

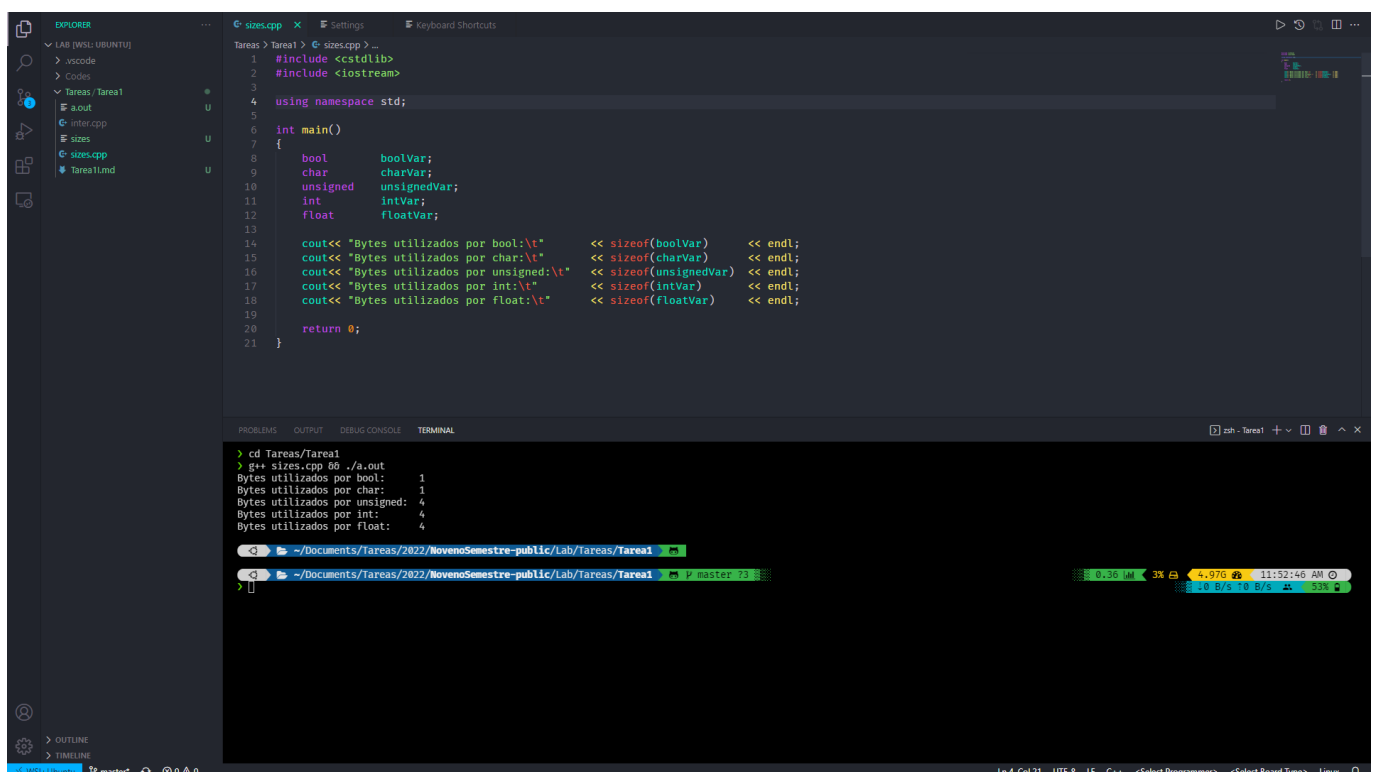
Tarea 1.I.

Tarea 1.I

- Compile, corra y analice el funcionamiento de los códigos:
 - sizes.cpp
 - inter.cpp
- Investigue la representación binaria con complemento a 2 para números enteros negativos. Calcule la representación a 32 bits de los siguientes números: -125, -4096, -1000000.

Compile, corra y analice el funcionamiento de:

sizes.cpp



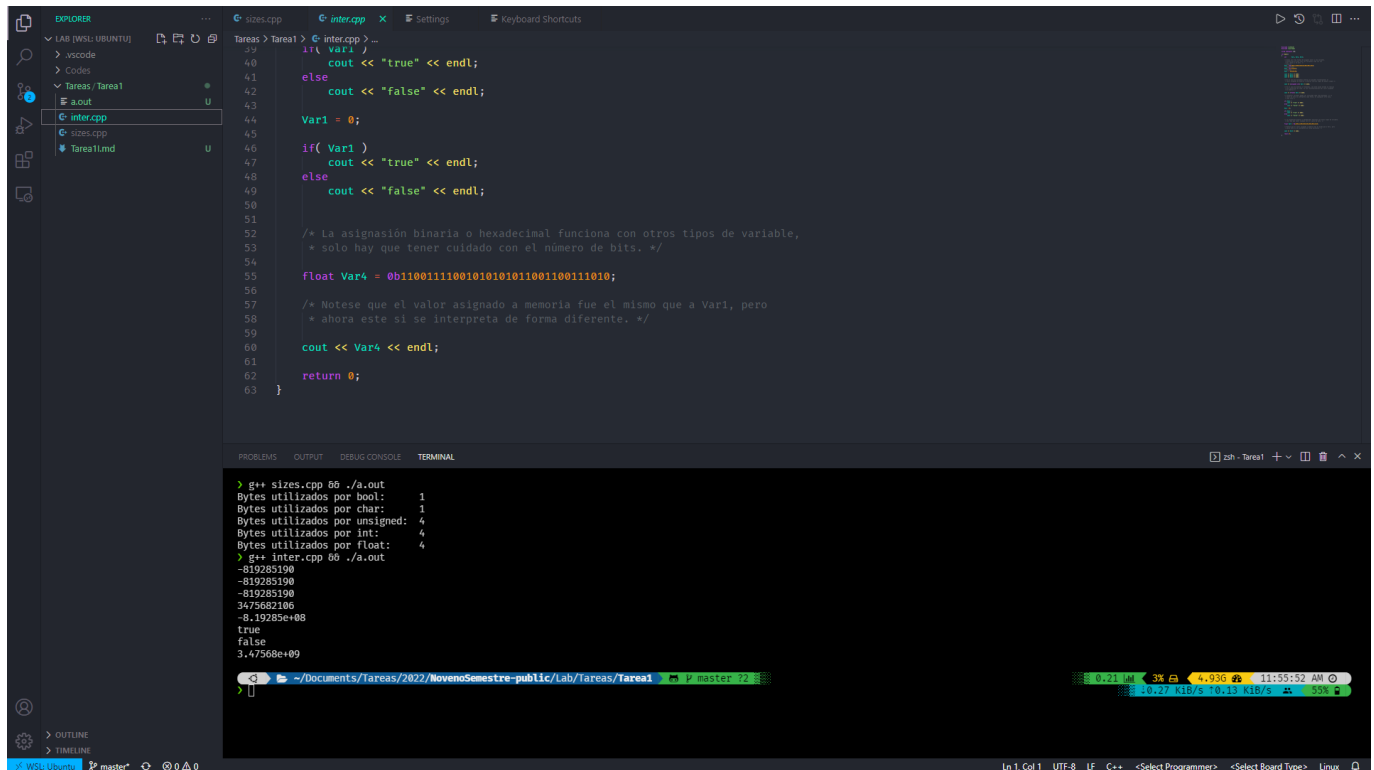
The screenshot shows the Visual Studio Code editor with the file `sizes.cpp` open. The code defines variables of different types and prints their sizes in bytes. The terminal output shows the results of running the program.

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     bool        boolVar;
9     char        charVar;
10    unsigned    unsignedVar;
11    int         intVar;
12    float       floatVar;
13
14    cout<< "Bytes utilizados por bool:\t"    << sizeof(boolVar)    << endl;
15    cout<< "Bytes utilizados por char:\t"    << sizeof(charVar)   << endl;
16    cout<< "Bytes utilizados por unsigned:\t" << sizeof(unsignedVar) << endl;
17    cout<< "Bytes utilizados por int:\t"     << sizeof(intVar)    << endl;
18    cout<< "Bytes utilizados por float:\t"   << sizeof(floatVar)   << endl;
19
20    return 0;
21 }
```

```
> cd Tareas/Tarea1
> g++ sizes.cpp 06 ./a.out
Bytes utilizados por bool: 1
Bytes utilizados por char: 1
Bytes utilizados por unsigned: 4
Bytes utilizados por int: 4
Bytes utilizados por float: 4
```

Como vimos en clase, este comando imprime los tamaños ocupados en memoria para cada tipo de variable.

inter.cpp



```
29 int vari;
40 cout << "true" << endl;
41 else
42 cout << "false" << endl;
43
44 Var1 = 0;
45
46 if( Var1 )
47 cout << "true" << endl;
48 else
49 cout << "false" << endl;
50
51
52 /* La asignación binaria o hexadecimal funciona con otros tipos de variable,
53 * solo hay que tener cuidado con el número de bits. */
54
55 float Var4 = 0b1100111100101010101100110011010;
56
57 /* Notese que el valor asignado a memoria fue el mismo que a Var1, pero
58 * ahora este si se interpreta de forma diferente. */
59
60 cout << Var4 << endl;
61
62 return 0;
63 }
```

```
> g++ sizes.cpp 06 ./a.out
Bytes utilizados por bool: 1
Bytes utilizados por char: 1
Bytes utilizados por unsigned: 4
Bytes utilizados por int: 4
Bytes utilizados por float: 4
> g++ inter.cpp 06 ./a.out
-819285190
-819285190
-819285190
3475682106
-8.19285e+08
true
false
3.47568e+09
```

En este código ocurren varias cosas:

- En las primeras 3 líneas:
 - El valor `-819285190` es asignado a una variable tipo entero de 3 formas distintas:
 - Notación binaria
 - Notación hexadecimal
 - Notación decimal
- En la línea 4, se imprime este mismo valor como variable tipo entero pero a su interpretación a complemento a 2.
- En la línea 5, se imprime el mismo valor pero en su notación de punto flotante.
- En la línea 6 y 7 se interpreta este mismo valor como tipo booleano, que se interpreta como verdadero para cualquier valor menos cero. Por lo que se imprime `true` al principio, y `false` después de asignarle a esta variable el valor cero.

- Finalmente se asigna el mismo valor en binario de esta variable a una variable tipo punto flotante, por lo que la interpretación cambia, ya que solo algunos datos del binario son tomados para el número en sí, y los otros son asignados la posición del punto decimal.

Representación binaria con complemento a 2 para números enteros negativos:

Una forma útil de entender el complemento a dos de un número (que se me acaba de ocurrir así que es posible que no esté bien) es tomar el complemento como el inverso aditivo del número original, pero esta vez, el elemento neutro no es cero, si no es 2^n en base 2 donde n es la cantidad de bits utilizada para calcular el complemento.

Por ejemplo a 3 bits tenemos:

$$3 + 5 = 8 = 2^3$$
$$011_2 + 101_2 = 1000_2$$

Por lo que 101_2 es el complemento a 2 de 011_2 y viceversa.

Entonces podemos hacer una tabla de los números para n bits y sus respectivos complementos:

Número	Complemento
000	1000
001	111
010	110
011	101
100	100
101	011
110	010
111	001

Es fácil demostrar que el complemento a 2 de cualquier número puede obtenerse al invertir los bits y sumarle 1:

Demostración:

Para un número A de n bits b_i tenemos:

$$A = b_n b_{n-1} \cdots b_3 b_2 b_1$$

Invertir los bits $b_i \rightarrow \bar{b}_i$ nos devuelve un número B :

$$B = \bar{b}_n \bar{b}_{n-1} \cdots \bar{b}_3 \bar{b}_2 \bar{b}_1$$

Nota: los subíndices indican la posición del byte, y comienzan a contarse de derecha a izquierda.

Puesto que b_i y \bar{b}_i son necesariamente bits opuestos, es decir, no pueden ser ambos 0 o ambos 1, entonces tenemos que $b_i + \bar{b}_i = 1$:

$$\begin{aligned} A + B &= b_n b_{n-1} \cdots b_3 b_2 b_1 + \bar{b}_n \bar{b}_{n-1} \cdots \bar{b}_3 \bar{b}_2 \bar{b}_1 \\ A + B &= 1_n 1_{n-1} \cdots 1_3 1_2 1_1 \end{aligned}$$

Finalmente, si sumamos 1 a esta expresión:

$$\begin{aligned} (A + B) + 1 &= 1_n \cdots 1_2 1_1 + 0_n 0_{n-1} \cdots 0_3 0_2 1_1 \\ (A + B) + 1 &= 1_{n+1} 0_n 0_{n-1} \cdots 0_3 0_2 0_1 \end{aligned}$$

Por asociatividad:

$$A + (B + 1) = 1_{n+1} 0_n \cdots 0_2 0_1 = 2^n$$

Definimos entonces $\bar{A} = B + 1$ como el complemento a 2 de A .

Ahora que entendemos que es el complemento a 2 de un número pasamos a definir con este los números negativos representados por bytes. Revisemos de nuevo la tabla, pero esta vez, escribiremos las representaciones enteras, y diremos que el complemento a 2 es el negativo del entero en cuestión.

Entero	Base 2	Complemento a 2	Entero negativo
0	000	1000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3
4	100	100	-4
5	101	011	-5
6	110	010	-6
7	111	001	-7

Esto tiene sentido, sin embargo hay un problema: Una misma representación binaria hace referencia a dos números enteros, uno positivo y uno negativo. Esto claramente no puede pasar. Pero la solución es sencilla, partimos la tabla a la mitad y aunque reducimos a la mitad la cantidad de enteros que se pueden representar, esta vez ya no habrá duplicados:

Entero	Base 2	Complemento a 2	Entero negativo
0	000	1000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3

Ahora todas las combinaciones de bits tienen solamente un posible valor entero. La única combinación que no lo tiene es la de 100, que por convención representa al -4 . Esta convención aplica en la generalización a más bits, ya que siempre la representación de en medio: $1_n 0_{n-1} \dots 0_2 0_1$ para n bits será siempre igual a su complemento a 2 además de ser la posición en donde la lista se corta por la mitad, por convención se toma como -2^{n-1} .

Representación a 32 bits de:

- -125:
 - 00110111 00110101 00111000 00110001 00111001 00111000 00111000 00110011
00110111
 - 37 35 38 31 39 38 38 33 37

- -4096:
 - 00110111 00110101 00111000 00110011 00111001 00110100 00111001 00110011
 00110111 00100000 00111001 00110000 00110101 00111001 00110110 00111001
 00110110 00110110 00110100
 - 37 35 38 33 39 34 39 33 37 20 39 30 35 39 36 39 36 36 34
- -1000000:
 - 00110111 00110101 00111000 00110001 00111001 00111000 00110011 00110010
 00110000 00100000 00111000 00110000 00111000 00110100 00110110 00110100
 00110100 00110011 00110010
 - 37 35 38 31 39 38 33 32 30 20 38 30 38 34 36 34 34 33 32