REFERENCIA DE CLASES DE UTILIDAD EN C++ (2021)

<STRING> FUNCIONES MIEMBRO	
La constante **std::string::npos** representa o bien el final del string o que ha habido un error (no se encuentra un texto, por ejemplo)	
Constructores: string(**int** c, **char** ch) string(string& str) string(string& str, **int** p, **int** c = npos) string(**char*** s, **int** c) string(**char*** s) string(iterador p, iterador u)	 Construye un string con 'c' copias del carácter 'ch' Construye una copia del string 'str' Construye un string con los caracteres del rango ['p' ; 'p'+'c') del string 'str'. Si saliéramos del rango de 'str', el rango se modifica a ['p', size()) Construye un string con los primeros 'c' caracteres de la cadena de caracteres 's' Construye un string con la cadena de caracteres 's' (¡ojo!, 's' debe acabar con '\0') Construye un string con el contenido del rango ['p' ; 'u')
int size()	Retorna la cantidad de caracteres que almacena actualmente el string
bool empty()	Comprueba si el string está vacío
int capacity()	Retorna la cantidad máxima de caracteres que puede almacenar el string
char& operator[] (**int** pos)	Retorna una referencia al carácter almacenado en la posición 'pos'
char& front()	Retorna una referencia al primer carácter del string
char& back()	Retorna una referencia al último carácter del string
void reserve (**int** tam)	Aumenta la capacidad de almacenamiento del string a 'tam' caracteres
void clear()	Borra el contenido del string pero no cambia su capacidad
string& **operator=** (string& rhs) string& **operator=** (**char*** rhs)	Operador de asignación por copia Operador de asignación por copia (¡ojo!, 's' debe acabar en '\0')
void insert (**int** p, **int** c, **char** ch) **void** insert (**int** p, **char*** s) **void** insert (**int** p, **char*** s, **int** c) **void** insert (**int** p, string& str) **void** insert (**int** p, string& str, **int** p_str, **int** c_str)	Inserta 'c' copias of carácter 'ch' en la posición 'p' Inserta la cadena de caracteres 's' en la posición 'p' (¡ojo!, 's' debe acabar con '\0') Inserta 'c' caracteres de la cadena de caracteres 's' en la posición 'p' (¡ojo!, 's' debe acabar con '\0') Inserta el string 'str' en la posición 'p' Inserta los elemento del rango ['p_str' ; 'p_str'+'c_str') de 'str' en la posición 'p'
append	Exactamente igual que insert solo que sin el parámetro 'p'. Siempre añade al final del string
void erase(**int** p = 0, **int** c = npos) iterador erase(iterator p) iterador erase(iterator p, iterator u)	Elimina 'c' caracteres del string comenzando por la posición 'p' Elimina el carácter en la posición apuntada por 'p' Elimina los caracteres del rango ['p' ; 'u')
void pop_back()	Elimina el último carácter del string
string substr(**int** p = 0, **int** c = npos)	Retorna un nuevo string con el rango ['p' ; 'p'+'c'). Si saliéramos del rango, el rango se modifica a ['p', size())
int find(string& str, **int** p = 0) **int** find(**char*** s, **int** p, **int** c) **int** find(**char*** s, **int** p=0)	Retorna la posición de la primera ocurrencia de 'str' a partir de la posición 'p', o **npos** si no se encuentra 'str' Retorna la posición de la primera ocurrencia de los primeros 'c' caracteres de la cadena de caracteres 's' a partir de la posición 'p', o **npos** si no se encuentra Retorna la posición de la primera ocurrencia de la cadena de caracteres 's' a partir de la posición 'p', o **npos** si no se encuentra (¡ojo!, 's' debe acabar con '\0')
int rfind(string& str, **int** p = npos) **int** rfind(**char*** s, **int** p, **int** c) **int** rfind(**char*** s, **int** p=npos)	Igual que **find**, pero en reversa. Busca las primeras apariciones de la cadena a partir de la posición 'p' pero "avanzando" hasta el principio del string
int find_first_of (...) mismas versiones que de **find**	Encuentra el primer carácter que aparece en el string que coincide con alguno de los caracteres que se pasan en los parámetros de tipo string o cadena de caracteres
int find_last_of (...) mismas versiones que de **rfind**	Encuentra el último carácter que aparece en el string que coincide con alguno de los caracteres que se pasan en los parámetros de tipo string o cadena de caracteres
int find_first_not_of (...) mismas versiones que de **find**	Encuentra el primer carácter que aparece en el string que no coincide con alguno de los caracteres que se pasan en los parámetros de tipo string o cadena de caracteres
int find_last_not_of (...) mismas versiones que de **rfind**	Encuentra el último carácter que aparece en el string que no coincide con alguno de los caracteres que se pasan en los parámetros de tipo string o cadena de caracteres
OTRAS FUNCIONES (NO MIEMBRO) DEFINIDAS EN <STRING>	
int stoi(string& str) **long** stol(string& str) **double** stod(string& str)	Obtiene un número **int**, **long** o **double** a partir del string 'str'
string to_string(**int** v) string to_string(**long** v) string to_string(**double** v)	Obtiene un string a partir del parámetro 'v'
ostream& **operator<<** (ostream &os, string &str)	Inserta el contenido del string 'str' en el stream 'os'
istream& **operator>>** (istream &is, string &str)	Lee caracteres del stream 'is' hasta encontrar un separador (espacio en blanco, tabulador o retorno de carro) y los almacena en 'str'
bool operator<, <=, >, >=, ==, != (string l, string r)	Compara el contenido de los strings 'l' y 'r' caracter a caracter. Los operadores de igualdad requieren que los strings tengan el mismo tamaño. Los restantes no, pero si todos los caracteres son iguales, es <= el que tiene menos caracteres
istream& getline(istream& is, string& str, **char** d='\n')	Lee del stream de entrada 'is' todos los caracteres que pueda hasta encontrar el delimitador 'd' y lo almacena en 'str'
string **operator+**(string& l, string& r) string **operator+**(string& l, **char*** r) string **operator+**(**char*** l, string& r) string **operator+**(string& l, **char** r) string **operator+**(**char** l, string& r)	Retorna un string formado por la concatenación de 'l' y 'r' ¡ojo!, las versiones con parámetros de tipo cadena de caracteres (char*) requieren que haya un '\0'
void swap (string &l, string &r)	Intercambia el contenido de los strings 'l' y 'r'

<UTILITY> PAIR – FUNCIONES MIEMBRO	
template<class T1, **class** T2> pair () pair (T1 t1, T2 t2)	Construye un objeto de tipo par: • Invocando los constructores por defecto de los tipos de la plantilla • Invocando los constructores de copia con los valores suministrados
pair& **operator**=(**const** pair& rhs)	Operador de asignación por copia
first, second	Nombre de los campos de la estructura (tienen visibilidad pública)
OTRAS FUNCIONES (NO MIEMBRO) DEFINIDAS EN <UTILIY>	
bool operator<, <=, >, >=, ==, != (pair &l, pair &r)	Compara el contenido de los pares 'l' y 'r'. Primero, el campo 'first'. Si son iguales, pasa al campo 'second'. El orden de los elementos importa
void swap (pair &l, pair &r)	Intercambia el contenido de los pares 'l' y 'r'
template<class T1, **class** T2> pair<T1,T2> make_pair(T1 t, T2 u)	Crea y retorna un par con los valores y los tipos suministrados como parámetros

<TUPLE> TUPLE – FUNCIONES MIEMBRO	
template<class ... Tipos> tuple () tuple (Tipos ... args)	Construye un objeto de tipo tupla: • Invocando los constructores por defecto de los tipos de la plantilla • Invocando los constructores de copia con los valores suministrados
tuple& **operator**=(**const** tuple& rhs)	Operador de asignación por copia
OTRAS FUNCIONES (NO MIEMBRO) DEFINIDAS EN <TUPLE>	
bool operator<, <=, >, >=, ==, != (tuple &l, tuple &r)	Compara el contenido de las tuplas 'l' y 'r' elemento a elemento, igual que en pair. El orden de los elementos en la tupla importa
void swap (pair &l, pair &r)	Intercambia el contenido de las tuplas 'l' y 'r'
template< class ... Tipos > tuple<Tipos ...> make_tuple(Tipos ... args)	Crea y retorna una tupla con los valores y los tipos suministrados como parámetros
X get<**int**> (tuple t)	Retorna el valor (X) almacenado en la posición que indica el parámetro genérico

<OPTIONAL> FUNCIONES MIEMBRO	
Define la constante std::nullopt, que modela un optional genérico sin valor	
void reset()	Invoca el destructor del valor que contiene el optional
bool has_value()	Comprueba si el optional almacena un valor
X value_or (T valor)	Retorna el valor almacenado en el optional o 'valor' si no tiene
X operator*() X value()	• Retorna el valor (X) almacenado en el optional. Undefined behaviour si no almacena valor • Retorna el valor (X) almacenado en el optional. Excepción si no almacena valor
optional& **operator**=(**const** optional& rhs)	Operador de asignación por copia
OTRAS FUNCIONES (NO MIEMBRO) DEFINIDAS EN <OPTIONAL>	
bool operator<, <=, >, >=, ==, != (optional &l, optional &r)	Compara el contenido de los optional 'l' y 'r'. Un no-valor es menor que cualquier valor y los no-valores son iguales entre sí
void swap (optional &l, optional &r)	Intercambia el contenido de los optional 'l' y 'r'
template<class T> optional<T> make_optional(T t)	Crea y retorna un optional con el valor del tipo suministrado como parámetro
template<class T, **class** ... Args> optional<T> make_optional(Args ... args)	Crea y retorna un optional con el valor del tipo 'T' creado in-place invocando el constructor de T que acepta la lista de parámetros 'args'

<VECTOR> FUNCIONES MIEMBRO	
template<class T> vector () vector (**int** t) vector (**int** t, T v) vector (InputIt p, InputIt u) vector (**const** vector& v) vector (initializer_list<T> l)	Vector requiere la especificación del tipo de dato a almacenar, T • Construye un vector vacío • Construye un vector de tamaño 't' con valores inicializados por defecto • Construye un vector de tamaño 't' con valores inicializados al 'v' • Construye un vector con los elementos del rango ['p' ; 'u') • Construye una copia del vector 'v' • Construye un vector con la lista de elementos proporcionados
int size()	Retorna la cantidad de elementos que almacena actualmente el vector
int capacity()	Retorna la cantidad máxima de elementos que puede almacenar actualmente el vector
void reserve(**int** tam)	Aumenta la capacidad de almacenamiento del vector a 'tam' elementos
T& front()	Retorna una referencia al primer elemento del vector
T& back()	Retorna una referencia al último elemento del vector
T& operator[] (**int** pos)	Obtiene una referencia al elemento almacenado en la posición 'pos'. No inserta valor. Solo lectura
void clear()	Borra el contenido del vector pero no cambia su capacidad
void erase(iterador pos) **void** erase(iterador p, iterador u)	Borra el elemento almacenado en 'pos' Borra los elementos del rango ['p' ; 'u')
void push_back(T& v);	Inserta el elemento 'v' al final del vector
void emplace_back(args...)	Invoca el constructor del elemento con parámetros 'args' y lo inserta al final del vector
iterator emplace(iterator pos, args...)	Invoca el constructor del elemento con parámetros 'args' y lo inserta antes de 'pos'. Retorna un iterador al nuevo elemento insertado
iterador insert(iterator pos, T v) iterador insert(iterador pos, **int** c, T v) iterador insert (iterator pos, InputIt p, InputIt u)	Inserta el elemento 'v' antes de 'pos' y retorna un iterador al valor 'v' Inserta 'c' copias del elemento 'v' antes de 'pos' y retorna un iterador a la primera copia Inserta los elementos del rango ['p' ; 'u') de otra colección antes de 'pos' y retorna un iterador al primer elemento insertado
void pop_back()	Elimina el último elemento del vector
bool empty()	Retorna **true** si el vector está vacío

`vector& operator= (vector& rhs)`	Operador de asignación por copia
OTRAS FUNCIONES (NO MIEMBRO) DEFINIDAS EN <VECTOR>	
`bool operator<, <=, >, >=, ==, != (vector &l, vector &r)`	Compara el contenido de los vectores 'l' y 'r' elemento a elemento. Los operadores de igualdad requieren que los vectores tengan el mismo tamaño. Los restantes no, pero si todos los elementos son iguales, es <= el que tiene menos elementos
`void swap (vector &l, vector &r)`	Intercambia el contenido de los vectores 'l' y 'r'

<MAP> FUNCIONES MIEMBRO	

`template<class Key, class T>`
`map ()`
`map (InputIt p, InputIt u)`
`map (const map& m)`
`map (initializer_list<K, T> l)`
	Map requiere la especificación de los tipos de las parejas clave-valor (Key-T) que almacena
	• Construye un mapa vacío
	• Construye un mapa con los elementos del rango ['p' ; 'u')
	• Construye una copia del mapa 'm'
	• Construye un mapa con la lista de elementos proporcionados en 'l'

`int size()`	Retorna la cantidad de elementos (parejas clave-valor) que almacena actualmente el mapa
`T& operator[] (Key pos)`	Obtiene una referencia al elemento almacenado en la posición 'pos' o crea uno nuevo si no existía
`void clear()`	Borra el contenido del mapa

`void erase(iterador pos)`
`void erase(iterador p, iterador u)`
`int erase (Key k)`
	Borra el elemento almacenado en 'pos'
	Borra los elementos del rango ['p' ; 'u')
	Elimina los elementos con clave equivalente a 'k' y retorna el número de elementos eliminados

`pair<iterador, bool> insert (pair<Key, T> v)`	Inserta el elemento 'v' en el mapa haciendo una copia de 'v'. Retorna una pareja que contiene un iterador a la posición donde se ha almacenado el elemento y si la inserción ha tenido éxito
`pair<iterador,bool> emplace (Args&& ...)`	Como insert, pero invoca el constructor de 'T' para crear el elemento en la posición concreta
`bool empty()`	Retorna **true** si el mapa está vacío
`map& operator= (map& rhs)`	Operador de asignación por copia
`int count (const Key &k)`	Retorna el número de elementos del mapa que tienen clave equivalente a 'k'
`iterador find (const Key &k)`	Retorna un iterador al elemento con clave equivalente 'k' o el iterador **end()** si no lo encuentra
OTRAS FUNCIONES (NO MIEMBRO) DEFINIDAS EN <MAP>	
`bool operator<, <=, >, >=, ==, != (map &l, map &r)`	Compara el contenido de los mapas 'l' y 'r' elemento a elemento. Los operadores de igualdad requieren que los vectores tengan el mismo tamaño. Los restantes no, pero si todos los elementos son iguales, es <= el que tiene menos elementos
`void swap (map &l, map &r)`	Intercambia el contenido de los mapas 'l' y 'r'

<SET> FUNCIONES MIEMBRO	

`template<class Key>`
`set ()`
`set (InputIt p, InputIt u)`
`set (const set& s)`
`set (initializer_list<K> l)`
	Set requiere la especificación del tipo de la Clave, puesto que no guarda valores repetidos
	• Construye un set vacío
	• Construye un set con los elementos del rango ['p' ; 'u')
	• Construye una copia del set 's'
	• Construye un set con la lista de elementos proporcionados en 'l'

`int size()`	Retorna la cantidad de claves que almacena actualmente el set
`void clear()`	Borra el contenido del set

`void erase (iterador pos)`
`void erase (iterador p, iterador u)`
`int erase (Key k)`
	Borra el elemento almacenado en 'pos'
	Borra los elementos del rango ['p' ; 'u')
	Elimina los elementos con clave equivalente a 'k' y retorna el número de elementos eliminados

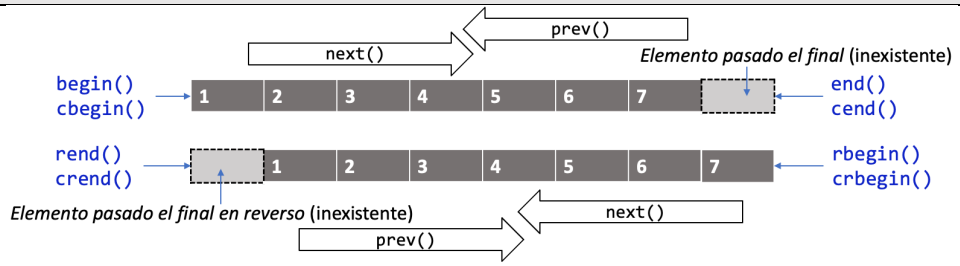
`pair<iterador,bool> emplace (Args&& ...)`	Invoca el constructor de 'Key' para crear el elemento en la posición concreta
`pair<iterador, bool> insert (Key v)`	Intenta insertar el elemento 'v' en el set. Igual que el mapa
`bool empty()`	Retorna **true** si el set está vacío
`set& operator= (set& rhs)`	Operador de asignación por copia
`int count (const Key &k)`	Retorna el número de elementos del set que tienen clave equivalente a 'k'
`iterador find (const Key &k)`	Retorna un iterador al elemento con clave equivalente 'k' o el iterador **end()**
OTRAS FUNCIONES (NO MIEMBRO) DEFINIDAS EN <SET>	
`bool operator<, <=, >, >=, ==, != (set &l, set &r)`	Compara el contenido de los sets 'l' y 'r' elemento a elemento. Los operadores de igualdad requieren que los vectores tengan el mismo tamaño. Los restantes no, pero si todos los elementos son iguales, es <= el que tiene menos elementos
`void swap (set &l, set &r)`	Intercambia el contenido de los sets 'l' y 'r'

<IOSTREAM> FUNCIONES MIEMBRO	
Define los streams `std::cout` (para salida por consola) y `std::cin` (para entrada por teclado), así como la funcionalidad básica que aplica a todos los streams soportados por C++: ficheros (**fstream**) y acceso a strings (**stringstream**)	
`void clear()`	Limpia los indicadores de error del stream (eof, fail y bad)
`void ignore(int m, char c)`	Elimina o bien 'm' caracteres del stream o hasta que encuentra el carácter 'c'
`bool good()`	Comprueba si no se ha producido ningún error en el stream
`bool eof()`	Comprueba si se ha llegado al final del stream
`bool fail()`	Comprueba si ha ocurrido un error recuperable
`bool bad()`	Comprueba si ha ocurrido un error no recuperable
`void setstate (ios_base s)`	Define el estado del stream. Posibles valores para 's': `ios_base::goobit, ios_base::badbit, ios_base::failbit, ios_base::eofbit`

<ITERATOR> (FUNCIONES NO MIEMBRO)

Las funciones para obtener un iterador a partir de una colección tienen la forma
`iterador funcion (coleccion)`

Ejemplo:
`auto it { cbegin(un_vector) };`



`iterador next (iterador i)`	Retorna un iterador al siguiente elemento apuntado por 'i'
`iterador prev (iterador i)`	Retorna un iterador al elemento previo al apuntado por 'i'
`int distance(iterador p, iterador u);`	Retorna el número de elementos contenidos en el rango ['p' ; 'u']
`void advance(iterador& i, int n)`	Avanza el iterador 'i' 'n' posiciones en la colección asociada
`iterador& operator= (iterador &s)`	Operador de asignación por copia
`bool operator==,!=, <, <=, >, >= (iterador& l, iterador& r)`	Comprueban si los iteradores apuntan a la misma colección, y en caso afirmativo si además el iterador 'l' apunta a un elemento que esté antes o después que el del iterador 'r'
`T operator*()`	Proporciona acceso al elemento apuntado por el iterador

<FSTREAM> FUNCIONES MIEMBRO

Define las clases `ifstream` y `ofstream` que proporcionan acceso de lectura y escritura, respectivamente, a ficheros contenidos en disco. `ifstream` proporciona **operator**>> para leer contenido con formato y `ofstream` **operator**<< para escribir contenido con formato

Constructores: `ifstream(string& f) / ofstream(string& f)`	Construye un nuevo flujo a un fichero a partir del texto que representa la ruta de acceso al fichero. Abre el fichero si tiene éxito
`bool isopen()`	Comprueba si el stream tiene asociado un fichero en disco
`void close()`	Cierra el stream asociado al fichero

<SSTREAM> FUNCIONES MIEMBRO

Define las clases `ostringstream`, `istringstream` y `stringstream`. Todas las clases generan un stream a partir de un string, de forma que se pueden aplicar **operator**<< y **operator**>> (lectura/escritura con formato). `stringstream` fusiona la funcionalidad de las otras clases.

Constructores:
`stringstream () / stringstream (string& str)`
`istringstream () / istringstream (string &str)`
`ostringstream () / ostringstream (string &str)`
 	
Los 6 constructores tienen una versión con un valor adicional del tipo 'mode'
	Mode:
	`ios_base::app` las escrituras se realizan al final (append)
	`ios_base::binary` abrir en modo binario
	`ios_base::in` abrir en modo lectura
	`ios_base::out` abrir en modo escritura
	`ios_base::trunc` descarta el contenido del stream (truncar)
	`ios_base::ate` ir al final del flujo al abrirlo (at the end)

`string str()`	Retorna una copia del string asociado a este stream
`void str(string& s)`	Cambia el contenido del string asociado al stream por 's'

<RANDOM>

Separa la generación de números aleatorios de las distribuciones probabilísticas que se muestrean para obtenerlos. El código para el generador es:
`std::random_device rd;`
`std::default_random_engine gen(rd());`
Algunas distribuciones soportadas se enumeran a continuación, pero todas se utilizan igual: `distribucion(gen)` genera el número aleatorio

`uniform_int_distribution<> (int a, int b)` `int a() / int b()`	Construye una distribución uniforme en el rango ['a' ; 'b'] Retorna el valor 'a' / el valor 'b' El siguiente código (`std::stringstream ss {"99 120"}; ss >> dis;`) cambia los parámetros de la distribución uniforme `dis`
`normal_distribution<> (double m, double d)` `double mean () / double stddev()`	Construye una distribución normal con media 'm' y desviación 'd' Retorna la media y la desviación de la distribución normal

<CMATH>

Basic functions: `abs, fabs, fmod, remainder, remquo, fma, fmax, fmin, fdim, pow, sqrt, cbrt, hypot`
Exponential functions: `exp, exp2, expm1, log, log10, log2, log1p`
Trigonometric functions: `sin, cos, tan, asin, acos, atan, atan2`
Rounding functions: `ceil, floor, trunc, round, rint, nearbyint`
Floating point functions: `frexp, ldexp, modf, scalbn, logb, ilogb, copysign`
`M_E: e; M_LOG2E: log2 (e); M_LOG10E: log10 (e); M_LN2: ln (2); M_LN10: ln (10); M_PI: pi; M_PI_2: pi/2; M_PI_4: pi/4; M_1_PI: 1/pi; M_2_PI: 2/pi; M_2_SQRTPI: 2/sqrt (pi) M_SQRT2: sqrt (2); M_SQRT1_2: 1/sqrt (2)`

<THREAD> FUNCIONES MIEMBRO

Constructores: `thread (Funcion f, args ...)`	'f' es el nombre de la función que ejecutará el hilo y 'args' es una lista separada por comas de valores para 'f'. Si 'f' es función miembro, la sintaxis para que un hilo la ejecute es: `Clase obj; thread t {&Clase::f, &obj, args};`
`thread::id get_id()`	Retorna el identificador del hilo
`void join()`	Bloquea la ejecución hasta que el hilo termina.
Namespace std::this_thread (funciones no miembro)	
`thread::id this_thread::get_id()`	Retorna el identificador del hilo que ejecuta la función
`void this_thread::sleep_for (chrono::duration &d)`	Duerme el hilo que ejecuta la función (espera relativa) por el tiempo 'd'
`void this_thread::sleep_until (chrono::time_point &t)`	Duerme el hilo que ejecuta la función (espera absoluta) hasta la hora 't'

<chrono>	
Define varios relojes, el tipo **duration** (representa un tiempo relativo) y **time_point** (representa un tiempo absoluto)	
`time_point<system_clock> h = std::chrono::system_clock::now() // hora real del sistema` `time_point<steady_clock> h = std::chrono::steady_clock::now() // hora según reloj monotónico`	
Funcionalidad relacionada con duraciones de tiempo (duration)	
`count ()`	Retorna la cantidad de ticks del objeto duration sobre el que se invoca (el tiempo que representa)
`duration& operator=(duration &d)`	Operador de asignación por copia

`chrono::nanoseconds chrono::microseconds`
`chrono::milliseconds chrono::seconds`
`chrono::minutes chrono::hours`
	Tipos de datos que representan duraciones (**duration**) concretas

`duration operator+,-` `(duration &l, duration &r)`	Obtiene un nuevo objeto duration resultado de la suma/resta de los dos párametros

`template<class T> duration`
`duration_cast<T>(duration v)`
	Convierte el objeto duration 'v' al tipo duration <T>. Ejemplo:
	`chrono::seconds s (100);`
	`auto int_ms = chrono::duration_cast<chrono::milliseconds>(s);`

`bool operator==,!=, <, <=, >, >=` `(duration &l, duration &r)`	Compara los objetos duration 'l' y 'r'
`time_point& operator+=(duration& d)` `time_point& operator-=(duration& d)`	Modifica el objeto time_point sobre el que se invoca la función sumando o restando 'd'
`bool operator==,!=, <, <=, >, >=` `(time_point &l, time_point &r)`	Compara los objetos time_point 'l' y 'r'

<mutex> Funciones miembro	
Define varios tipos de mutex, una clase para gestionar un mutex en un ámbito (**unique_lock**) y varias funciones de utilidad	
`Constructores: mutex ()`	Crea un mutex pero no intenta cogerlo
`void lock()`	Bloquea el hilo que invoca la función hasta que pueda coger el mutex. Si un hilo intenta coger un mutex que ya tiene cogido se produce un *deadlock*
`void unlock()`	Libera un mutex. Si el hilo no tiene el mutex, se produce comportamiento no definido
`bool try_lock()`	Intenta coger el mutex. Si lo consigue, retorna **true**.
	La clase **recursive_mutex** proporciona la misma interfaz, salvo que un mutex recursivo puede ser cogido múltiples veces por el mismo hilo, que tiene que liberarlo tantas veces como lo ha cogido
`template<class Mutex >` `unique_lock`	Coge el mutex que se pasa como parámetro y lo libera cuando termina la vida de la variable: `mutex m; std::unique_lock lg {m};`
Otras funciones (no miembro) definidas en <mutex>	
`void lock (args ...)`	Intenta coger todos los mutex especificados en la lista de argumentos, bloqueando. Si se produce algún error, invoca **unlock()** antes de terminar
`void try_lock (args ...)`	Intenta coger todos los mutex especificados en la lista de argumentos. Si se produce algún error o alguno de ellos no está libre, invoca **unlock()** antes de terminar

<condition_variable> Funciones miembro	
`Constructores:condition_variable()`	Construye una variable de condición
`void notify_one()`	Desbloquea uno de los hilos bloqueados por una llamada a **wait()** en la variable de condición
`void notify_all()`	Desbloquea todos los hilos bloqueados por una llamada a **wait()** en la variable de condición
`void wait(unique_lock<mutex> &m, funcion)`	Bloque un hilo en la variable de condición asociada a 'm' dependiendo de si se cumple la condición lógica implementada en la 'función' (lambda function)

<cctype>

Todas las funciones reciben un carácter como parámetro y devuelven el valor indicado dependiendo si coincide con el tipo especificado en la primera columna

characters	iscntrl	isprint	isspace	isblank	isgraph	ispunct	isalnum	isalpha	isupper	islower	isdigit	isxdigit
control codes (NUL, DEL, etc.)	≠0	0	0	0	0	0	0	0	0	0	0	0
tab (\t)	≠0	0	≠0	≠0	0	0	0	0	0	0	0	0
whitespaces (\n, \v, \f, \r)	≠0	0	≠0	0	0	0	0	0	0	0	0	0
space	0	≠0	≠0	≠0	0	0	0	0	0	0	0	0
!"#$%&'()*+-./:;<=>?@[\]^_`{|}~	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
0123456789	0	≠0	0	0	≠0	0	≠0	0	0	0	≠0	≠0
ABCDEF	0	≠0	0	0	≠0	0	≠0	≠0	≠0	0	0	≠0
GHIJKLMNOPQRS TUVWXYZ	0	≠0	0	0	≠0	0	≠0	≠0	≠0	0	0	0
abcdef	0	≠0	0	0	≠0	0	≠0	≠0	0	≠0	0	≠0
ghijklmnp qrstuvwxyz	0	≠0	0	0	≠0	0	≠0	≠0	0	≠0	0	0