Bibliotecas estándar #include<>

Para incluir las bibliotecas estándar en **C++** se utiliza la directiva **#include,** que indica al *preprocesador que se debe incluir el contenido de un archivo en el punto donde se encuentra la directiva*. Por ejemplo, para usar la biblioteca **iostream**, *que proporciona recursos para que un programa pueda aceptar datos de entrada y proporcionar una salida de resultados, se debe incluir el archivo de cabecera* **<iostream>**.

Las ***cabeceras*** son archivos que contienen declaraciones de funciones, definiciones de tipos y macros. Suelen tener la extensión **.h** o **.hpp**, y se incluyen en los archivos **.cpp** y **.h**.

La biblioteca estándar de **C++** es una colección de clases y funciones que proporciona un marco extensible para el lenguaje. Contiene componentes para:

- Diagnósticos
- Utilidades generales
- Cadenas
- Configuraciones regionales
- Entrada y salida
- Soporte para algunas características del lenguaje
- Funciones para tareas comunes

Las características de la biblioteca estándar están declaradas en el espacio de nombres **std**

Funcionalidades. Aquí tienes una lista de las principales bibliotecas estándar y una breve descripción de cada una:

1. <iostream>
Descripción: Proporciona funcionalidades para la entrada y salida estándar.
Clases y funciones clave: std::cin, std::cout, std::cerr, std::clog.
2. <fstream>
Descripción: Facilita la manipulación de archivos, tanto para lectura como para escritura.
Clases y funciones clave: std::ifstream, std::ofstream, std::fstream.
3. <string>
Descripción: Proporciona la clase std::string para manejar cadenas de texto de manera eficiente.
Clases y funciones clave: std::string, std::getline, std::stoi, std::to_string.
4. <vector>
Descripción: Implementa el contenedor de secuencia std::vector, que es una estructura de datos dinámica.
Clases y funciones clave: std::vector, std::vector::push_back, std::vector::pop_back.
5. <list>
Descripción: Proporciona una lista doblemente enlazada a través del contenedor std::list.
Clases y funciones clave: std::list, std::list::push_back, std::list::pop_front.
6. <deque>
Descripción: Implementa un contenedor de secuencia de acceso rápido por ambos extremos a través de std::deque.
Clases y funciones clave: std::deque, std::deque::push_front, std::deque::pop_back.
7. <set>
Descripción: Proporciona un contenedor que almacena elementos únicos en un orden específico mediante std::set.
Clases y funciones clave: std::set, std::multiset.
8. <map>
Descripción: Implementa un contenedor de pares clave-valor con orden específico a través de std::map.
Clases y funciones clave: std::map, std::multimap.
9. <unordered_set>
Descripción: Proporciona un contenedor para almacenar elementos únicos sin un orden específico mediante std::unordered_set.
Clases y funciones clave: std::unordered_set, std::unordered_multiset.
10. <unordered_map>
Descripción: Implementa un contenedor de pares clave-valor sin un orden específico mediante std::unordered_map.
Clases y funciones clave: std::unordered_map, std::unordered_multimap.

11. <algorithm>

Descripción: Proporciona una serie de algoritmos genéricos para manipular contenedores, como búsqueda, ordenación y transformación.

Clases y funciones clave: std::sort, std::find, std::transform.

12. <numeric>

Descripción: Ofrece funciones para realizar operaciones numéricas, como la suma y el producto de elementos.

Clases y funciones clave: std::accumulate, std::inner_product.

13. <utility>

Descripción: Proporciona utilidades varias, como pares y funciones de intercambio.

Clases y funciones clave: std::pair, std::make_pair, std::swap.

14. <thread>

Descripción: Facilita la creación y manejo de hilos para programación concurrente.

Clases y funciones clave: std::thread, std::mutex, std::lock_guard.

15. <chrono>

Descripción: Ofrece funcionalidades para manejar el tiempo y las duraciones.

Clases y funciones clave: std::chrono::steady_clock, std::chrono::duration.

16. <exception>

Descripción: Proporciona la base para el manejo de excepciones.

Clases y funciones clave: std::exception, std::runtime_error, std::logic_error.

17. <memory>

Descripción: Contiene herramientas para la gestión dinámica de memoria y punteros inteligentes.

Clases y funciones clave: std::unique_ptr, std::shared_ptr, std::weak_ptr.

18. <iterator>

Descripción: Proporciona funcionalidades para trabajar con iteradores de manera más general.

Clases y funciones clave: std::iterator, std::begin, std::end.

19. <typeinfo>

Descripción: Ofrece funcionalidades relacionadas con la obtención de información de tipo en tiempo de ejecución.

Clases y funciones clave: std::type_info, typeid.

20. <cassert>

Descripción: Proporciona macros para la verificación de condiciones en tiempo de ejecución.

Clases y funciones clave: assert.

LINUX

Explicación detallada de la biblioteca <unistd.h> en C++

La biblioteca <unistd.h> es un **header** de Unix/Linux que proporciona acceso a funciones del sistema operativo relacionadas con llamadas al sistema (syscalls). Estas funciones permiten realizar operaciones básicas como manipulación de archivos, control de procesos, gestión de la memoria y comunicación entre procesos.

Temas que cubriremos:

1. **¿Qué es <unistd.h>?**
2. **¿Para qué se usa y dónde se emplea?**
3. **Principales funciones disponibles en <unistd.h>**
4. **Ejemplos prácticos de uso en Linux**
5. **Consideraciones importantes**

¿Qué es <unistd.h>?

<unistd.h> es un archivo de cabecera que contiene declaraciones de funciones del sistema operativo UNIX y Linux. No está disponible en Windows, ya que Windows usa otras bibliotecas como <windows.h> para funcionalidades similares.

Significado de "unistd"

El nombre **unistd** proviene de "**UNIX standard**" y proporciona acceso a las funciones del sistema operativo, principalmente a nivel de llamadas del sistema (**syscalls**).

¿Para qué se usa y dónde se emplea?

Se usa principalmente en sistemas **Unix-like** (Linux, macOS, BSD) para tareas como: Manejo de procesos

- Manipulación de archivos y directorios
- Control del entorno del usuario
- Comunicación entre procesos
- Gestión de memoria

Es comúnmente usada en **programación de sistemas, scripts en C++, programación de redes y desarrollo de software a bajo nivel**.

No es compatible con Windows de forma nativa. Si necesitas portar código a Windows, puedes usar la biblioteca **POSIX para Windows (Cygwin o MinGW)** o buscar funciones equivalentes en <windows.h>.

Principales funciones disponibles en <unistd.h>

Aquí te dejo una lista de las funciones más importantes junto con una breve explicación:

Función	Descripción
`access()`	Verifica permisos de un archivo o directorio
`chdir()`	Cambia el directorio de trabajo
`close()`	Cierra un descriptor de archivo
`dup()` / `dup2()`	Duplica un descriptor de archivo
`exec()`	Reemplaza el proceso actual con otro proceso
`fork()`	Crea un nuevo proceso hijo
`getcwd()`	Obtiene el directorio de trabajo actual
`getpid()` / `getppid()`	Obtiene el ID del proceso y del padre
`isatty()`	Verifica si un descriptor de archivo es una terminal
`link()` / `unlink()`	Crea o elimina enlaces a archivos
`lseek()`	Mueve el puntero de un archivo abierto
`pipe()`	Crea un canal de comunicación entre procesos
`read()`	Lee datos de un descriptor de archivo
`write()`	Escribe datos en un descriptor de archivo
`usleep()`	Suspende la ejecución en microsegundos
`sysconf()`	Obtiene información del sistema
`_exit()`	Termina un proceso inmediatamente

Ejemplos prácticos de uso en Linux

1. Pausar la ejecución del programa (sleep)

```
#include <iostream>
#include <unistd.h> // Para usar sleep()

int main() {
    std::cout << "Esperando 3 segundos...\n";
    sleep(3);  // Pausa el programa por 3 segundos
    std::cout << "¡Tiempo completado!\n";
    return 0;
}
```

2. Obtener el directorio actual (getcwd)

```
#include <iostream>
#include <unistd.h>
#include <limits.h>

int main() {
    char directorio[PATH_MAX];

    if (getcwd(directorio, sizeof(directorio)) != NULL) {
        std::cout << "Directorio actual: " << directorio << std::endl;
    } else {
        perror("getcwd() error");
    }

    return 0;
}
```

3. Crear un proceso hijo con fork()

```
#include <iostream>
#include <unistd.h>

int main() {
```

```
    pid_t pid = fork(); // Crea un proceso hijo

    if (pid < 0) {
        std::cerr << "Error al crear proceso" << std::endl;
    } else if (pid == 0) {
        std::cout << "Proceso hijo ejecutándose, PID: " << getpid() << std::endl;
    } else {
        std::cout << "Proceso padre ejecutándose, PID: " << getpid() << std::endl;
    }

    return 0;
}
```

fork() genera un proceso hijo duplicado del proceso actual.

4. Ejecutar otro programa con exec()

```
#include <iostream>
#include <unistd.h>

int main() {
    std::cout << "Ejecutando ls en la terminal..." << std::endl;
    execl("/bin/ls", "ls", "-l", NULL);
    std::cout << "Esto no se imprimirá si exec() tiene éxito" << std::endl;
    return 0;
}
```

execl() reemplaza el proceso actual con otro proceso, en este caso ls -l.

5. Crear una tubería (pipe) para comunicar procesos

```
#include <iostream>
#include <unistd.h>
#include <cstring>

int main() {
    int fd[2]; // Array para la tubería (fd[0] = lectura, fd[1] = escritura)
    pipe(fd); // Crear la tubería

    pid_t pid = fork(); // Crear un proceso hijo

    if (pid == 0) {
        // Código del hijo
        close(fd[0]); // Cerrar lectura
        char mensaje[] = "Hola desde el proceso hijo";
        write(fd[1], mensaje, strlen(mensaje) + 1);
        close(fd[1]); // Cerrar escritura
    } else {
        // Código del padre
        close(fd[1]); // Cerrar escritura
        char buffer[100];
        read(fd[0], buffer, sizeof(buffer));
        std::cout << "Mensaje recibido: " << buffer << std::endl;
        close(fd[0]); // Cerrar lectura
    }

    return 0;
}
```

pipe() permite la comunicación entre procesos.

Consideraciones importantes

<unistd.h> es exclusivo para **sistemas UNIX y Linux**
Algunas funciones pueden no estar disponibles en todas las distribuciones
Para programas multiplataforma, usar alternativas como <windows.h> en Windows
Se usa en **programación de sistemas, redes y administración de procesos**

¿Qué es <limits.h>?

Es una biblioteca estándar de **C/C++** que define constantes relacionadas con los límites de los tipos de datos primitivos, como el valor máximo y mínimo de int, char, long, short, etc.

Es parte del **estándar ANSI C** y **POSIX**, por lo que está disponible en **Windows**, **Linux**, **macOS** y otros sistemas operativos.

Ejemplo de uso

```
#include <iostream>
#include <limits.h>

int main() {
    std::cout << "Valor máximo de int: " << INT_MAX << std::endl;
    std::cout << "Valor mínimo de int: " << INT_MIN << std::endl;
    std::cout << "Valor máximo de char: " << CHAR_MAX << std::endl;
    std::cout << "Valor mínimo de char: " << CHAR_MIN << std::endl;
    std::cout << "Valor máximo de long: " << LONG_MAX << std::endl;
    std::cout << "Valor mínimo de long: " << LONG_MIN << std::endl;
    return 0;
}
```

Salida esperada (depende del sistema y compilador):

```
Valor máximo de int: 2147483647
Valor mínimo de int: -2147483648
Valor máximo de char: 127
Valor mínimo de char: -128
Valor máximo de long: 9223372036854775807
Valor mínimo de long: -9223372036854775808
```

Límites más comunes en <limits.h>

Constante	Descripción
`CHAR_BIT`	Número de bits en un `char` (normalmente 8)
`SCHAR_MIN` / `SCHAR_MAX`	Valor mínimo/máximo de `signed char`
`UCHAR_MAX`	Valor máximo de `unsigned char`
`SHRT_MIN` / `SHRT_MAX`	Valor mínimo/máximo de `short`
`USHRT_MAX`	Valor máximo de `unsigned short`
`INT_MIN` / `INT_MAX`	Valor mínimo/máximo de `int`
`UINT_MAX`	Valor máximo de `unsigned int`
`LONG_MIN` / `LONG_MAX`	Valor mínimo/máximo de `long`
`ULONG_MAX`	Valor máximo de `unsigned long`

¿Se debe usar <limits.h> en C++ o <limits>?

En C, usa <limits.h>
En C++, se recomienda usar <limits> de la STL:

```
#include <iostream>
#include <limits>

int main() {
    std::cout << "Valor máximo de int: " << std::numeric_limits<int>::max() << std::endl;
    std::cout << "Valor mínimo de int: " << std::numeric_limits<int>::min() << std::endl;
    return 0;
}
```

std::numeric_limits<> ofrece más flexibilidad y es más seguro en C++.