Generación de números aleatorios

En C++, la generación de números aleatorios se puede realizar de varias formas. A partir de C++11, se introdujo un nuevo conjunto de herramientas para generar números aleatorios, lo que hizo que la generación de números aleatorios fuera más robusta y flexible en comparación con las funciones más antiguas como rand().

Voy a explicarte ambos enfoques:

1. Usando rand() (Método clásico, pre-C++11)
Paso 1: rand() y srand()
rand(): Genera un número entero *pseudoaleatorio* entre 0 y RAND_MAX (*valor constante, generalmente 32767*).
srand(seed): Establece la semilla para la función rand(). Si no se llama a srand(), el generador usa una semilla predeterminada, lo que produce los mismos números aleatorios en cada ejecución del programa.

Ejemplo básico con rand():

```
#include <iostream>
#include <cstdlib>   // Para rand() y srand()
#include <ctime>     // Para time()

int main() {
    // Inicializa la semilla usando el tiempo actual
    srand(static_cast<unsigned>(time(0)));

    // Genera y muestra un número aleatorio
    int random_num = rand();
    std::cout << "Número aleatorio: " << random_num << std::endl;

    // Para generar un número aleatorio dentro de un rango [min, max]
    int min = 1;
    int max = 100;
    int random_in_range = min + rand() % (max - min + 1);
    std::cout << "Número aleatorio entre " << min << " y " << max << ": " << random_in_range <<
std::endl;

    return 0;
}
```

Explicación:
srand(time(0)): Se usa la función time(0) para obtener el tiempo actual como semilla, lo que asegura que los números generados sean diferentes en cada ejecución del programa.
rand() % (max - min + 1): Para generar un número dentro de un rango específico, se usa el operador % (módulo). Este operador limita el rango de los valores que puede generar rand().

Limitaciones de rand():
La función rand() no garantiza una buena calidad en la distribución de los números aleatorios.
Además, RAND_MAX es relativamente pequeño.
La secuencia de números generados es repetible si no se cambia la semilla.

2. Usando la biblioteca <random> (C++11 y posteriores)
El enfoque moderno de C++ usa la biblioteca <random>, que ofrece más control y opciones para generar números aleatorios, incluyendo diferentes distribuciones.

Paso 2: Elementos clave de la generación de números aleatorios en C++11+
Generador de números pseudoaleatorios: Por ejemplo, std::mt19937 (***Mersenne Twister, un generador rápido y de alta calidad***).
Distribuciones: Determinan cómo se distribuyen los números generados. Las más comunes son std::uniform_**int**_distribution (para enteros) y std::uniform_**real**_distribution (para números en punto flotante).

Ejemplo básico usando <random>:

```
#include <iostream>
#include <random>  // Para las funciones de generación aleatoria

int main() {
    // Crea un generador basado en el tiempo actual
    std::random_device rd;  // Fuente de entropía aleatoria (si está disponible)
    std::mt19937 gen(rd()); // Mersenne Twister (motor)

    // Distribución uniforme de enteros en el rango [1, 100]
    std::uniform_int_distribution<> dis(1, 100);
```

```
    // Generar y mostrar 5 números aleatorios en ese rango
    for (int i = 0; i < 5; ++i) {
        std::cout << "Número aleatorio: " << dis(gen) << std::endl;
    }

    return 0;
}
```

Explicación:
std::random_device: Genera una semilla aleatoria de alta calidad basada en una fuente de entropía del hardware (si está disponible). Si no lo está, suele recurrir a otras fuentes pseudoaleatorias.
std::mt19937: Es un motor de generación de números aleatorios basado en el algoritmo *Mersenne Twister*, conocido por su rendimiento y calidad.
std::uniform_int_distribution<>: Define una distribución uniforme para enteros. Aquí se establece que los números deben estar entre **1** y **100**.
Ejemplo con números en punto flotante:

```
#include <iostream>
#include <random>

int main() {
    // Inicialización del generador de números aleatorios
    std::random_device rd;
    std::mt19937 gen(rd());

    // Distribución uniforme de números en punto flotante [0.0, 1.0]
    std::uniform_real_distribution<> dis(0.0, 1.0);

    // Generar y mostrar 5 números aleatorios en punto flotante
    for (int i = 0; i < 5; ++i) {
        std::cout << "Número aleatorio (real): " << dis(gen) << std::endl;
    }

    return 0;
}
```

Explicación:
std::uniform_real_distribution<>: Se utiliza para generar números aleatorios en punto flotante dentro del rango [0.0, 1.0].

Ventajas del enfoque moderno:
Calidad de los números: Los generadores como **std::mt19937** ofrecen secuencias de números pseudoaleatorios de mayor calidad.

Flexibilidad: Hay varias distribuciones disponibles, como normal (**std::normal_distribution**), binomial (**std::binomial_distribution**), entre otras.

Portabilidad: Los generadores modernos son más portables entre diferentes plataformas.

Resumen:
rand() es un *método más antiguo, útil para situaciones simples*, pero con limitaciones.
<random> (C++11 y posteriores) ofrece más control, mejor rendimiento y una mayor calidad en la generación de números aleatorios, *siendo la opción recomendada para la mayoría de los casos*.

Enfoque:

Numero aleatorio sin que se repita:

Para evitar que un número aleatorio se repita en tu programa, puedes mantener un registro de los números generados y verificar si ya se generaron antes de agregarlos a los arreglos. Una manera eficiente de hacerlo es utilizar una estructura como un arreglo auxiliar o un contenedor como `std::set` que automáticamente garantiza la **unicidad**.

Aquí está el código para evitar números repetidos utilizando `std::set`:

*/*Se realiza un programa el cual debe generar 40 números aleatorios y almacenarlos por separado en un arreglo, por un lado los números pares y por otro la (otro arreglo), con los números impares*/*

(sin repetición de números aleatorios):

```
#include <iostream>
#include <random>
#include <set> // Para mantener un conjunto de números únicos

int main() {
    system("clear");
    std::cout << "\n\n";

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 100);

    std::cout << "Realizando la Carga de los valores aleatorios: \n";

    int arrPares[40], arrImpares[40], generar{}, conPares = 0, conImpares = 0;
    std::set<int> numerosGenerados; // Para registrar números únicos

    while (numerosGenerados.size() < 40) { // Continuar hasta generar 40 números únicos
        generar = dis(gen);

        // Verificar si el número ya fue generado
        if (numerosGenerados.find(generar) == numerosGenerados.end()) {
            numerosGenerados.insert(generar); // Agregar número al conjunto

            if (generar % 2 == 0) {
                arrPares[conPares] = generar;
                conPares += 1;
            } else {
                arrImpares[conImpares] = generar;
                conImpares += 1;
            }
        }
    }

    // Mostrar los números pares
    std::cout << "Mostrando los Valores Aleatorios Pares:\n";
    for (int i = 0; i < conPares; i++) {
        std::cout << "Posición [" << i << "] contenido [" << arrPares[i] << "]\n";
    }

    // Mostrar los números impares
    std::cout << "\nMostrando los Valores Aleatorios Impares:\n";
    for (int i = 0; i < conImpares; i++) {
        std::cout << "Posición [" << i << "] contenido [" << arrImpares[i] << "]\n";
    }

    std::cout << "\n\n";
    return 0;
}
```

---

1. **Uso de** std::set:
    - std::set es un contenedor que solo almacena valores únicos. Esto garantiza que no se repitan números.
    - Antes de usar un número aleatorio, se verifica si ya está en el conjunto numerosGenerados. Si no está, se agrega.
2. **Bucle** while:
    - El bucle principal a un while para continuar generando números hasta que tengamos exactamente 40 números únicos.
3. **Llenado de arreglos**:
    - Los números generados se clasifican como pares o impares y se almacenan en los arreglos correspondientes.

Explicación:

- **Verificación de unicidad**:
    - Antes de agregar un número a los arreglos arrPares o arrImpares, se verifica si el número ya está en el conjunto numerosGenerados.
    - Si ya fue generado, se ignora y se genera un nuevo número.
- **Eficiencia de** std::set:
    - La búsqueda en un conjunto tiene una complejidad de O(logn), lo que lo hace eficiente para evitar duplicados.

Ejemplo de salida:

```
Realizando la Carga de los valores aleatorios:
Mostrando los Valores Aleatorios Pares:
Posición [0] contenido [2]
Posición [1] contenido [28]
Posición [2] contenido [90]
...

Mostrando los Valores Aleatorios Impares:
Posición [0] contenido [1]
Posición [1] contenido [77]
Posición [2] contenido [55]
...
```

Con este enfoque, puedes garantizar que los números generados sean únicos.

.find():

La operación .find() se utiliza para buscar un elemento dentro de un contenedor en C++. Es una función miembro de algunos contenedores estándar, como std::string, y también existe como función genérica en <algorithm> para otros contenedores, como std::set, std::vector, o std::map.

---

1. En contenedores asociativos (std::set, std::map, etc.)

- **Sintaxis**:

  ```
  auto it = contenedor.find(valor);
  ```

- **Función**: Busca el elemento valor en el contenedor y devuelve un iterador apuntando al elemento si se encuentra. Si el elemento no existe, devuelve un iterador al final del contenedor (contenedor.end()).

Ejemplo con std::set:

```
#include <iostream>
#include <set>

int main() {
    std::set<int> numeros = {10, 20, 30, 40};

    // Buscar un valor
    auto it = numeros.find(20);

    if (it != numeros.end()) {
        std::cout << "Valor encontrado: " << *it << "\n";
    } else {
        std::cout << "Valor no encontrado.\n";
    }

    return 0;
}
```

Salida:

Valor encontrado: 20

---

2. En std::string

- **Sintaxis**:

  ```
  size_t pos = cadena.find(subcadena);
  ```

- **Función**: Busca la posición de la primera aparición de **subcadena** dentro de cadena. Si no la encuentra, devuelve std::string::npos.

Ejemplo con std::string:

```
#include <iostream>
#include <string>

int main() {
    std::string texto = "Hola, mundo";
    size_t pos = texto.find("mundo");

    if (pos != std::string::npos) {
        std::cout << "Subcadena encontrada en la posición: " << pos << "\n";
    } else {
        std::cout << "Subcadena no encontrada.\n";
    }

    return 0;
}
```

Salida:

Subcadena encontrada en la posición: 6

---

3. En contenedores secuenciales (con <algorithm>)

Para contenedores como std::vector, std::list, o arreglos, .find() no es un método, pero puedes usar la función genérica std::find() del encabezado <algorithm>.

- **Sintaxis**:

  ```
  auto it = std::find(contenedor.begin(), contenedor.end(), valor);
  ```

- **Función**: Busca el valor en el rango especificado y devuelve un iterador apuntando al primer elemento que coincide con el valor buscado. Si no se encuentra, devuelve contenedor.end().

Ejemplo con std::vector:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numeros = {1, 2, 3, 4, 5};

    // Buscar el valor 3
    auto it = std::find(numeros.begin(), numeros.end(), 3);

    if (it != numeros.end()) {
        std::cout << "Valor encontrado: " << *it << "\n";
    } else {
        std::cout << "Valor no encontrado.\n";
    }

    return 0;
}
```

Salida:

```
Valor encontrado: 3
```

Resumen de .find():

Contenedor	Resultado si el elemento se encuentra	Resultado si el elemento no se encuentra
`std::set`, `std::map`	Iterador al elemento encontrado	Iterador al final del contenedor (`contenedor.end()`)
`std::string`	Índice (posición) de la primera aparición	`std::string::npos`
`<algorithm>::find()`	Iterador al elemento encontrado en el rango especificado	Iterador al final del rango (`contenedor.end()`)

Ventajas de .find():

1. **Contenedores Asociativos (std::set, std::map):**
    - La operación tiene una complejidad de O(logn) gracias a las propiedades de los árboles binarios balanceados.
2. **Cadenas (std::string):**
    - Ideal para localizar subcadenas en texto de manera eficiente.
3. **Función Genérica (std::find):**
    - Funciona con cualquier contenedor iterativo, aunque la complejidad depende del contenedor (es lineal O(n) para std::vector y otros contenedores secuenciales).