Operadores con Bits

C++ dispone de 6 operadores para manejo de bits que realizan dos tipos de operaciones.

Son los siguientes:

~	Complemento a uno
<<	Desplazamiento a izquierda
>>	Desplazamiento a derecha
&	AND; compara dos bits
^	XOR (OR exclusivo); compara dos bits
|	OR inclusivo; compara dos bits

- El primero es un operador unario, los restantes son binarios.
- Los tres primeros realizan manipulaciones en los bits del operando.
- Los restantes realizan comparaciones lógicas entre los bits de ambos operandos, similares a las que realizan los operadores lógicos entre objetos booleanos.

~ Complemento a uno

El operador de complemento de uno (~), denominado a veces operador de complemento bit a bit, produce un complemento de uno bit a bit de su operando. Es decir, cada bit que es 1 en el operando es 0 en el resultado. Y a la inversa: cada bit que es 0 en el operando es 1 en el resultado. El operando del operador de complemento de uno debe ser de tipo entero.

Sintaxis:	~ cast-expression

Ejemplo:

```
// operator_complement_one.cpp
#include <iostream>

using namespace std;

int main () {
   unsigned short y = 0xFFFF;
   cout << hex << y << endl;
   y = ~y;   // Toma el complemento a uno
   cout << hex << y << endl;
}
```

En este ejemplo, el nuevo valor asignado a y es el complemento de uno del valor sin signo 0xFFFF, or 0x0000.

<< Desplazamiento a izquierda

Este operador binario realiza un desplazamiento de bits a la izquierda. El bit más significativo (más a la izquierda) se pierde, y se le asigna un 0 al menos significativo (el de la derecha). El operando derecho indica el número de desplazamientos que se realizarán.

Recuérdese que los desplazamientos no son rotaciones; los bits que salen por la izquierda se pierden, los que entran por la derecha se rellenan con ceros. Este tipo de desplazamientos se denominan lógicos en contraposición a los cíclicos o rotacionales.

Sintaxis: expr-desplazada << expr-desplazamiento

Ejemplo:

```
// operator_shift_left.cpp
#include <iostream>

short signed cero = 0, uno = 1, dos = 2;

int main () {
  cout << "0 << 1 == " << (cero << 1) << endl;
  cout << "1 << 1 == " << (uno << 1) << endl;
  cout << "2 << 1 == " << (dos << 1) << endl;
}
```

0 == 0000 0000	=>	0 << 1 == 0000 0000	==	0
1 == 0000 0001	=>	1 << 1 == 0000 0010	==	2
2 == 0000 0010	=>	2 << 1 == 0000 0100	==	4
-3 == 1111 1101	=>	-3 << 1 == 1111 1010	==	-6

<< Desplazamiento a la derecha

El operador de desplazamiento a la derecha hace que el patrón de bits de expr-desplazada se desplace hacia la derecha el número de posiciones especificado por expr-desplazamiento. En el caso de números sin signo, las posiciones de bits que quedan vacantes debido a la operación de desplazamiento se rellenan con ceros. Si se trata de números con signo, el bit de signo se emplea para rellenar las posiciones de bits vacantes. Es decir, si el número es positivo se usa 0 y si el número es negativo se usa 1.

Sintaxis: expr-desplazada >> expr-desplazamiento

0 == 0000 0000	=>	0 >> 1 == 0000 0000	==	0
2 == 0000 0010	=>	2 >> 1 == 0000 0001	==	1
-2 == 1111 1110	=>	-2 >> 1 == 1111 1111	==	-1
-16 == 1111 0000	=>	-6 >> 2 == 1111 1100	==	-4

Ejemplo:

```
// operator_shift_right.cpp
#include <iostream>

short signed cero = 0, dos = 2, mdos = -2;

int main () {
  cout << "0 >> 1 == " << (cero >> 1) << endl;
  cout << "2 >> 1 == " << (dos >> 1) << endl;
  cout << "-2 >> 1 == " << (mdos >> 1) << endl;
}
```

& AND

El operador AND bit a bit (&) compara cada bit del primer operando con el bit correspondiente del segundo operando. Si ambos bits son 1, el bit del resultado correspondiente se establece en 1. De lo contrario, el bit del resultado correspondiente se establece en 0.

Sintaxis: AND-expresion & equality-expresion

Ejemplo:

Según las reglas del enunciado, el operador & aplicado entre los valores 2 y -2 resultaría:

```
 2  == 0000 0010
-2  == 1111 1110
       --------------
       0000 0010  == 2
```

Comprobación:

```
// and_logico_bit.cpp
#include <iostream>

using namespace std;

    2 & -2 == 2
```

```
Este operador se utiliza para verificar el estado de
los bits individuales de un número, al que llamaremos
incognita, mediante comparación con un patrón cuya
disposición de bits es conocida. La expresión puede
ser:
    if (incognita & patron) {
            /* concordancia con el patrón */
    } else {
            /* desacuerdo con el patrón */
    }
```

^ **XOR** (OR exclusivo)

El operador OR exclusivo bit a bit (^) compara cada bit de su primer operando con el bit correspondiente de su segundo operando. Si el bit de uno de los operandos es 0 y el bit del otro operando es 1, el bit del resultado correspondiente se establece en 1. De lo contrario, el bit del resultado correspondiente se establece en 0.

Sintaxis: expr-OR-exclusiva ^ AND-expresion

Ejemplo:

Según el enunciado, el operador ^ aplicado entre los valores 7 y -2 resultaría:

```
 7  ==  0000 0111
-2  ==  1111 1110
       --------------
        1111 1001   ==  -7
```

Comprobación:

```
// or_exclusivo_bit.cpp
#include <iostream>

using namespace std;

int main () {
   cout << "7 ^ -2 == " << (7 ^ -2) << endl;
}
```

Salida:

```
7 & -2 == -7
```

| **OR** (inclusivo)

Este operador binario tiene un funcionamiento parecido a los anteriores (AND y XOR), salvo que en este caso el resultado es 1 si alguno de ellos está a 1.

Sintaxis: expr-OR-inclusiva | expr-OR-exclusiva

Ejemplo:

Según el enunciado, el operador | aplicado entre los valores 6 y 13 resultaría:

```
 6  ==  0000 0110
13  ==  0000 1101
       ---------------
       0000 1111   ==  15
```

Comprobación:

```
// or_exclusivo_bit.cpp
#include <iostream>

using namespace std;

int main () {
   cout << "6 | 13 == " << (6 | 13) << endl;
}
```

Salida:

```
6 | 13 == 15
```

Problemas Propuestos.

1) Realizar 2 ejemplos demostrativos donde utilice 1 ó varios Operador de bits, incorporar lo siguiente:
2) Debe mencionar brevemente el contexto del ejemplo. Comentar la linea donde se ha usado el operador.
3) Mostrar la salida de ejecucion del programa (una corrida).