C++ Detalles ejercicios exercism

std::string::find (pangram)

Cuando `**std::string::find**` no encuentra el carácter en la cadena, devuelve `` `std::string::npos` ``. `` `std::string::npos` `` es una constante especial que representa el valor máximo posible para el tipo `size_t` (que es el tipo devuelto por `find`). Comúnmente, `` `std::string::npos` `` se define como `**-1**` en implementaciones de C++, pero es importante notar que es un valor que no puede representarse con precisión en un tipo sin signo como `size_t`.

Por lo tanto, cuando buscas un carácter y `find` no lo encuentra, devuelve `` `std::string::npos` ``, que es un valor grande que se evalúa como `true` en un contexto booleano, lo que puede llevar a la confusión.

Por eso, al utilizar `**std::string::find**` en una condición, debes comparar el resultado con `` `std::string::npos` `` para determinar si el carácter está presente o no en la cadena.

```
if(frase.find(static_cast<char>(i)) != std::string::npos)
```

C++ 23 introduce `std::string::contains` que devuelve un booleano directamente.

std::bitset (pangram)

En las soluciones más eficientes utilizan bitset.

`std::bitset` en C++ es una plantilla de la biblioteca estándar que representa un conjunto de bits, y cada bit puede tener un valor de 0 o 1. Puedes pensar en él como un array de bits fijos, donde el tamaño (número de bits) está determinado en tiempo de compilación.

Cuando creas un `std::bitset<N>`, estás creando un conjunto de N bits. Estos bits se numeran desde el bit 0 hasta el bit N-1. Puedes realizar diversas operaciones a nivel de bit, como establecer un bit en 1 (`set`), borrar un bit (`reset`), cambiar el valor de un bit (`flip`), comprobar si todos los bits están establecidos (`all`), comprobar si alguno de los bits está establecido (`any`), entre otras.

Se utiliza `std::bitset<26>` para representar las letras del alfabeto, donde cada bit representa la presencia de una letra específica. Esto hace que sea eficiente verificar si todas las letras del alfabeto han aparecido al menos una vez en la frase, ya que la información se almacena de manera compacta en los bits del `std::bitset`.

std::map (grade-school)

`second` es un nombre estándar en el contexto de un `std::pair`, que es el tipo de dato que se utiliza para representar los elementos individuales de un `std::map`.

Cuando iteras sobre un `std::map`, cada elemento es un `std::pair` que tiene dos miembros: `first` y `second`. `first` es la clave y `second` es el valor asociado a esa clave.

Por lo tanto, en el código:

```
it->second.push_back(nombre);
```

`it` es un iterador que apunta a un elemento en el `std::map`, y `it->second` es una referencia al valor asociado con la clave de ese elemento. En este caso, el valor es un `std::vector<std::string>`, y `push_back(nombre)` se utiliza para agregar `nombre` al final de ese vector.

En el bucle `for (auto& it : copia)`, `it` es un iterador que apunta a un par clave-valor en el mapa `copia`. El tipo de `it` es `std::map<int, std::vector<std::string>>::value_type`, que es equivalente a `std::pair<const int, std::vector<std::string>>`. Esto significa que `it` contiene tanto la clave (`first`) como el valor (`second`). Puedes acceder a la clave y al valor utilizando `it.first` y `it.second`, respectivamente.

Por lo tanto, en cada iteración del bucle, `it` representa uno de los pares clave-valor en el mapa, permitiéndote trabajar con ambos componentes de manera conveniente.

Uso de ‘auto’ (grade-school)

En C++, la palabra clave `auto` se utiliza para permitir que el compilador infiera automáticamente el tipo de una variable según el valor que recibe en la inicialización. En este caso, la línea:

```
auto it = lista.find(grado);
```

significa que el compilador debe deducir el tipo de `it` basándose en el tipo de retorno de `lista.find(grado)`.

La función `std::map::find` devuelve un iterador que apunta al elemento con la clave especificada, o al final del mapa si la clave no se encuentra. Por lo tanto, el tipo deducido para `it` en este caso sería `std::map<int, std::vector<std::string>>::iterator`.

En otras palabras, `it` será de tipo "iterador al mapa donde la clave es un entero y el valor es un vector de cadenas de texto". Esto es útil porque luego puedes usar `it` para acceder al par clave-valor asociado en el mapa.

std::sort (grade-school)

La función `std::sort` no devuelve nada (especificado como `void`), y su trabajo es ordenar los elementos en el rango que se le pasa como argumento. Por lo tanto, no puedes simplemente devolver el resultado de `std::sort` directamente. Funciona con strings.

No se puede ordenar directamente un `std::map` utilizando `std::sort`. La razón es que `std::map` está intrínsecamente ordenado por sus claves, utilizando un orden definido por el comparador de la clave. La ordenación se realiza de forma automática y se mantiene mientras se insertan y eliminan elementos en el mapa.

const según la posición (grade-school)

En la firma de la función `roster`:

```
const std::map<int, std::vector<std::string>>
&school::roster() const {
    return lista;
}
```

- El primer `const` al principio de la línea significa que la función `roster` devuelve una referencia constante.

- El segundo `const` al final de la línea significa que esta función es constante y, por lo tanto, no puede modificar miembros de la clase (excepto aquellos marcados como `mutable`).

Ahora, desglosemos cada uno de ellos:

1. **`const` al principio:** Indica que la función devuelve una referencia constante. Esto significa que el objeto referenciado no puede ser modificado a través de la referencia devuelta.
2. **`const` al final:** Indica que la función en sí es constante. Esto significa que no puede modificar los miembros de la clase a menos que esos miembros sean declarados como `mutable`. La función `roster` no modificará los miembros de la clase `school` (como `lista`), por lo que se marca como constante.

if usando operaciones bitwise (secret handshake)

Las operaciones bitwise (operaciones a nivel de bits) son operaciones que se realizan directamente sobre los bits individuales de los números en un nivel binario. Estas operaciones se aplican a cada bit de manera independiente y son muy eficientes a nivel de hardware. Son comunes en programación de bajo nivel y en situaciones donde es necesario manipular o analizar la representación binaria de los datos.

Las operaciones bitwise más comunes son:

1. **AND Bitwise (`&`):** Realiza una operación AND bit a bit entre los operandos. El resultado es 1 solo si ambos bits correspondientes son 1.

Ejemplo:
```
    1010 (10 en decimal)
  & 1100 (12 en decimal)
  ------------------
    1000 (8 en decimal)
```

2. **OR Bitwise (`|`)**: Realiza una operación OR bit a bit entre los operandos. El resultado es 1 si al menos uno de los bits correspondientes es 1.

Ejemplo:

```
    1010 (10 en decimal)
  | 1100 (12 en decimal)
  -----------------
    1110 (14 en decimal)
```

3. **XOR Bitwise (`^`):** Realiza una operación XOR bit a bit entre los operandos. El resultado es 1 si los bits correspondientes son diferentes.

Ejemplo:

```
    1010 (10 en decimal)
  ^ 1100 (12 en decimal)
  -----------------
    0110 (6 en decimal)
```

4. **NOT Bitwise (`~`):** Invierte cada bit individual del operando. Cambia 1 por 0 y viceversa.

Ejemplo:

```
   ~1010 (10 en decimal)
  -----------------
    0101 (-11 en decimal en complemento a dos, dependiendo del
tamaño de los bits)
```

Estas operaciones son fundamentales en la manipulación de bits, y se utilizan a menudo en situaciones como codificación/decodificación, algoritmos de compresión, y configuración de registros de hardware, entre otros. También son útiles en el contexto de las máscaras de bits y la manipulación de flags (banderas) en programación.

Solución del ejercicio usando bitwise:

```
if (signal & 0b00001) result.emplace_back("wink");
if (signal & 0b00010) result.emplace_back("double blink");
if (signal & 0b00100) result.emplace_back("close your eyes");
if (signal & 0b01000) result.emplace_back("jump");
if (signal & 0b10000) std::reverse(begin(result),
end(result));
```

Explicación:

La expresión `signal & 0b00001` utiliza la operación **bitwise AND (`&`)** para comprobar si el bit menos significativo (el bit en la posición 0) del número `signal` es 1 o 0. Aquí hay una explicación detallada:

1. **Representación binaria:**
   - `signal` es un número entero sin signo.
   - `0b00001` es una constante en notación binaria que representa el número 1

2. **AND Bitwise (`&`):**
   - La operación `&` realiza una operación AND bit a bit entre los bits correspondientes de los dos operandos.
   - El resultado es 1 solo si ambos bits correspondientes son 1; de lo contrario, el resultado es 0.

3. **Operación específica:**
   - `signal & 0b00001` realiza la operación AND bitwise entre cada bit correspondiente de `signal` y `0b00001`.
   - Los bits más significativos en `0b00001` son 0, y el bit menos significativo es 1.

4. **Resultado:**
   - El resultado será 1 solo si el bit menos significativo de `signal` también es 1
   - Si el bit menos significativo de `signal` es 0, el resultado será 0.

La operación bitwise AND (`&`) puede aplicarse a cualquier tipo de datos entero, como `**int**`, `**unsigned int**`, `**char**`, `**short**`, `**long**`, `**unsigned long**`, etc. Sin embargo, no es aplicable a tipos de datos que no sean enteros, como `std::string` o `double`.

Explicación de la expresión lambda (trinary)

```
if (std::find_if(tri.begin(), tri.end(), [&](char c){ return
!(c == '0' || c == '1' || c == '2'); }) != tri.end()) {
    return resultado;
}
```

- Si todos los caracteres en la cadena `tri` son '0', '1' o '2', entonces el resultado de la expresión lambda será siempre `**false**` para cada uno de ellos.

- En ese caso, `std::find_if` no encontrará ningún elemento que cumpla con la condición especificada en la expresión lambda. Por lo tanto, devolverá el iterador `**tri.end()**`.

- Por otro lado, si encuentra al menos un carácter que no es '0', '1' o '2', entonces la expresión lambda devolverá `**true**` para ese carácter específico.

- En este caso, `std::find_if` devolverá un iterador apuntando al primer elemento que satisface la condición.

Por lo tanto, el condicional `if` comprueba si `std::find_if` ha encontrado algún carácter no válido. Si devuelve `**tri.end()**`, significa que todos los caracteres son válidos, y la condición del `if` se evalúa como `**false**`. Si devuelve un iterador diferente de `**tri.end()**`, significa que al menos un carácter no es válido, y la condición del `if` se evalúa como `**true**`.

Uso de std::stack y aclaración size_t (matching brackets)

`'std::stack'` es una adaptación de contenedor que proporciona la funcionalidad de una pila - una estructura de datos que sigue el principio de último en entrar, primero en salir (LIFO).

Para usar `'std::stack'`, necesitas incluir la biblioteca **`<stack>`**.

Descripción de las operaciones más comunes:

- ➢ **push(g)**: agrega el elemento 'g' en la parte superior de la pila.
- ➢ **pop()**: elimina el elemento superior de la pila. Notar que pop() no devuelve el elemento eliminado.
- ➢ **top()**: accede al elemento superior de la pila. Puedes usar top() para obtener el último elemento que se agregó a la pila (sin eliminarlo).
- ➢ **empty()**: comprueba si la pila está vacía. Devuelve true si la pila no tiene ningún elemento y false en caso contrario.
- ➢ **size()**: devuelve el número de elementos en la pila.

size() de std::stack (y de muchos otros contenedores de la biblioteca estándar de C++) devuelve un **size_t**. **size_t** es un tipo de entero sin signo que se utiliza para representar tamaños y es capaz de representar el tamaño máximo de un objeto en C++.

En cuanto a las operaciones entre **size_t** e **int**, sí, puedes realizarlas, pero debes tener cuidado. Si realizas operaciones entre **size_t** (que es sin signo) e **int** (que es con signo), el **int** se convertirá automáticamente a **size_t** antes de realizar la operación. Esto puede llevar a resultados inesperados si el **int** es negativo.

Por ejemplo, si haces:

```
size_t s = 5;
int i = -3;
auto result = s + i;
```

`result` será un número muy grande en lugar de 2, porque -3 se convierte a un **size_t** muy grande antes de la suma.

Por lo tanto, si estás trabajando con **size_t** e **int** juntos, es una buena práctica convertir explícitamente el **size_t** a **int** antes de realizar la operación, o asegurarte de que tu **int** nunca sea negativo.

Iterar sobre integrales usando el operador >> (pop_count)

```
#include "pop_count.h"

namespace chicken_coop {
    int positions_to_quantity(int numero){

        int contador{0};

        while(numero){
            contador += numero & 1;
            numero >>= 1;
        }
        return contador;
    }
}  // namespace chicken_coop
```

La línea `` `contador += numero & 1;` `` suma el resultado de `` `numero & 1` `` a `` `**contador**` ``. Si el bit menos significativo de `` `**numero**` `` es 1, entonces suma 1. Si el bit menos significativo es 0, entonces suma 0. Aunque sumar 0 no cambia el valor de `` `**contador**` ``, técnicamente sí se realiza la operación de suma.

Después, la línea `numero >>= 1;` desplaza los bits de `**numero**` un lugar a la derecha, lo que tiene el efecto de "sacar" el bit menos significativo. Si piensas en los bits de `**numero**` como una fila de dígitos, es como si estuvieras eliminando el dígito de la derecha.