**OPERADORES EN C++**

**Operadores aritméticos:**

Estos nos permiten realizar cálculos de números reales y aritmética de punteros, tales como suma, resta, multiplicación, división, etc. En C se utilizan los siguientes operadores aritméticos.

* + : suma binaria, más unitario.
* ++ : Incremento
* - : resta binaria, menos unitario.
* -- : Decremento
* \* : Multiplicación
* / : División
* %: Resto o Modulo.

**Operadores de asignación:**

Hay dos clases de operandos de asignación: asignación simple, en la que él valor del segundo operando se almacena en el objeto especificado por el primer operando y asignación compuesta, en la que se realiza una operación aritmética, de desplazamiento o bit a bit antes de almacenar el resultado.

|  |  |
| --- | --- |
| **=** | Almacena el valor del segundo operando en el objeto especificado por el primer operando (asignación simple). |
| **\*=** | Multiplica el valor del primer operando por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando. |
| **/=** | Divide el valor del primer operando por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando. |
| **%=** | Toma el módulo del primer operando especificado por el valor del segundo operando; almacena el resultado en el objeto especificado por el primer operando. |
| **+=** | Suma el valor del segundo operando al valor del primer operando; almacena el resultado en el objeto especificado por el primer operando. |
| **–=** | Resta el valor del segundo operando del valor del primer operando; almacena el resultado en el objeto especificado por el primer operando. |

**Operadores de relación:**

Se utilizan principalmente en las expresiones condicionales, con el fin de establecer relación entre dos o más variables.

>: Mayor >=: Mayor igual ==: Igual

<: Menor <=: Menor igual !=: Distinto

**Operadores Lógicos:**

* El operador "&&" equivale al "AND" o "Y"; devuelve **true** sólo si los dos operandos **true** o lo que es equivalente, distintas de cero. En cualquier otro caso el resultado es **false**.
* El operador "||" equivale al "OR" u "O inclusivo"; devuelve **true** si cualquiera de las expresiones evaluadas es **true**, o distinta de cero, en caso contrario devuelve **false**.
* El operador "!" es equivalente al "NOT", o "NO", y devuelve **true** cuando la expresión evaluada es **false** o cero, en caso contrario devuelve **false**.

**{}** Las llaves nos sirven para separar secciones o sentencias dentro del cuerpo de un programa, dando proporcionando un mejor orden de cómo se ejecutaran.

**()** Los paréntesis sirven para identar una expresión o una llamada a función.

**PALABRAS RESERVADAS:**

**RETURN:** Finaliza la ejecución de una función y devuelve el control a la función de llamada (o al sistema operativo si se transfiere el control de la función**main**).La ejecución se reanuda en la función de llamada, en el punto que sigue inmediatamente a la llamada.

**BREAK:** La instrucción **break** finaliza la ejecución del bucle o la instrucción condicional envolvente más próximo en el que aparece. El control pasa a la instrucción que hay a continuación del final de la instrucción, si hay alguna.

**IF**: La sentencia if sirve para ejecutar una serie de instrucciones si se cumple una determinada condición.

**IF ELSE:** La sintaxis de un condicional if-else, es en principio similar a la del condicional if, pero adicionando una nueva "estructura" que es el else, el cual indica la acción o conjunto de acciones a llevar a cabo, en caso de que la condición del if no se cumpla.

**SWITCH:** Permite la selección entre varias secciones de código, dependiendo del valor de una expresión entera.

**CASE:** Esta etiqueta se circunscribe al ámbito de la sentencia **switch**, La estructura condicional **switch-case** se utiliza cuando queremos evitarnos las llamadas escaleras de decisiones. La estructura if nos puede proporcionar, únicamente, dos resultados, uno para verdadero y otro para falso. Una estructura **switch-case**, por su parte, nos permite elegir entre muchas opciones.

**WHILE:** En este ciclo el cuerpo de instrucciones se ejecuta mientras una condición permanezca como verdadera en el momento en que la condición se convierte en falsa el ciclo termina. Su formato general es:

Cargar o inicializar variable de condición;

while (condición)

{

Grupo cierto de instrucciones;

Instrucción(es) para salir del ciclo;

}

**DO WHILE:** Ejecuta un elemento statement repetidamente veces hasta que la condición de finalización (expresión) se evalúa como cero:

* Se ejecuta el cuerpo de instrucción.
* A continuación, se evalúa expresión. Si expresión es false, la instrucción do-while finaliza y el control pasa a la siguiente instrucción del programa. Si expresión es true (distinta de cero), el proceso se repite a partir del paso 1.

int main()

{

int i = 0;

do

{

printf\_s("\n%d",i++);

} while (i < 3);

}

**FOR:** La sentencia for permite la construcción de una forma compacta de los ciclos controlados por contador, aumentando la legibilidad del código. Su sintaxis es la siguiente:

For (i=valor inicial; i<=valor final; i++){

Sentencias

}

**PALABRAS RESERVADAS DE TIPOS**

**INT:** Almacena datos de tipo entero dentro de los límites de cada uno de sus tamaños. A su vez, esos tamaños dependen de la plataforma, del compilador, y del número de bits que use por palabra de memoria: 8, 16, 32... No hay reglas fijas para saber el tamaño, y por lo tanto, el mayor número que podemos almacenar en cada tipo entero: short int, int o long int; depende en gran medida del compilador y del sistema operativo.

**CHAR:** Almacena datos de tipo carácter, es el tipo básico alfanumérico, es decir que puede contener un carácter, un dígito numérico o un signo de puntuación, El tamaño de memoria es de 1 byte.

**FLOAT:** Las variables de este tipo almacenan números en formato de coma decimal es decir números decimales, este formato nos permite almacenar números entre -0.99 x 109 y 0.99 x 109. Es decir, entre -990000000 y 99000000.

**DOUBLE:** Sirve para declarar una variable de tipo decimal pero más precisa, ya que acepta hasta 15 factores decimales.

**VOID:** Cuando se utiliza como un tipo de valor devuelto de función, la palabra clave void especifica que la función no devuelve ningún valor. Cuando se utiliza para la lista de parámetros de una función, void especifica que la función no toma ningún parámetro. Cuándo se utiliza en la declaración de un puntero, void especifica que el puntero es "universal". Si el tipo de puntero es void \*, el puntero puede señalar a cualquier variable que no se declare con la palabra clave const o volatile. Un puntero void no se puede des referenciar a menos que se convierta en otro tipo. Un puntero void se puede convertir en cualquier otro tipo de puntero de datos.

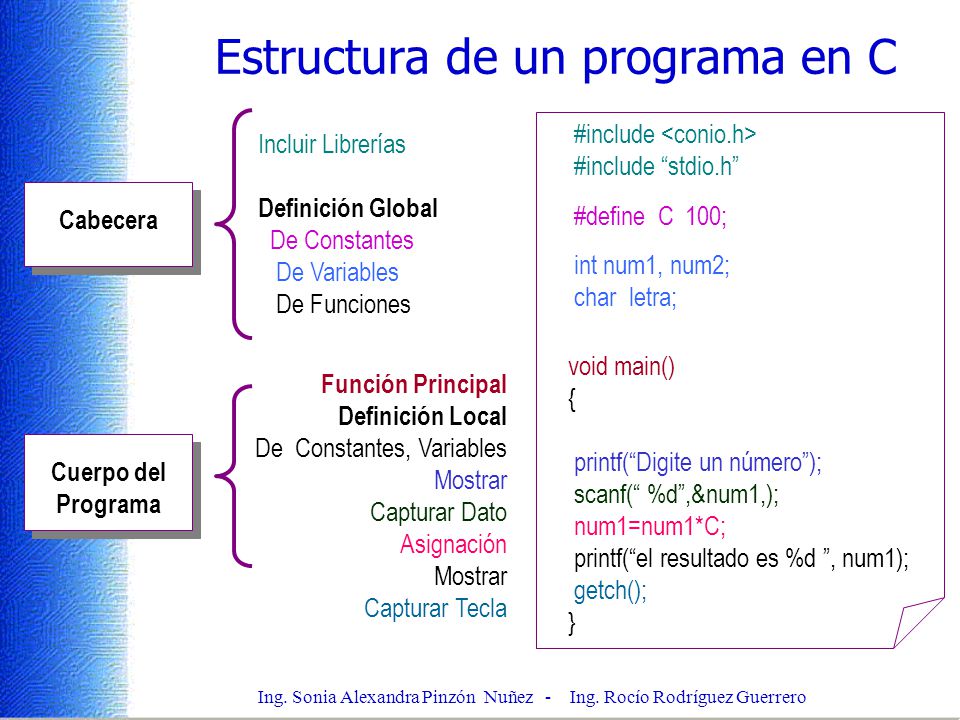
**CONST:** La palabra clave const se usa para declarar un campo constante o una local constante. Los campos locales y constantes no son variables y no se pueden modificar. Las constantes pueden ser números, valores booleanos, cadenas o una referencia nula.

**VOLATILE:** La palabra clave volatile indica que varios subprocesos que se ejecutan a la vez podrían modificar un campo. Los campos que se declaran como volatile no están sujetos a optimizaciones del compilador que suponen el acceso por un subproceso único. Esto garantiza que el valor más actualizado está en todo momento presente en el campo. La palabra clave volatile sólo se puede aplicar a los campos de una clase o struct. Las variables locales no se pueden declarar como **volatile**.

**UNSIGNED:** Significa `sin signo'... es como la diferencia que hay entre los números naturales (0...) o los números enteros (...-1, 0,1,...), Una variable que se declare unsigned siempre empleará valores enteros positivos, nunca negativos.

**SIGNED:** Le indica al compilador que el valor utilizado en la variable int o char, que los valores serán positivos y negativos.

**PARTES DE UN PROGRAMA EN C:**



**LLAMADO RECURSIVO:** Se dice que una función es recursiva cuando se define en función de si misma, C++ permite la recursividad. Cada vez que se llama a una función, se crea un juego de variables locales, de este modo, si la función hace una llamada a sí misma, se guardan sus variables y parámetros, usando la pila, y la nueva instancia de la función trabajará con su propia copia de las variables locales. Cuando esta segunda instancia de la función retorna, recupera las variables y los parámetros de la pila y continúa la ejecución en el punto en que había sido llamada.

**LLAMADO ITERATIVO:** La iteración utiliza una estructura repetitiva y la recursión una estructura de selección. La iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetitivas a funciones.La iteración termina si la condición del bucle no se cumple, mientras que la recursión termina cuando se reconoce un caso base.