

Algoritmos 1

Apéndice 01 - Introducción al Lenguaje C++
Primer Semestre, 2026

Contents

1.1	Introducción	4
1.2	¿Cómo instalo un compilador de C++?	4
1.2.1	Crear el contenedor	4
1.3	Hello World en C++	5
1.4	Corriendo programas en C++	6
1.5	Repaso de C y C++	7
1.5.1	Compilación y ejecución	7
1.5.2	Sintaxis básica	8
1.5.3	Variables y tipos de datos	8
1.5.4	Operadores	9
1.5.5	Estructuras de control	10
1.5.6	Punteros	11
1.5.7	Funciones	11
1.5.8	Manejo de memoria	13
1.5.9	Matrices	14
1.5.10	Entrada y salida de datos en C	15
1.5.11	Entrada y salida de datos en C++	16
1.5.12	Estructuras	18
1.5.13	Cadenas en C	19
1.5.14	Cadenas en C++	21
1.6	Scopes	21
1.7	Errores	23
1.7.1	Errores de compilación	23
1.7.2	Errores de ejecución	24
1.8	C++	25
1.8.1	Contenedores de la biblioteca estándar de C++	25
1.8.2	Iteradores	27
1.8.3	Vector	29
1.8.4	Set	30
1.8.5	Unordered Set	30
1.8.6	Map	31
1.8.7	Unordered Map	31
1.8.8	Pair	32
1.8.9	Tuple	33
1.8.10	Composición de Contenedores	34

¡Bienvenidos a Algoritmos 1!

Esta es una guía adicional para que tengas como referencia del lenguaje C++ que es lo que vamos a estar utilizando durante la cursada. Si bien no es necesario que sepas C++, es importante que tengas una base sólida para que puedas entender los errores y poder resolver los ejercicios que se van a plantear durante la materia.

El objetivo de esta guía es que puedas:

- **Aprender a compilar y ejecutar código en C / C++**
- **Aprender y repasar los conceptos básicos de C / C++** que son necesarios para la materia.
- **Tener una referencia rápida de la sintaxis de C / C++** para poder consultar cuando tengas dudas sobre cómo escribir algo en C / C++.

Si aún así te quedan dudas sobre cómo escribir alguna cosa puntual **¡Utilizá las herramientas para consultar!**

1.1 Introducción

Para poder trabajar en la materia vas a tener que tener instalado un compilador C++ para poder probar tus propios ejercicios y problemas. Si sólo vas a correr los ejemplos y problemas vistos en clase podés utilizar simplemente la plataforma de programación competitiva de la universidad en: UP Competitive Programming Platform.

1.2 ¿Cómo instalo un compilador de C++?

La forma más sencilla es utilizando docker. Docker Desktop es una aplicación que permite crear y administrar contenedores de Docker en tu computadora. Un contenedor de Docker es una instancia de una imagen de docker (muy similar a una máquina virtual) que se ejecuta de manera aislada del sistema operativo subyacente, lo que permite ejecutar aplicaciones y servicios de manera consistente en diferentes entornos.

La ventaja de utilizar docker es que no estás instalando nada más que docker (que es muy utilizada en entornos profesionales, con lo cual seguramente lo sigas usando), y luego todo lo que se instala dentro del contenedor es totalmente efímero y si un día querés eliminarlo, simplemente eliminás el contenedor y listo, sin tener que preocuparte por desinstalar nada ni por conflictos de versiones de software.

1.2.1 Crear el contenedor

La primera cosa que tenés que hacer es crear el contenedor de docker con el compilador que vamos a utilizar en la materia. Para eso tenés que ir al directorio `docker/` que se encuentra en este repositorio y ejecutar:

```
docker build -t algoritmos_1 .
```

Esto va a crear una imagen de docker con el nombre `algoritmos_1` que tiene instalado el compilador de C++ y todas las herramientas necesarias para poder compilar y ejecutar código en C++.

¡Con esto ya podés correr los ejemplos y ejercicios de la materia!

1.3 Hello World en C++

En general uno asocia C++ con la programación orientada a objetos, sin embargo en esta materia, vamos a utilizar el lenguaje C++ como un C potenciado. Es decir, vamos a utilizar la sintaxis de C++ pero sin utilizar las características de la programación orientada a objetos, pero vamos a tomar ventaja de las bibliotecas de alto nivel que ofrece, como las estructuras de datos ya incluidas y sus algoritmos para poder tener un código que se ejecute en alto nivel que, además, pueda compilar C que es algo que verán en el resto de la carrera.

El código de ejemplo para imprimir un Hello World es un poco diferente en C++ que en C, porque la interfaz de entrada y salida de datos es diferente. En C++ se utiliza la biblioteca `iostream` para la entrada y salida:

```
1 #include <iostream> // esto es lo mismo que en C, pero sin la extensión .h
2
3 int main() {
4     // std::cout es el objeto de salida estándar en C++, y std::endl es un
5     // salto de línea
6     std::cout << "Hola mundo.... desde C++!!!" << std::endl;
7     return 0;
}
```

Listing 1: Código de ejemplo Hello World en C++

En general el estudio de Algoritmos se puede hacer en cualquier lenguaje, pero C es un lenguaje de muy bajo nivel que no tiene en su biblioteca estándar algunas funcionalidades que necesitamos para poder estudiar cierto tipo de algoritmos.

1.4 Corriendo programas en C++

Para ejecutar programas en C++ pueden utilizar una instalación local, pueden usar ‘docker’ como ya vimos en la sección anterior 1.2.1, pueden utilizar la plataforma de programación competitiva de la universidad, o pueden utilizar alguna plataforma como OneCompiler o OnlineGDB.

En cualquier caso si van a utilizar un contenedor de docker o una instalación local de g++ para compilar el código, el comando para compilar un programa en C++ es el siguiente:

```
g++ -Wall -o programa programa.cc
```

Donde `-Wall` es un flag que le indica al compilador que muestre todas las advertencias, `-o programa` es el flag que le indica al compilador que el archivo ejecutable se llame `programa`, y `programa.cc` es el archivo de código fuente que queremos compilar.

1.5 Repaso de C y C++

ATENCIÓN

En esta sección vamos a repasar los conceptos básicos de C++, algunas cosas no van a estar explicadas porque el objetivo de esta materia no es enseñar C++, con lo cual si alguna de las cosas que están escritas acá no se entiende, podrás preguntar en clase o investigar un poco.

Los conceptos IMPORTANTES van a estar explicados, pero la sintaxis del lenguaje no va a estar explicada en detalle, porque el libro de C++ en sí mismo es un libro de más de 600 páginas y no es obligatorio para la materia, simplemente aprendé la sintaxis.

Es cierto que escribir `print "hola"` puede resultar más ameno que escribir `std::cout << "hola" << std::endl`, pero en definitiva es una cuestión exclusivamente de sintaxis.

Igual iremos explicando lo que significa cada parte del código, pero lo importante será enfocarnos en algoritmos.

1.5.1 Compilación y ejecución

Este apartado está repetido en la práctica para que puedas ver cómo se compila y ejecuta un programa en C y C++. Supongamos que tenés el siguiente código en un archivo llamado `cuadrado.cc`:

Versión en C

```

1 #include <stdio.h>
2
3 int main() {
4     int n;
5
6     scanf("%d", &n); // Lee un número desde la entrada estándar
7     printf("%d\n", n*n); // Imprime el cuadrado del número
8
9     return 0;
10 }
```

Listing 2: Cuadrado de un número

Versión en C++

```

1 #include <iostream>
2
3 using namespace std; // Esto no es obligatorio en versiones nuevas de C++
4
5 int main() {
6     int n;
7
8     cin >> n; // Lee un número desde la entrada estándar (similar a scanf en C)
9     cout << n*n << endl; // Imprime el cuadrado del número (similar a printf en C)
10
11     return 0;
12 }
```

Listing 3: Cuadrado de un número

```
g++ -Wall -o cuadrado cuadrado.cc
./cuadrado
5
25
```

La idea es que cada vez que ejecutemos este programa, se quedará esperando a que ingresemos un número por pantalla y luego imprimirá el cuadrado de ese número.

1.5.2 Sintaxis básica

El lenguaje C++ es un lenguaje de programación de propósito general, que se caracteriza por su simplicidad y eficiencia. Un programa en C++ se compone de una o más funciones, siendo la función `main` la función principal que se ejecuta al iniciar el programa. La sintaxis básica es la siguiente:

```
1 // #include sirve para incluir bibliotecas
2 #include <iostream> // Biblioteca para entrada y salida de datos
3 #include <algorithm> // Biblioteca para algoritmos predefinidos
4 #include <vector> // Biblioteca para arrays dinámicos (vectores)
5 // etc.
6
7 // main es la función principal del programa que DEBE estar presente
8 // argc y argv son los argumentos de la linea de comandos
9 int main(int argc, char *argv[]) {
10     cout << "Hola mundo.... desde C++!!!" << endl; // Imprime un mensaje en
11     // la consola
12     return 0;
13 }
```

Listing 4: Sintaxis básica de un programa en C++

1.5.3 Variables y tipos de datos

En C++, las variables se utilizan para almacenar datos y deben ser declaradas antes de ser utilizadas. Los tipos de datos más comunes en C son:

- `int`: Enteros (números enteros).
- `float`: Números de punto flotante (decimales).
- `double`: Números de punto flotante de doble precisión.
- `char`: Caracteres individuales.
- `void`: Tipo de dato vacío, utilizado para funciones que no retornan un valor.

Además se pueden utilizar modificadores de tipo como `unsigned` para enteros sin signo, o `long` para enteros de mayor tamaño. Por ejemplo, `unsigned int` es un entero sin signo que puede almacenar valores desde 0 hasta 2^{32-1} , mientras que `long int` es un entero de mayor tamaño que puede almacenar valores desde -2^{63} hasta 2^{63-1} .

1.5.4 Operadores

C cuenta con una variedad de operadores que se utilizan para realizar operaciones aritméticas, lógicas y de comparación. Los operadores más comunes son:

- Aritméticos: +, -, *, /, %.
- Logicos: && (AND), || (OR), ! (NOT).
- De comparacion: ==, !=, <, >, <=, >=.
- Asignación: =, +=, -=, *=, /=, %=.
- Bit a bit: &, |, ^, ~, <<, >>.

Los operadores aritméticos no requieren una explicación adicional, con excepción del operador % que es el operador de módulo que nos devuelve el resto de una división entera. Por ejemplo, si tenemos $5 \% 2$ el resultado es 1 porque $5 / 2$ da como resultado 2 y el resto es 1.

Los operadores lógicos se utilizan para combinar expresiones que devuelven un valor de verdad (**true** o **false**. En C y C++: **NULL**, 0 o **false**, todo lo demás se considera **true**.

Los operadores de comparación se utilizan para comparar dos valores y devolver un valor de verdad. Con lo cual si tenemos una expresión como $5 > 3$ el resultado es **true**, mientras que si tenemos $5 < 3$ el resultado es **false** y estas expresiones se pueden combinar con los operadores lógicos para formar expresiones más complejas, por ejemplo $(a > 3) \&\& (b < 5)$.

Los operadores de asignación se utilizan para asignar un valor a una variable, por ejemplo $x = 5$ asigna el valor 5 a la variable x. Además hay operadores de asignación compuestos ($+=$, $-=$, $*=$, $/=$, $%=$) son una forma abreviada de escribir una operación de asignación combinada con una operación aritmética. Por ejemplo, $x += 5$ es equivalente a $x = x + 5$.

En una operación de asignación existe el **lvalue** y el **rvalue**, donde el **lvalue** es la variable donde se va a almacenar el resultado de la operación y el **rvalue** es el valor que se va a asignar. Esta es la razón por la cual no se puede hacer una asignación como $5 = x$ ya que 5 no es un **lvalue**, lo que significa que no se puede asignar un valor a él.

Además existen operadores bit a bit se utilizan para realizar operaciones a nivel de bits, estos son importantes para poder manipular datos a nivel de bits, lo cual es útil para la optimización de memoria.

- **&**: $5 \& 3$ da como resultado 1 porque en binario 5 es 101 y 3 es 011, y al hacer la operación AND bit a bit, el resultado es 001.
- **|**: $5 | 3$ da como resultado 7 porque en binario 5 es 101 y 3 es 011, y al hacer la operación OR bit a bit, el resultado es 111.
- **^**: $5 ^ 3$ da como resultado 6 porque en binario 5 es 101 y 3 es 011, y al hacer la operación XOR bit a bit, el resultado es 110.
- **~**: \sim es el operador de complemento a uno, que invierte todos los bits de un número. Por ejemplo, ~ 5 da como resultado -6.
- **<<**: $5 << 1$ desplaza los bits de 5 una posición a la izquierda, lo que equivale a multiplicar por 2, dando como resultado 10.

- `>>`: `5 >> 1` desplaza los bits de 5 una posición a la derecha, lo que equivale a la división entera por 2, dando como resultado 2.

1.5.5 Estructuras de control

C cuenta con estructuras de control que permiten tomar decisiones y repetir bloques de código. Las estructuras más comunes son:

- Condicionales: `if`, `else if`, `else`
- Bucles: `for`, `while`
- Saltos: `break`, `continue`, `return`.

Ejemplos:

```

1 #include <stdio.h>
2
3 int main() {
4     // Bucle for
5     for (int x = -10; x < 11; x++) {
6         if((x > 5) && (x % 3 == 0)) {
7             printf("x=%d es mayor a 5 y multiplo de 3\n", x);
8         }
9
10    // Condicional if
11    if (x > 0) {
12        printf("x=%d es positivo\n", x);
13    } else if (x < 0) {
14        printf("x=%d es negativo\n", x);
15    } else {
16        printf("x=%d es cero\n", x);
17    }
18 }
19
20 // Bucle while
21 int j = 0;
22 while (j < 5) {
23     printf("Iteracion %d\n", j);
24     j++;
25 }
26
27 return 0;
28 }
```

Listing 5: Estructuras de Control

El `if` es una estructura de control que permite ejecutar o no un bloque de código (como veremos en la sección Scopes 1.6) dependiendo de si se cumple o no una condición, pueden haber múltiples condiciones utilizando `else if`, y un bloque de código que se ejecuta si ninguna de las condiciones se cumple utilizando `else`.

El `while` es una estructura de control que permite repetir un bloque de código mientras se cumpla una condición, es decir, se chequea la condición al inicio de la ejecución del bloque,

y mientras esa condición sea verdadera, el bloque de código se ejecutará, y luego se volverá a chequear la condición.

El **for** es una estructura de control que permite repetir un bloque de código un número determinado de veces donde se chequea una condición al inicio de la ejecución del bloque. Minetras que esa condición sea verdadera, el bloque de código se ejecutará, y luego se ejecutará una expresión de actualización que generalmente se utiliza para modificar la variable de control del bucle. Podemos decir que el **for** es una versión más compacta del **while** ya que permite inicializar variables, chequear la condición y actualizar la variable en una sola línea de código.

A medida que programen verán cuál de las estructuras de control es más conveniente utilizar dependiendo del caso.

1.5.6 Punteros

Los punteros son una característica poderosa de C y C++ que permite manipular direcciones de memoria directamente. Un puntero es una variable que almacena la dirección de memoria de otra variable. Se declaran utilizando el operador ***** y se accede al valor al que apuntan utilizando el operador **&**. Por ejemplo:

```

1 #include <stdio.h>
2
3 int main() {
4     int x = 10; // Declaracion de una variable entera
5     int *p = &x; // Declaracion de un puntero que apunta a la direccion de x
6
7     printf("Valor de x: %d\n", x); // Imprime el valor de x
8     printf("Direccion de x: %p\n", (void*)&x); // Imprime la direccion de x
9     printf("Valor al que apunta p: %d\n", *p); // Imprime el valor al que
10    apunta p (valor de x)
11    printf("Direccion almacenada en p: %p\n", (void*)p); // Imprime la
12    direccion almacenada en p
13 }
```

Listing 6: Ejemplo de punteros en C

1.5.7 Funciones

Las funciones en C++ son bloques de código que realizan una tarea específica y pueden ser reutilizadas en diferentes partes del programa. Se declaran utilizando la siguiente sintaxis:

```

1 int sumar_enteros(int a, int b) {
2     return a + b; // Retorna la suma de a y b
3 }
```

Listing 7: Declaración de una función en C

Para llamar a una función, simplemente se utiliza su nombre seguido de los argumentos entre paréntesis:

```

1 #include <stdio.h>
2
```

```

3 int sumar_enteros(int a, int b) {
4     return a + b;
5 }
6
7 int main() {
8     int resultado = sumar_enteros(5, 10); // Llama a la funcion sumar_enteros
9     printf("Resultado: %d\n", resultado); // Imprime el resultado
10    return 0;
11 }
```

Listing 8: Llamada a una función en C

Hay que tener algunas consideraciones cuando se llaman a funciones, porque en C y C++ los argumentos (o parámetros) de las funciones se pasan **por valor**, lo que significa que se crea una copia de los argumentos al llamar a la función. Esto tiene una ventaja y una desventaja. La ventaja es que no se pueden modificar los argumentos originales desde la función, lo que muchas veces es útil cuando no queremos que la función modifique los datos que le pasamos. La desventaja es que si los datos que vamos a pasar son muy grandes, la llamada a esa función se vuelve extremadamente ineficiente. Esto es un problema super común y hay que tenerlo en cuenta.

En C, la solución a este problema es utilizar punteros para pasar los argumentos que tienen un tamaño grande o que se desea modificar. Por ejemplo supongamos que queremos pasar un valor que queremos modificar en una función:

```

1 #include <stdio.h>
2
3 void incrementar(int *x) { // El argumento x se pasa como puntero
4     (*x)++; // Incrementa el valor al que apunta x
5 }
6
7 int main() {
8     int numero = 5;
9     incrementar(&numero); // Llama a la funcion incrementar pasando la
10    direccion de numero
11    printf("Numero incrementado: %d\n", numero); // Imprime el numero
12    incrementado
13    return 0;
14 }
```

Listing 9: Ejemplo de paso por puntero en C

Si se fijan, la forma de llamar a la función `incrementar` es igual a la sintaxis utilizada en la función `printf`, porque en ambos casos se está pasando la dirección de memoria de la variable `numero` para que la función pueda modificar su valor original.

En C++ se puede hacer lo mismo, pero la sintaxis es mucho más sencilla gracias que tiene una sintaxis especial para pasar argumentos **por referencia** que es similar a las direcciones de memoria. La diferencia es que no es necesario dereferenciarla para poder usarla dentro de la función como sí ocurre dentro de los punteros.

¿Cómo sería la sintaxis para pasar un argumento por referencia en C++? Simplemente se utiliza el operador `&` en la declaración de la función para indicar que el argumento se pasa por referencia, y luego se puede modificar el valor original desde la función. Por ejemplo:

```

1 #include <iostream>
2
3 using namespace std;
4
5 void incrementar(int &x) { // El argumento x se pasa por referencia
6     x++; // Incrementa el valor de x
7 }
8
9 int main() {
10     int numero = 5;
11     incrementar(numero); // Llama a la funcion incrementar
12     cout << "Numero incrementado: " << numero << endl; // Imprime el numero
13     incrementado
14     return 0;
}

```

Listing 10: Ejemplo de paso por referencia en C++

En definitiva es muy parecido a pasar un puntero en C, con la diferencia de que en C++ no es necesario utilizar el operador * para declarar un puntero, ni el operador & para acceder al valor al que apunta el puntero, sino que se utiliza el operador & para indicar que el argumento se pasa por referencia, y luego se puede acceder al valor original directamente desde la función.

1.5.8 Manejo de memoria

En C, el manejo de memoria básico de C se realiza utilizando las funciones `malloc` y `free` de la biblioteca `stdlib.h`. La función `malloc` se utiliza para asignar memoria dinámica y la función `free` se utiliza para liberar la memoria asignada. Por ejemplo:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int *arr = (int*)malloc(5 * sizeof(int)); // Asigna memoria para un
6     // arreglo de 5 enteros
7     if (arr == NULL) {
8         printf("Error al asignar memoria\n");
9         return 1; // Sale del programa si no se pudo asignar memoria
10    }
11
12    for (int i = 0; i < 5; i++) {
13        arr[i] = i * 10; // Inicializa el arreglo
14    }
15
16    for (int i = 0; i < 5; i++) {
17        printf("%d ", arr[i]); // Imprime el arreglo
18    }
19    printf("\n");
20
21    free(arr); // Libera la memoria asignada
22    return 0;
}

```

Listing 11: Manejo de memoria en C

Este tipo de cosas pueden resultar un poco complicadas al principio porque ese necesario tener cuidado de no irnos de los límites de memoria asignada para evitar incurrir en errores de memoria (*segmentation fault*), y también es necesario recordar liberar la memoria asignada para evitar fugas de memoria (*memory leaks*). Los *memory leaks* ocurren cuando se asigna memoria dinámica pero no se libera, lo que puede llevar a que el programa consuma cada vez más memoria y eventualmente se quede sin memoria disponible, lo que puede causar que el programa se bloquee o se comporte de manera inesperada, a esto se lo conoce muchas veces como OOM (*Out of Memory*).

Sin embargo en C++ tenemos la ventaja de que el manejo de memoria es mucho más sencillo gracias a la utilización de contenedores de la biblioteca estándar como `std::vector` que se encargan automáticamente de asignar y liberar memoria, lo que nos permite centrarnos en la lógica de nuestro programa sin tener que preocuparnos por el manejo de memoria como veremos en la sección específica de C++ 1.8.

Sin embargo para algunos ejercicios utilizaremos memoria dinámica en C, por lo cual es importante que entiendan cómo funciona el manejo de memoria en C, y cómo se puede utilizar para crear estructuras de datos dinámicas como listas enlazadas, árboles, grafos, etc.

1.5.9 Matrices

Al pedir memoria con `malloc` C nos retorna un puntero a la memoria asignada, y esta memoria asignada es contigua, con lo cual podemos utilizar punteros o subíndices para acceder a los elementos de la memoria asignada. Por ejemplo si tenemos un vector de tamaño 100 y queremos acceder al elemento 47 podemos hacerlo de la siguientes maneras:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int *vector = (int*)malloc(100 * sizeof(int)); // Asigna memoria para un
6     // vector de 100 enteros
7     if (vector == NULL) {
8         printf("Error al asignar memoria\n");
9         return 1; // Sale del programa si no se pudo asignar memoria
10    }
11
12    // Acceso al elemento 47 del vector usando subindice
13    vector[47] = 42;
14    printf("Elemento en la posicion 47: %d\n", vector[47]);
15
16    *(vector + 47) += 1; // Acceso al elemento 47 del vector usando puntero
17    printf("Elemento en la posicion 47: %d\n", *(vector + 47));
18
19    free(vector); // Libera la memoria asignada
20}

```

Listing 12: Acceso a elementos de un vector en C

Supongamos que ahora tenés que trabajar con una matriz de 100x200 enteros (20.000 elementos), en este caso lo que se puede hacer es calcular la dirección de memoria del elemento en la fila `fila` y la columna `col` de la matriz haciendo lo siguiente:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int filas = 100;
6     int columnas = 200;
7     int *matriz = (int*)malloc(filas * columnas * sizeof(int)); // Asigna
        memoria para una matriz de 100x100 enteros
8     if (matriz == NULL) {
9         printf("Error al asignar memoria\n");
10    return 1; // Sale del programa si no se pudo asignar memoria
11 }
12
13 int fila = 47;
14 int col = 42;
15
16 // Acceso al elemento en la fila y columna especificadas
17 int posicion_memoria = fila * columnas + col; // Calcula la posicion en
        memoria
18 matriz[posicion_memoria] = 1234; // Acceso usando subindice
19 printf("Elemento en la posicion [%d][%d]: %d\n", fila, col, matriz[
        posicion_memoria]);
20
21 *(matriz + posicion_memoria) += 1; // Acceso usando puntero
22 printf("Elemento en la posicion [%d][%d]: %d\n", fila, col, *(matriz +
        posicion_memoria));
23
24 free(matriz); // Libera la memoria asignada
25 return 0;
26 }
```

Listing 13: Acceso a elementos de una matriz en C

1.5.10 Entrada y salida de datos en C

C proporciona funciones para la entrada y salida de datos a través de la biblioteca `stdio.h`. Las funciones más comunes son:

- `printf`: Para imprimir datos en la consola.
- `scanf`: Para leer datos desde la entrada estándar (teclado).
- `getchar`: Para leer un carácter desde la entrada estándar.
- `putchar`: Para imprimir un carácter en la salida estándar.
- `fgets`: Para leer una línea completa desde la entrada estándar.
- `fputs`: Para imprimir una línea completa en la salida estándar.

Estas funciones son ideales para interactuar con el usuario, pero también funcionan para obtener datos de STDIN y enviarlo a STDOUT, que es lo que se puede utilizar luego para redirigir la entrada y salida de datos desde y hacia archivos.

Supongamos que tenemos un matriz de `filas x columnas` enteros y queremos leer los datos desde la entrada estándar, podemos crear un archivo así:

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```

Y luego leerlo desde un programa en C de la siguiente manera:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int filas, columnas;
6     scanf("%d %d", &filas, &columnas); // Lee las dimensiones de la matriz
7
8     int **matriz = (int**)malloc(filas * sizeof(int*)); // Asigna memoria
9     para las filas
10    for (int i = 0; i < filas; i++) {
11        matriz[i] = (int*)malloc(columnas * sizeof(int)); // Asigna memoria
12        para las columnas
13    }
14
15    // Lee los datos de la matriz
16    for (int i = 0; i < filas; i++) {
17        for (int j = 0; j < columnas; j++) {
18            scanf("%d", &matriz[i][j]);
19        }
20
21    // Imprime la matriz
22    printf("Matriz:\n");
23    for (int i = 0; i < filas; i++) {
24        for (int j = 0; j < columnas; j++) {
25            printf("%d ", matriz[i][j]);
26        }
27        printf("\n");
28    }
29
30    return 0;
}
```

Listing 14: Lectura de datos desde la entrada estándar en C

1.5.11 Entrada y salida de datos en C++

En C++, la entrada y salida de datos se realiza a través de la biblioteca `iostream`, ya que el C++ utiliza el concepto de flujos para la entrada y salida de datos. Los objetos más comunes para la entrada y salida de datos son:

- `std::cin`: Para leer datos desde la entrada estándar (teclado).

- `std::cout`: Para imprimir datos en la consola.

La sintaxis para la entrada y salida de datos en C++ es diferente a la de C, ya que se utilizan los operadores de inserción (`<<`) y extracción (`>>`) para enviar y recibir datos a través de los flujos. Podemos pensar esos operadores como flechas que indican la dirección del flujo de datos, por ejemplo `std::cout << "Hola"` indica que la cadena "Hola" se envía al flujo de salida estándar, mientras que `std::cin >> x` indica que el valor ingresado por el usuario se almacena en la variable `x`.

Como se puede ver no es necesario indicar la dirección de memoria de la variable al leer datos desde la entrada estándar, como se hace en C con `scanf`, ya que en C++ el operador de extracción (`>>`) se encarga de almacenar el valor ingresado directamente en la variable, lo que hace que la sintaxis sea más sencilla y fácil de entender.

Por ejemplo si quisieramos leer una matriz de `filas x columnas` enteros desde la entrada estándar en C++, podríamos hacerlo de la siguiente manera:

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     int filas, columnas;
7     cin >> filas >> columnas; // lee las dimensiones de la matriz
8
9     int **matriz = new int*[filas]; // asigna memoria para las filas
10    for (int i = 0; i < filas; i++) {
11        matriz[i] = new int[columnas]; // asigna memoria para las columnas
12    }
13
14    // lee los datos de la matriz
15    for (int i = 0; i < filas; i++) {
16        for (int j = 0; j < columnas; j++) {
17            cin >> matriz[i][j];
18        }
19    }
20
21    // imprime la matriz
22    cout << "matriz:" << endl;
23    for (int i = 0; i < filas; i++) {
24        for (int j = 0; j < columnas; j++) {
25            cout << matriz[i][j] << " ";
26        }
27        cout << endl;
28    }
29}

```

Listing 15: Lectura de datos desde la entrada estándar en C++

1.5.12 Estructuras

Las estructuras en C son una forma de agrupar diferentes tipos de datos bajo un mismo nombre. Se declaran utilizando la palabra clave **struct** y se pueden utilizar para crear tipos de datos personalizados. Por ejemplo:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Persona {
5     char nombre[50]; // Nombre de la persona
6     int edad; // Edad de la persona
7 };
8
9 int main() {
10     struct Persona* persona = (struct Persona*)malloc(sizeof(struct Persona))
11         ; // Asigna memoria para una estructura
12     if (persona == NULL) {
13         printf("Error al asignar memoria\n");
14         return 1; // Sale del programa si no se pudo asignar memoria
15     }
16     // Inicializa la estructura
17     sprintf(persona->nombre, sizeof(persona->nombre), "Juan Perez"); //
18         Asigna el nombre
19     persona->edad = 30; // Asigna la edad
20     printf("Nombre: %s, Edad: %d\n", persona->nombre, persona->edad); //
21         Imprime los datos de la persona
22     free(persona); // Libera la memoria asignada
23     return 0;
24 }
```

Listing 16: Declaración de una estructura en C

1.5.13 Cadenas en C

En C, las cadenas de caracteres se representan como arreglos de caracteres terminados en un carácter nulo (NULL). Es **muy** importante recordar que para ser una cadena válida, el último carácter debe ser el carácter nulo, ya que muchas de las funciones de manipulación de cadenas en C dependen de este carácter para determinar el final de la cadena.

El carácter nulo en C es el carácter con número ASCII 0, y se representa como 0 o NULL.

Si uno utiliza una cadena en C, el compilador automáticamente agrega el carácter nulo al final de la cadena, por lo que no es necesario agregarlo. Pero si estamos utilizando memoria dinámica para almacenar una cadena, debemos asegurarnos nosotros de que el último carácter sea el carácter nulo, para que las funciones de manipulación de cadenas funcionen bien.

Por ejemplo:

```

1 #include <stdio.h>
2 #include <string.h> // Incluye la biblioteca para manipulacion de cadenas
3
4 int main() {
5     char cadena[20]; // Declara un arreglo de 20 caracteres
6
7     cadena[0] = 'H'; // Asigna el primer caracter
8     cadena[1] = 'o'; // Asigna el segundo caracter
9     cadena[2] = 'l'; // Asigna el tercer caracter
10    cadena[3] = 'a'; // Asigna el cuarto caracter
11
12    printf("El largo de cadena es: %d\n", strlen(cadena)); // 'cadena' NO
13        termina en NULO!!!, CUIDADO!!
14
15    char* cadena2 = "Hola"; // Declara una cadena literal, termina en NULO
16        automaticamente
17    printf("El largo de cadena2 es: %d\n", strlen(cadena2)); // cadena2
18        termina en NULO
19
20    return 0;
21 }
```

Listing 17: Ejemplo de cadenas en C

La salida de este programa (puede) ser:

```
El largo de cadena es: 9
El largo de cadena2 es: 4
```

¿Por qué la cadena `cadena` tiene un largo de 9? Porque no termina en NULO, y por lo tanto la función `strlen` cuenta todos los caracteres hasta que encuentra un carácter NULL. Esto puede llevar a comportamientos inesperados, ya que si la cadena no termina en NULO, la función `strlen` seguirá contando hasta que encuentre un carácter NULL en memoria o hasta que se salga del rango de memoria asignada al programa y termine con un **Segmentation Fault**.

Para ello se utiliza la función `strncpy` que permite especificar el tamaño máximo de la cadena a contar, de esta forma evitamos tener problemas de memoria, ya que podemos especificar cuántos caracteres vamos a contar como máximo antes de que la función se salga del rango de memoria asignada al programa.

```
1 #include <stdio.h>
2 #include <string.h> // Incluye la biblioteca para manejo de cadenas
3
4 int main() {
5     char cadena[20]; // Declara un arreglo de 20 caracteres
6
7     cadena[0] = 'H'; // Asigna el primer caracter
8     cadena[1] = 'o'; // Asigna el segundo caracter
9     cadena[2] = 'l'; // Asigna el tercer caracter
10    cadena[3] = 'a'; // Asigna el cuarto caracter
11
12    printf("El largo de cadena es: %d\n", strlen(cadena, sizeof(cadena)));
13    // Ahora si termina en NULO!!!
14
15    return 0;
}
```

Listing 18: Ejemplo de cadenas en C con strlen

Importante: Esto NO elimina el problema de que la cadena no termine en NULL, para que sólo cuente 4 caracteres deberíamos haber agregado `cadena[4] = NULL;`, para hacer que nuestra cadena Hola termine en el carácter nulo.

1.5.14 Cadenas en C++

En C++, las cadenas de caracteres se pueden representar utilizando la clase `string` de la biblioteca estándar, que proporciona una forma más fácil y segura de manejar cadenas de caracteres en comparación con las cadenas en C. La clase `string` se encuentra en la biblioteca `<string>` y se puede utilizar de la siguiente manera:

```

1 #include <iostream>
2 #include <string> // Incluye la biblioteca para manejo de cadenas
3
4 using namespace std;
5
6 int main() {
7     string cadena = "Hola"; // Declara una cadena utilizando la clase string
8     cout << "La cadena es: " << cadena << endl; // Imprime la cadena
9     cout << "El largo de la cadena es: " << cadena.length() << endl; //
    Imprime el largo de la cadena
10
11 return 0;
12 }
```

Listing 19: Ejemplo de cadenas en C++

Vemos entonces que estas cadenas son mucho más fáciles de manejar que las cadenas en C ya que no hay que hacer ningún tipo de administración de la memoria ni preocuparse por el carácter nulo al final de la cadena o por el tamaño máximo de la cadena, ya que la clase `string` se encarga automáticamente de gestionar la memoria y el tamaño de la cadena. Además, la clase `string` proporciona una gran cantidad de funciones para manipular cadenas de caracteres, como concatenar, comparar, buscar, etc.

1.6 Scopes

En C y C++, el **scope** (o ámbito) de una variable se refiere a la región del código donde la variable es accesible. Existen diferentes tipos de scopes en C:

- **Scope global:** Las variables declaradas fuera de cualquier función tienen un scope global y son accesibles desde cualquier parte del programa.
- **Scope local:** Las variables declaradas dentro de una función tienen un scope local y sólo son accesibles dentro de esa función.
- **Scope de bloque:** Las variables declaradas dentro de un bloque (por ejemplo, dentro de llaves `{}`) tienen un scope limitado a ese bloque.

Saber utilizar los scopes correctamente es fundamental para evitar errores de acceso a variables, por ejemplo supongamos el siguiente código:

```

1 #include <stdio.h>
2
3 int x = 10;
4
5 void asignar_valor_a_x(int nuevo_x) {
```

```

6 int x = nuevo_x;
7 }
8
9 int main() {
10    int x = 30;
11    asignar_valor_a_x(20);
12    printf("%d\n", x);
13    return 0;
14 }
```

Listing 20: Ejemplo de scopes en C

En este caso la salida del programa será 30. Y esto se debe a que la variable que se le pasa al `printf` es una variable local al *scope* de la función `main`.

El scope global no tiene ninguna sintaxis particular, simplemente se declara una variable fuera de cualquier función y en el mismo archivo que estamos trabajando¹. El resto de los scopes se declaran dentro de llaves {}, ya sea scope de funciones o scope de bloques.

Si el scope tiene una sola línea de código, se puede omitir las llaves, excepto en el caso de que se necesite declarar una función. Ejemplos:

```

1 #include <stdio.h>
2
3 int x = 10; // scope global
4
5 int main() {
6    printf("%d\n", x); // imprime el valor de x en el scope global
7
8    int x = 30; // scope local a la funcion main
9    printf("%d\n", x); // imprime el valor de x en el scope local de main
10
11   for(int i = 0; i < 5; i++) // scope de bloque
12      printf("iteracion %d\n", i); // scope de una linea
13
14   int j = 0;
15   while(j < 5) { // scope de bloque
16      int cuadrado = j * j; // scope local a la iteracion del while
17      printf("cuadrado de %d es %d\n", j, cuadrado);
18      j++;
19   }
20
21   return 0;
22 }
```

Listing 21: Ejemplo de scopes en C

En versiones antiguas de C (C89), las variables había que declararlas al principio del bloque o de las funciones. Sin embargo eso ya no es necesario en las versiones modernas de C ni en e C++, de hecho mi recomendación para esta materia es que siempre que sea posible se declare la variable lo más cerca posible de su uso, para evitar confusiones y mejorar la legibilidad del código. Por ejemplo, en un `for` es común declarar la variable de iteración dentro del mismo `for`.

¹En C además existe `extern` que permite declarar variables globales en otros archivos, pero no es necesario para esta materia.

1.7 Errores

Hay algo que no se explica en los libros de algoritmos porque excede el ámbito de esos libros, pero los errores de un lenguaje de programación son un tema fundamental a la hora de programar. Porque no sólo es importante saber escribir código correcto, sino también es importante saber **entender** los errores del compilador para poder luego **corregirlos**. Es por eso que independientemente del lenguaje de programación que vayan a utilizar en un futuro profesional, deben conocer los errores y los *warnings* que el compilador (o el intérprete) les entregue para poder saber qué es lo que está mal.

1.7.1 Errores de compilación

Por ejemplo, supongamos el siguiente código:

```

1 #include <stdio.h>
2
3 int main() {
4     int n = 10;
5     int* ptr = (int*)malloc(n * sizeof(int)); // Error: falta el include <
6     // stdlib.h>
7
8     for(int i = 0; i < n; i++) {
9         printf("Iteracion %d\n", i);
10    }

```

Listing 22: Código con error de compilación

Al compilar este código, el compilador nos dará un error similar al siguiente:

```

ejemplo.c:5:20: error: call to undeclared library function 'malloc'
                  with type 'void *(unsigned long)'; ISO C99 and later do not s
upport implicit function declarations [-Wimplicit-function-declaration]
    int* ptr = (int*)malloc(n * sizeof(int)); // Error: falta el include <stdlib.h>
                                         ^
ejemplo.c:5:20: note: include the header <stdlib.h> or explicitly provide a declaration

```

En este caso, el error nos indica que la función `malloc` no está declarada, lo que significa que no hemos incluido la biblioteca `stdlib.h` que contiene la declaración de la función `malloc`. Para corregir el error, simplemente debemos incluir la biblioteca al principio del código:

```

1 #include <stdio.h>
2 #include <stdlib.h> // Agregamos la biblioteca stdlib.h
3
4 int main() {
5     int n = 10;
6     int* ptr = (int*)malloc(n * sizeof(int)); // Ahora no hay error
7
8     for(int i = 0; i < n; i++) {
9         printf("Iteracion %d\n", i);
10    }
11
12    free(ptr); // No olvides liberar la memoria asignada
13 }
```

Listing 23: Código corregido

A esto se le llama **error de compilación**. Son los errores más fáciles de detectar porque el compilador nos informa de ellos antes de ejecutar el programa, sólo tenemos que ser cuidadosos sobre el mensaje de error que nos entrega el compilador.

1.7.2 Errores de ejecución

Los errores de ejecución son aquellos que ocurren mientras el programa está ejecutando, con lo cual son errores mucho más difíciles de detectar, porque pueden ocurrir en cualquier momento y no siempre son evidentes. Por ejemplo, si tenemos el siguiente código:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int m = 10;
6     int n = 10000;
7     int* ptr = (int*)malloc(n * m * sizeof(int));
8
9     for(int i = 0; i < n*n; i++) { // deberia ser n * m
10        ptr[i] = i;
11    }
12
13    free(ptr);
14 }
```

Listing 24: Código con error de ejecución

Al ejecutar este código, el programa puede fallar con un error de segmentación (segmentation fault) porque estamos intentando acceder a una posición de memoria que no hemos asignado. El error de segmentación ocurre cuando el programa intenta acceder a una dirección de memoria que no le corresponde y el sistema operativo lo detecta, terminando el programa abruptamente.

```
$ ./ejemplo
[1] 29897 segmentation fault ./ejemplo
```

1.8 C++

Hasta acá lo que vimos fueron los conceptos básicos de C y algunas cosas que son paralelas en C++. Si bien C++ se lo asocia con la programación orientada a objetos (OOP), en esta materia no vamos a ver nada de OOP, pero sí vamos a utilizar los contenedores de C++ que son parte de la biblioteca estándar de C++ (STL) y que nos permiten manejar estructuras de datos como vectores, listas, colas, pilas, etc. de una forma mucho más sencilla que en C, ya que se encargan automáticamente de gestionar la memoria y el tamaño de las estructuras de datos, lo que nos permite centrarnos en la lógica de nuestro programa sin tener que preocuparnos por el manejo de memoria como vimos anteriormente.

1.8.1 Contenedores de la biblioteca estándar de C++

Los contenedores de la biblioteca estándar de C++ (STL) son una colección de clases y funciones que proporcionan estructuras de datos que manejan la memoria por nosotros y tienen una interfaz fácil de usar. Algunos de los contenedores más comunes son:

- `std::vector`: Un arreglo dinámico que puede cambiar de tamaño automáticamente.
- `std::set`: Un conjunto que almacena elementos únicos en orden.
- `std::unordered_set`: Un conjunto que almacena elementos únicos sin orden.
- `std::map`: Un mapa que almacena pares clave-valor en orden.
- `std::unordered_map`: Un mapa que almacena pares clave-valor sin orden.
- `std::pair`: Una clase que almacena un par de valores, que pueden ser de diferentes tipos.

Cuando uno utiliza estos contenedores, no es necesario preocuparse por el manejo de memoria, ya que se encargan automáticamente de asignar y liberar memoria según sea necesario. Además, estos contenedores proporcionan una gran cantidad de funciones para manipular los datos almacenados, como insertar, eliminar, encontrar el primer elemento, el último elemento.

La sintaxis para utilizar estos contenedores en la documentación es la siguiente:

- `std::vector<T>`: Donde T es el tipo de datos que se va a almacenar en el vector.
- `std::set<T>`: Donde T es el tipo de datos que se va a almacenar en el set.
- `std::unordered_set<T>`: Donde T es el tipo de datos que se va a almacenar en el `unordered_set`.
- `std::map<K, V>`: Donde K es el tipo de datos que se va a utilizar como clave y V es el tipo de datos que se va a utilizar como valor en el mapa.
- `std::unordered_map<K, V>`: Donde K es el tipo de datos que se va a utilizar como clave y V es el tipo de datos que se va a utilizar como valor en el `unordered_map`.
- `std::pair<T1, T2>`: Donde T1 es el tipo de datos del primer elemento del par y T2 es el tipo de datos del segundo elemento del par.

La sintaxis de los contenedores con <T> o <K, V> se llama sintaxis de plantillas (templates) y es una característica de C++ que permite crear clases y funciones genéricas que pueden trabajar con cualquier tipo de datos. Esto hace que los contenedores de la biblioteca estándar de C++ sean muy flexibles y reutilizables, ya que se pueden utilizar con cualquier tipo de datos sin tener que escribir código específico para cada tipo de datos.

1.8.2 Iteradores

Intrínsecamente relacionados con los contenedores de la biblioteca estándar de C++ (STL) están los iteradores, que son objetos que permiten recorrer los elementos de un contenedor de manera secuencial. Los iteradores proporcionan una interfaz uniforme para acceder a los elementos de diferentes tipos de contenedores, lo que facilita la manipulación de datos sin tener que preocuparse por la estructura interna del contenedor. Por ejemplo, si queremos recorrer un vector de enteros utilizando un iterador, podríamos hacerlo de la siguiente manera:

```

1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main() {
7     vector<int> numeros = {1, 2, 3, 4, 5}; // Crea un vector de enteros
8
9     // Crea un iterador para el vector
10    vector<int>::iterator it;
11
12    // Recorre el vector utilizando el iterador
13    for (it = numeros.begin(); it != numeros.end(); ++it) {
14        cout << *it << " "; // Imprime el valor al que apunta el iterador
15    }
16    cout << endl;
17
18    return 0;
19 }
```

Listing 25: Ejemplo de iteradores en C++

Expliquemos un poco lo que está pasando acá. Primero, creamos un vector de enteros llamado `numeros` e inicializamos con algunos valores.

Vemos que `numeros.begin()` y `numeros.end()` son funciones que devuelven iteradores que apuntan al primer elemento del vector y al elemento siguiente al último elemento del vector. Con lo cual `numeros.begin()` es un iterador que apunta al primer elemento del vector (y de cualquier otro contenedor), mientras que `numeros.end()` es un iterador que apunta a una posición de memoria que está justo después del último elemento del vector, lo que significa que no es un iterador válido para acceder a los elementos del vector, pero es útil para determinar cuándo hemos llegado al final del vector durante la iteración. A esto en matemáticas se lo conoce como que un conjunto está cerrado por abajo y abierto por arriba, porque `numeros.begin()` pertenece al conjunto pero `numeros.end()` no pertenece al conjunto.

Lo interesante de este tipo de iteradores es que se pueden utilizar con cualquier tipo de contenedor de la biblioteca estándar de C++, por ejemplo imaginemos que hacemos lo mismo pero con un `std::set`:

```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main() {
```

```

7  set<int> numeros; // Crea un set de enteros
8  numeros.insert(5);
9  numeros.insert(3);
10 numeros.insert(1);
11 numeros.insert(4);
12 numeros.insert(2);

13
14 // Crea un iterador para el set
15 set<int>::iterator it;

16
17 // Recorre el set utilizando el iterador
18 for (it = numeros.begin(); it != numeros.end(); ++it) {
19     cout << *it << " "; // Imprime el valor al que apunta el iterador
20 }
21 cout << endl;

22
23 return 0;
24 }
```

Listing 26: Ejemplo de iteradores con un set en C++

De todas formas C++ también permite el uso de `auto` para declarar iteradores de forma automática, lo que hace que el código sea más limpio y fácil de leer, por ejemplo:

```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main() {
7     set<int> numeros = {5, 3, 1, 4, 2}; // Crea un set de enteros
8
9     // Recorre el set utilizando un iterador declarado con auto
10    for (auto e : numeros) {
11        cout << e << " "; // Imprime el valor del elemento
12    }
13    cout << endl;
14    return 0;
15 }
```

Listing 27: Ejemplo de iteradores con auto en C++

Pero para esto hay que utilizar las extesiones de C++11, que es una versión de C++ que introdujo muchas mejoras y nuevas características al lenguaje, incluyendo el uso de `auto` para la declaración automática de variables, lo que hace que el código sea más limpio y fácil de leer, ya que no es necesario especificar el tipo de la variable cuando se declara, sino que el compilador lo infiere automáticamente a partir del valor que se le asigna. Otro cambio es que se puede inicializar los `std::set` utilizando una lista `{5, 3, 1, 4, 2}` en lugar de tener que insertar cada elemento por separado utilizando la función `insert`.

Para poder compilar hay que explicitar el uso de C++11 utilizando la opción `-std=c++11` al compilar, por ejemplo:

```
g++ -std=c++11 ejemplo.cpp -o ejemplo
```

1.8.3 Vector

El vector es un contenedor de la biblioteca estándar de C++ que representa un arreglo dinámico, lo que significa que puede cambiar de tamaño automáticamente a medida que se agregan o eliminan elementos. El vector es uno de los contenedores más utilizados en C++ debido a su flexibilidad y eficiencia, ya que proporciona acceso aleatorio a los elementos y tiene un rendimiento similar al de un arreglo tradicional. El vector se declara utilizando la clase `std::vector` y se puede utilizar de la siguiente manera:

```

1 #include <iostream>
2 #include <vector> // Incluye la biblioteca para manejo de vectores
3
4 using namespace std;
5
6 int main() {
7     vector<int> numeros; // Crea un vector de enteros
8
9     // Agrega elementos al vector
10    numeros.push_back(1);
11    numeros.push_back(2);
12    numeros.push_back(3);
13
14    // Imprime los elementos del vector
15    cout << "Elementos del vector: ";
16    for (int i = 0; i < numeros.size(); i++) {
17        cout << numeros[i] << " ";
18    }
19    cout << endl;
20
21    return 0;
22}

```

Listing 28: Ejemplo de vector en C++

La función `push_back` se utiliza para agregar elementos al final del vector, y la función `size` se utiliza para obtener el número de elementos en el vector. Además, el vector proporciona acceso aleatorio a los elementos, lo que significa que se puede acceder a cualquier elemento del vector utilizando su índice, como se muestra en el ejemplo anterior con `numeros[i]` como los *arrays* tradicionales de C.

La complejidad de las operaciones de inserción y eliminación en un vector es amortizada O(1) cuando se agrega o elimina un elemento al final del vector, pero puede ser O(n) en el peor caso cuando se agrega o elimina un elemento en una posición intermedia del vector, ya que es necesario mover los elementos para mantener la contigüidad de la memoria. Sin embargo, el acceso a los elementos del vector es O(1) debido a que el vector almacena los elementos de manera contigua en memoria, lo que permite acceder a cualquier elemento utilizando su índice de manera eficiente.

1.8.4 Set

El set es un contenedor de la biblioteca estándar de C++ que representa un conjunto de elementos únicos, lo que significa que no puede contener elementos duplicados. El set se declara utilizando la clase `std::set` y se puede utilizar de la siguiente manera:

```

1 #include <iostream>
2 #include <set> // Incluye la biblioteca para manejo de sets
3
4 using namespace std;
5
6 int main() {
7     set<int> numeros; // Crea un set de enteros
8
9     // Agrega elementos al set
10    numeros.insert(5);
11    numeros.insert(3);
12    numeros.insert(1);
13    numeros.insert(4);
14    numeros.insert(2);
15
16    // Imprime los elementos del set
17    cout << "Elementos del set: ";
18    for (auto e : numeros) {
19        cout << e << " ";
20    }
21    cout << endl;
22
23    return 0;
24}

```

Listing 29: Ejemplo de set en C++

La complejidad de las operaciones de inserción, eliminación y búsqueda en un set es $O(\log n)$ debido a que el set se implementa utilizando un árbol binario de búsqueda balanceado (generalmente un árbol rojo-negro), lo que permite mantener los elementos ordenados y realizar las operaciones de manera eficiente. Sin embargo, el acceso a los elementos del set no es $O(1)$ como en un vector, ya que el set no almacena los elementos de manera contigua en memoria, sino que los organiza en una estructura de árbol, lo que hace que el acceso a los elementos sea más lento en comparación con un vector.

1.8.5 Unordered Set

La diferencia entre `set` y `unordered_set` es la implementación interna. Mientras que `set` está implementado utilizando un árbol balanceado, el `unordered_set` está implementado con un hash, con lo cual la complejidad en tiempo de acceso cambia, ya que se puede acceder, insertar y eliminar elementos en $O(1)$, pero los elementos no mantienen un orden.

1.8.6 Map

El contenedor `map` es una estructura de datos que almacena valores a partir de una clave (piénsenlo como una base de datos clave-valor). El `map` se declara utilizando la clase `std::map` y se puede utilizar de la siguiente manera:

```

1 #include <iostream>
2 #include <map> // Incluye la biblioteca para manejo de mapas
3
4 using namespace std;
5
6 int main() {
7     map<string, int> edades; // Crea un mapa de string a int
8
9     // Agrega elementos al mapa
10    edades["Juan"] = 30;
11    edades["Maria"] = 25;
12    edades["Pedro"] = 35;
13
14    // Imprime los elementos del mapa
15    cout << "Edades:" << endl;
16    for (const auto& par : edades) {
17        cout << par.first << ":" << par.second << endl; // Imprime la clave y
18        el valor
19    }
20
21    return 0;
}

```

Listing 30: Ejemplo de map en C++

Aquí introducimos el concepto de `const reference` con `const auto& par`, que es una forma de declarar una variable que es una referencia constante a un elemento del mapa. Esto significa que `par` es una referencia a un elemento del mapa, lo que evita la copia de los elementos del mapa durante la iteración (como sucedía en los argumentos de las funciones). La diferencia en este caso es que es una referencia **constante**, lo que significa que no se puede modificar el valor de `par` dentro del bucle, lo que garantiza que no se alteren los datos del mapa durante la iteración.

¿Cómo es que `par` tiene dos campos, `first` y `second`? Porque el tipo de dato de los elementos del mapa es un `pair` que veremos a continuación. Este contenedor tiene dos valores solamente y los métodos para poder acceder a esos valores son `first` y `second`.

1.8.7 Unordered Map

El contenedor `unordered_map` es similar al `map`, pero no mantiene un orden específico de los elementos debido a que está implementado utilizando un hash. Esto significa que las operaciones de inserción, eliminación y búsqueda en un `unordered_map` tienen una complejidad de $O(1)$ en promedio, lo que lo hace más eficiente que un `map` en situaciones donde el orden de los elementos no es importante, esto es similar a lo que ocurría con el `set`.

1.8.8 Pair

El contenedor `pair` es una clase que almacena un par de valores, que pueden ser de diferentes tipos. El `pair` se declara de la siguiente manera:

```

1 #include <iostream>
2 #include <utility>
3
4 using namespace std;
5
6 int main() {
7     pair<string, int> persona; // Crea un par de string e int
8
9     // Asigna valores al par
10    persona.first = "Juan"; // Asigna el primer valor del par
11    persona.second = 30; // Asigna el segundo valor del par
12
13    // Imprime los valores del par
14    cout << "Nombre: " << persona.first << ", Edad: " << persona.second <<
15        endl;
16
17    return 0;
}

```

Listing 31: Ejemplo de pair en C++

Al principio puede parecer un contenedor un poco extraño, pero es súper útil para almacenar pares de valores relacionados, por ejemplo "latitud,longitud".

1.8.9 Tuple

En C++ también existe el contenedor `tuple`, que es similar al `pair` pero puede almacenar un número arbitrario de valores de diferentes tipos. El `tuple` se declara utilizando la clase `std::tuple` y se puede utilizar de la siguiente manera:

```

1 #include <iostream>
2 #include <tuple> // Incluye la biblioteca para manejo de tuplas
3
4 using namespace std;
5
6 int main() {
7     tuple<string, int, double> persona; // Crea una tupla de string, int y
8     // double
9
10    // Asigna valores a la tupla
11    get<0>(persona) = "Juan"; // Asigna el primer valor de la tupla
12    get<1>(persona) = 30; // Asigna el segundo valor de la tupla
13    get<2>(persona) = 1.75; // Asigna el tercer valor de la tupla
14
15    // Imprime los valores de la tupla
16    cout << "Nombre: " << get<0>(persona) << ", Edad: " << get<1>(persona) <<
17    // ", Altura: " << get<2>(persona) << endl;
18
19    return 0;
20 }
```

Listing 32: Ejemplo de tuple en C++

Seguramente no lleguemos a hacer ningún ejercicio que requiera el uso de tuplas, pero es importante conocer su existencia para poder utilizarlas en el futuro.

1.8.10 Composición de Contenedores

Quizás se pregunten **¿Entonces puedo utilizar un vector de sets?**, la respuesta es sí, ya que los contenedores de la biblioteca estándar de C++ se consideran tipos de datos, con lo cual cuando nosotros escribimos `vector<T>` esa T puede ser cualquier tipo de dato, incluyendo otros contenedores de la biblioteca estándar de C++. Esto permite crear estructuras de datos más complejas y flexibles al combinar diferentes contenedores entre sí. Por ejemplo, podríamos declarar un vector de sets de la siguiente manera:

```

1 #include <iostream>
2 #include <vector>
3 #include <set>
4
5 using namespace std;
6
7 int main() {
8     vector<set<int>> vector_de_sets; // Declara un vector de sets de enteros
9
10    set<int> set1 = {1, 2, 3}; // Crea un set de enteros
11    set<int> set2 = {4, 5, 6}; // Crea otro set de enteros
12
13    vector_de_sets.push_back(set1); // Agrega el primer set al vector
14    vector_de_sets.push_back(set2); // Agrega el segundo set al vector
15
16    // Imprime los elementos del vector de sets
17    for (const auto& s : vector_de_sets) {
18        for (const auto& e : s) {
19            cout << e << " "; // Imprime cada elemento del set
20        }
21        cout << endl; // Salto de línea después de cada set
22    }
23
24    return 0;
25 }
```

Listing 33: Ejemplo de un vector de sets en C++

Con lo cual se pueden hacer otras composiciones, por ejemplo una matriz de enteros componiendo un vector de vectores de enteros:

```

1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main() {
7     vector<vector<int>> matriz; // Declara un vector de vectores de enteros
8
9     // Crea una matriz de 3x3
10    for (int i = 0; i < 3; i++) {
11        vector<int> fila; // Crea un vector para la fila
12        for (int j = 0; j < 3; j++) {
13            fila.push_back(i * 3 + j + 1); // Agrega elementos a la fila
14        }
15        matriz.push_back(fila); // Agrega la fila a la matriz
}
```

```
16 }
17
18 // Imprime los elementos de la matriz
19 for (const auto& fila : matriz) {
20     for (const auto& elemento : fila) {
21         cout << elemento << " "; // Imprime cada elemento de la fila
22     }
23     cout << endl; // Salto de línea después de cada fila
24 }
25
26 return 0;
27 }
```

Listing 34: Ejemplo de una matriz de enteros en C++