# OPERACIONES

**CADA APARTADO PRINCIPAL CORRESPONDE CON EL NOMBRE DEL PROYECTO CON EJEMPLOS DE LO APUNTADO AQUÍ**

## 1. OPERACIONES BÁSICAS

- Sumar
- Restar
- Multiplicar
- Dividir
- Calcular el módulo

## 2. PRECEDENCIA Y ASOCIATIVIDAD

- Son el conjunto de reglas que usamos para determinar el orden a la hora de operar cuando hay varios operadores diferentes.
- Precedencia: Que operación hacer primero.
- Asociatividad: En qué dirección se hace la operación.

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | :: | Scope resolution | Left-to-right → |
| 2 | a++ a--<br>*type*() *type*{}<br>a()<br>a[]<br>. -> | Suffix/postfix increment and decrement<br>Functional cast<br>Function call<br>Subscript<br>Member access | |
| 3 | ++a --a<br>+a -a<br>! ~<br>(*type*)<br>*a<br>&a<br>sizeof<br>co_await<br>new new[]<br>delete delete[] | Prefix increment and decrement<br>Unary plus and minus<br>Logical NOT and bitwise NOT<br>C-style cast<br>Indirection (dereference)<br>Address-of<br>Size-of[note 1]<br>await-expression (C++20)<br>Dynamic memory allocation<br>Dynamic memory deallocation | Right-to-left ← |
| 4 | .* ->* | Pointer-to-member | Left-to-right → |
| 5 | a*b a/b a%b | Multiplication, division, and remainder | |
| 6 | a+b a-b | Addition and subtraction | |
| 7 | << >> | Bitwise left shift and right shift | |
| 8 | <=> | Three-way comparison operator (since C++20) | |
| 9 | < <= > >= | For relational operators < and ≤ and > and ≥ respectively | |
| 10 | == != | For equality operators = and ≠ respectively | |
| 11 | a&b | Bitwise AND | |
| 12 | ^ | Bitwise XOR (exclusive or) | |
| 13 | \| | Bitwise OR (inclusive or) | |
| 14 | && | Logical AND | |
| 15 | \|\| | Logical OR | |
| 16 | a?b:c<br>throw<br>co_yield<br>=<br>+= -=<br>*= /= %=<br><<= >>=<br>&= ^= \|= | Ternary conditional[note 2]<br>throw operator<br>yield-expression (C++20)<br>Direct assignment (provided by default for C++ classes)<br>Compound assignment by sum and difference<br>Compound assignment by product, quotient, and remainder<br>Compound assignment by bitwise left shift and right shift<br>Compound assignment by bitwise AND, XOR, and OR | Right-to-left ← |
| 17 | , | Comma | Left-to-right → |

## 3. OPERADORES DE INCREMENTO Y DECREMENTO PREFIJOS Y POSTFIJOS

- Decir “incrementar una variable” significa aumentar el valor de la misma.
- Decir “decrementar una variable” significa disminuir el valor de la misma.
- Podemos usar ++ como prefijo o postfijo de la variable para aumentar su valor en 1.
- Podemos usar −− como prefijo o postfijo de una variable para disminuir su valor en 1.
- La diferencia entre el prefijo y el postfijo, es cuando se realiza el aumento o decremento de la variable, si es un prefijo se realizará antes de ejecutar toda la sentencia y si es un postfijo se realizará tras la ejecución de la sentencia.
- Hay ejemplos más explicativos en los proyectos.
- Estos prefijos y postfijos solo sirven para aumentar o para disminuir en 1 el valor de la variable. No existen estos operadores para multiplicar o dividir.

## 4. OPERADORES COMPUESTOS

- Permiten hacer operaciones de una forma más simplificada a la hora de escribir, pero su uso principal es el de asignar el nuevo valor de la variable directamente tras realizar la operación.
- Para usar operadores compuestos, ponemos el símbolo de la operación que queremos realizar (suma, resta, multiplicación, división y módulo) seguido de un igual y el valor por el que queremos operar con nuestra variable. Ejemplo:

```
int x = 5;
x += 6; //Es lo mismo que poner x = x + 6
std::cout << “x: ” << x << std::endl; // x = 11
```

## 5. OPERADORES RELACIONALES

- Permiten hacer operaciones comparativas.
- Las operaciones relaciones se evalúan como bools.

- Es importante tener en cuenta la precedencia de C++ al usar estos operadores, porque puede aparecer un error al compilar.
  Para evitar este error de compilación se usan paréntesis envolviendo la operación de comparación. En el proyecto lo explico más en detalle.
- Menor que: >
- Mayor que: <
- Menor o igual que: <=
- Mayor o igual que: >=
- Distinto que:!! =
- Igual que: ==

## 6. OPERADORES LÓGICOS

- Operadores que funcionan en operaciones con bools.
- AND: Se representa con "&&". Si cualquiera de los operandos es `false`, el resultado será `false`. Por lo tanto, el resultado será `true` solo si ambos operandos son `true`.
- OR: Se representa con "||". Si cualquiera de los operandos es `true`, el resultado será `true`. Por lo tanto, el resultado será `false` solo si ambos operandos son `false`.
- NOT: Se representa con "!" delante de la variable. Niega cualquier valor. Es decir, si `a = true`, entonces `!a = false`.
- Se pueden combinar varias operaciones lógicas en una única operación.
- Se pueden combinar operaciones lógicas y relaciones en una única operación, ya que las operaciones relacionales se evalúan como bools.

## 7. FORMATO A LA SALIDA DE DATOS CON MANIPULADORES

- Para dar formato a la salida por la terminal de datos mediante `std::cout` usamos lo que se denominan manipuladores. Estos están principalmente incluidos en 2 librerías :
  - `<ios>`. Viene incluido en `<iostream>` que en total incluye: `<ios>`, `<streambuf>`, `<istream>`, and `<ostream>`

- `<iomanip>`

- Los manipuladores son los siguientes:
  - `std::endl`: Requiere de `<ostream>`. Incluye un salto de línea al final. También podemos usar "`\n`" dentro de la cadena texto para incluir el salto de línea. `std::endl` y `\n` son cosas diferentes, pero hacen lo mismo.
  - `std::flush`: Requiere de `<ostream>`. Envía inmediatamente los datos que se encuentren en el buffer de salida al terminal. Es decir al ejecutar la sentencia los datos van a pasar antes por un buffer intermedio (como una especie de almacén al que van los datos antes de ir al terminal) y una vez se llene, los datos se enviarán todos de 1 vez al terminal.
  - `std::setw`: Requiere de `<iomanip>`. Especifica un ancho de caracteres para el texto que quieras imprimir. Digamos que sirve para crear "celdas" de caracteres para mostrar el texto de una forma más bonita.
  - `std::right`: Requiere de `<ios>`. Justificar el texto a la derecha.
  - `std::left`: Requiere de `<ios>`. Justifica el texto a la izquierda.
  - `std::internal`: Requiere de `<ios>`. Justifica el texto de forma "interna" si queremos mostrar el texto en forma de tabla con los signos de los valores justificados a la izquierda y los propios valores a la derecha.
  - `std::setfill()`: Requiere de `<iomanip>`. Rellena los espacios con el caracter indicado.
  - `std::boolalpha`: Requiere de `<ios>`. Muestra los valores de los bools como "true" y "false" en vez de como "1" y "0".
  - `std::noboolalpha`: Requiere de `<ios>`. Muestra los valores de los bools como "1" y "0" en vez de como "true" y "false".
  - `std::showpos`: Requiere de `<ios>`. Muestra los signos de números positivos.
  - `std::noshowpos`: Requiere de `<ios>`. Oculta los signos de números positivos.
  - `std::dec`: Requiere de `<ios>`. Muestra los datos en base decimal. Si el dato es un número decimal, no tendrá efecto ya que estos números se representan en memoria siguiendo otro sistema numérico distinto más avanzado, el IEEE_754.

- `std::oct`: Requiere de `<ios>`. Muestra los datos en base octal. Si el dato es un número decimal, no tendrá efecto ya que estos números se representan en memoria siguiendo otro sistema numérico distinto más avanzado, el IEEE_754.
- `std::hex`: Requiere de `<ios>`. Muestra los datos en base hexadecimal. Si el dato es un número decimal, no tendrá efecto ya que estos números se representan en memoria siguiendo otro sistema numérico distinto más avanzado, el IEEE_754.
- `std::showbase`: Requiere de `<ios>`. Muestra la base delante de los números del sistema numérico empleado.
- `std::noshowbase`: Requiere de `<ios>`. Oculta la base delante de los números del sistema numérico empleado.
- `std::uppercase`: Requiere de `<ios>`. Muestra los datos en mayúscula.
- `std::nouppercase`: Requiere de `<ios>`. Deja de mostrar los datos en mayúscula.
- `std::scientific`: Requiere de `<ios>`. Muestra los datos en notación científica.
- `std::fixed`: Requiere de `<ios>`. Muestra los datos en notación fija.
  En caso de que quiera mostrar los números decimales en su formato por defecto de nuevo no hay una forma concreta de hacerlo, pero en el curso se ha dado este truco para conseguirlo. Es avanzado pero funciona:
  - `std::cout.unsetf(std::ios::scientific | std::ios::fixed);`
- `std::setprecision`: Requiere de `<iomanip>`. Específica la precisión con la que los datos se muestran por terminal. La precisión es la cantidad de caracteres de un número. La precisión por defecto es 6.
- `std::showpoint`: Requiere de `<ios>`. Fuerza la muestra de decimales acorde con la precisión.
- `std::noshowpoint`: Requiere de `<ios>`. Oculta la muestra de decimales.

## 8. LÍMITES NUMÉRICOS

- Vamos a usar funciones de la librería: `<limits>`

- Estas funciones son:
  - `std::numeric_limits<T>::min()`: La "T" actúa de *placeholder* del tipo de dato del cual queremos saber su mínimo.
    En números decimales, el mínimo es el número positivo más pequeño que se puede representar.
    Si el `integer` es signed, el mínimo es el número negativo más pequeño que se puede representar, mientras que si es unsigned, el mínimo será 0.
  - `std::numeric_limits<T>::max()`: La "T" actúa de *placeholder* del tipo de dato del cual queremos saber su máximo.
    En números decimales, el máximo es el número positivo más grande que se puede representar.
    En `integers`, el máximo es el número positivo más grande que se puede representar.
  - `std::numeric_limits<T>::lowest()`: La "T" actúa de *placeholder* del tipo de dato del cual queremos saber su punto más bajo.
    En números decimales el punto más bajo es el número negativo más pequeño que se puede representar.
    En `integers` no se puede aplicar esta función.

## 9. FUNCIONES MATEMÁTICAS

- Funciones matemáticas útiles para hacer ciertas operaciones incluidas en la Standard Library de C++
- Pertenecen a la librería `<cmath>` y son:
  - `std::floor()`: Redondea hacia abajo.
  - `std::ceil()`: Redondea hacia arriba.
  - `std::round()`: Redondea en base al decimal. Si el decimal es 0.5 redondea hacia arriba.
  - `std::abs()`: Obtiene el valor absoluto.
  - `std::cos()`: Calcula el coseno.
  - `std::sin()`: Calcula el seno.
  - `std::tan()`: Calcula la tangente.

- `std::exp()`: Calcula el exponencial. Es decir $f(x) = e^x$
- `std::log()`: Calcula el logaritmo.
- `std::pow()`: Calcula la potencia.
- `std::sqrt()`: Calcula la raíz cuadrada.

## 10. OTROS TIPOS DE INTEGER

- `Integers` que ocupen menos de 4 bytes en memoria no soportan operaciones aritméticas. Algunos de ellos son:
  - `char`: Ocupa 1 byte en memoria.
  - `short int`: Ocupa 2 bytes en memoria.
- Esto se debe al diseño del procesador, ya que se decidió que el `integer` más pequeño para hacer operaciones aritméticas fuera el `int`.
- Los compiladores convertirán implícitamente estos tipos a int si queremos hacer operaciones con ellos.
- Es importante estar atento a este tipo de cosas, ya que muchas veces la lógica de tu programa va a depender del tamaño que se ocupa en memoria.

## 11. RESUMEN

- Suma, resta, multiplicación, división, módulo.
- Precedencia y asociatividad.
- Operadores de incremento y decremento prefijos y postfijos, operadores compuestos, operadores relacionales, operadores lógicos.
- `<ostream>`, `<ios>`, `<iomanip>`, `<limits>`, `<cmath>`
- No puedes realizar operaciones aritméticas con tipos de dato que ocupen menos de 4 bytes en memoria.