
for (inicialización; condición; incremento) {
    // Código a repetir
}
```

Ejemplo:

```
for (int i = 0; i < 5; i++) {
    std::cout << "Iteración " << i << std::endl;
}
```

---

2. Ciclo `while`:

El ciclo `while` se utiliza para repetir un bloque de código mientras una condición sea verdadera.

Estructura y Sintaxis:

```
while (condición) {
    // Código a repetir
}
```

Ejemplo:

```
int contador = 0;
while (contador < 3) {
    std::cout << "Contador: " << contador << std::endl;
    contador++;
}
```

---

3. Ciclo `do-while`:

El ciclo `do-while` es similar al ciclo `while`, pero garantiza que el bloque de código se ejecute al menos una vez, incluso si la condición es falsa.

Estructura y Sintaxis:

```
do {
    // Código a repetir
} while (condición);
```

Ejemplo:

```
int numero = 10;
do {
    std::cout << "Número: " << numero << std::endl;
    numero--;
} while (numero > 0);
```

---

4. Ciclo `for-each` (o `for-in`):

El ciclo `for-each` se utiliza para iterar a través de los elementos de un contenedor, como un arreglo o un contenedor de la biblioteca estándar.

Estructura y Sintaxis:

```
for (tipo elemento : contenedor) {
    // Código a ejecutar para cada elemento
}
```

Ejemplo:

```
#include <iostream>

int main() {
    int numeros[5] = {10, 20, 30, 40, 50};

    for (int num : numeros) {
```

```
        std::cout << "Número: " << num << std::endl;
    }

    return 0;
}
```

4. Estructuras condicionales

Índice

TARGET DECK: Materia:: Parcial I::

Estructuras condicionales

1. Estructura de Control `if`:

La estructura `if` se utiliza para ejecutar un bloque de código si una condición es verdadera.

Sintaxis:

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

Ejemplo:

```
int edad = 20;
if (edad >= 18) {
    std::cout << "Eres mayor de edad." << std::endl;
} else {
    std::cout << "Eres menor de edad." << std::endl;
}
```

2. Estructura de Control `if-else if-else`:

La estructura `if-else if-else` se utiliza para evaluar múltiples condiciones en orden y ejecutar el bloque de código correspondiente a la primera condición verdadera.

Sintaxis:

```
if (condición1) {
    // Código si condición1 es verdadera
} else if (condición2) {
    // Código si condición2 es verdadera
} else {
    // Código si ninguna condición es verdadera
}
```

Ejemplo:

```
int puntaje = 85;
if (puntaje >= 90) {
    std::cout << "Calificación: A" << std::endl;
} else if (puntaje >= 80) {
    std::cout << "Calificación: B" << std::endl;
} else if (puntaje >= 70) {
    std::cout << "Calificación: C" << std::endl;
} else {
    std::cout << "Calificación: D" << std::endl;
}
```

3. Estructura de Control `switch`:

La estructura `switch` se utiliza para realizar una selección entre diferentes casos según el valor de una expresión.

Sintaxis:

```
switch (expresión) {
    case valor1:
        // Código si expresión es igual a valor1
        break;
    case valor2:
        // Código si expresión es igual a valor2
        break;
    // ...
    default:
        // Código si ninguno de los casos coincide
}
```

Ejemplo:

```
char operador = '+';
int a = 5, b = 3;
switch (operador) {
    case '+':
        std::cout << a + b << std::endl;
        break;
    case '-':
        std::cout << a - b << std::endl;
        break;
    case '*':
        std::cout << a * b << std::endl;
        break;
    case '/':
        std::cout << a / b << std::endl;
        break;
    default:
```

```
        std::cout << "Operador no válido" << std::endl;
}
```