Apuntes de C++

migueluisV

Realizadas: Febrero 2022

Índice

List of Tables

1 Conceptos básicos

C++ es un famoso lenguaje de programación multi-plataforma para crear aplicaciones tales como sistemas operativos, navegadores, videojuegos, aplicaciones de ciencia o arte, etcétera. Algunas de las nociones más básicas de este lenguaje son:

- Para terminar una instrucción en C++, se utiliza el **;**.
- **main()** es la función principal de C++, es la primer función que se ejecuta al abrir la aplicación o programa.

```
int main()
{
    // Código.
    return 0;
}
```

- **cout** es el comando de despliegue de información de C++. Para decirle a cout que despliegue algo, se usan los símbolos << y lo que se quiera desplegar, en el caso de querer mostrar alguna variable numérica, no es necesario convertirla a texto. Pueden ponerse varios <<uno tras otro para concatenar cadenas de texto.

```
cout << "MENSAJE";
cout << variable;
cout << "mensaje" << "MENSAJE" << "Mensaje";
```

- **cin** es el comando de entrada de datos de C++, se utiliza igual que *cout*, es decir, utilizando los dos >> seguido de a qué variable le asignaremos un valor. Al igual que *cout*, podemos realizar varias asignaciones de valores en la misma línea *cin*.

```
cin >> variable;
cin >> a >> b;
```

- Para dar un salto de línea en C++, se usa el comando **endl** o el comando de escape (escape character) **\n**.

Nota: para asignar o desplegar un valor, previamente debe existir la variable o que tenga un valor inicializado.

```
int main()
{
    cout << "Hola mundo"; // Esto imprime el mítico mensaje.
    cout << "Hola mundo\nTe saludo"; // Esto imprime dos mensajes separados por un salto de
        linea hecho con \n.
    cout << "Hola mundo" << endl << "Te saludo"; // Esto imprime dos mensajes separados por
        un salto de linea hecho con endl.
}
```

2 Cabeceras y librerías

C++ puede utilizar varias cabeceras para volver más rico el ambiente de código, el que viene por defecto suele ser <**iostream**>, para agregar más cabeceras al programa se utiliza la palabra reservada **#include**.

```
#include <iostream>;
```

Los **namespaces** son librerías que dan un plus para incluir más objetos y darle más alcance a los identificadores dentro del código, el namespace que siempre aparece o que es usualmente utilizado es **std**, se usa las palabras reservadas **using namespace**.

```
using namespace std;
```

Nota: estas cabeceras y librerías se escriben antes de *main()*.

3 Comentarios

Los comentarios en los lenguajes de programación son utilizados para explicar lo que el programa y código hacen, el compilador simplemente ignora el inicio de un comentario hasta que se hace un salto de línea. Al igual que en C#, C++ realiza comentarios para una sola línea con las \\, si quieres encerrar un bloque de varias líneas, al comienzo de este bloque, escribes: *, y al final: *\, esto encierra varias líneas de código y las convierte en comentarios.

```
int main()
{
    cout << "Hola mundo"; // Esto imprime el mítico mensaje.
    cout << "Hola mundo";
    /*cout << "Hola mundo";
    cout << "Hola mundo";
    cout << "Hola mundo";
    cout << "Hola mundo";*/ // Todo esto es ignorado.
}
```

4 Variables

Una variable es un espacio reservado de memoria en la cual uno almacena información, C++ requiere que le indiques el tipo de dato que almacenará la variable, al mismo tiempo que le des un nombre identificatorio significativo, ya que en este lenguaje hay variables, funciones, arreglos, módulos y clases que también deben tener un nombre distinguible, una vez que lo declares, no tienes que hacerlo cada que quieras usar la variable. En esa misma línea donde estás creando una variable, puedes darle un valor (inicializarla) o dejarla vacía (sin inicializar), por último, podemos crear e inicializar varias variables del mismo tipo en la misma línea.

[tipo de dato] [nombre identificatorio];
[tipo de dato] [nombre identificatorio] = [valor de inicio];
[tipo de dato] [nombre identificatorio], [nombre identificatorio], ..., [nombre identificatorio];

Las variables pueden usarse para operaciones de texto (string, char) o para operaciones aritméticas (int, float)

```
int main()
{
    int a;
    int b = 42;

    int sum = a + b;
    cout << sum
}
```

La palabra reservada **auto** permite crear variables y que automáticamente se declare el tipo de dato que le corresponde a dicha variable, con esto, dejamos que C++ le asigne el tipo de dato a nuestras variables dependiendo únicamente del valor que se le dé por primera vez (inicializado), si declaramos una variable como auto y le damos un valor después de esta línea de declaración, ocurrirá un error de sintaxis.

```
auto a = 5; // Variable del tipo entero, está bien.
auto b = 5.1; // Variable del tipo float, está bien.
auto a = "hola"; // Variable del tipo string, está bien.
auto a; // Variable del tipo auto.
a = 5; // Esto no está bien.
```

4.1 Reglas para nombrar variables

Para administrar mejor el nombre de las variables se pueden seguir los siguientes consejos:

- Ponerle una letra mayúscula o un guión bajo (_).
- Ponerle un nombre significativo que se pueda recordar fácilmente.
- No se permiten caracteres especiales o espacios en blanco.
- No se permiten palabras reservados como nombres de variables.
- C++ distingue entre mayúsculas y minúsculas, para tener cuidado nombrando así las variables.

5 Aritmética básica

Con las variables matemáticas se pueden realizar diversas operaciones aritméticas:

Table 1: Operadores aceptados en C++

Operador	Símbolo	Ejemplo
Adición	+	x + y
Sustracción	-	x - y
Multiplicación	*	x * y
División	/	x / y
Modulo	%	x % y

Nota: dividir entre 0 causa un error en el programa.

Como bien se sabe, la operación módulo (%) es usada para conocer el residuo de una división (el módulo de 25/5 es 0 porque esa es una división exacta, el módulo de 50/26 es 24 porque 26 cabe solo una vez y restan 24 unidades).

```
int x = 50, y = 26, res;
res = x % y;
cout << res; // Imprime 24.
```

La prioridad de operaciones u operadores también está presente en la aritmética de C++:

Table 2: Prioridad de operaciones

Nivel de prioridad	Operaciones	Dirección de ejecución
1	(), [], {}, ., ->	Izquierda a derecha
2	*, /, %	Izquierda a derecha
3	+, -	Izquierda a derecha

6 Asignación e incremento en variables

Ya hemos visto como se le asignan o se inicializa una variable, pero cabe recalcar y aclarar que el símbolo = asigna lo que esté a la derecha de este al lado izquierdo (int a = 5, ahora a vale 5), pero en ocasiones, requeriremos que a una variable se le asigne lo que esta vale actualmente, más otro valor, para ello podríamos hacer lo siguiente:

```
int main()
{
    int x = 20;
    x = x + 12;
    cout << x; // Con esto, primer x vale 20, luego se le asigna lo que vale (20) mas 12,
        dando como resultado al final 32.
    return 0;
}
```

Sin embargo, podemos utilizar los **operadores de asignación**, que son un acceso rápido a la operación x = x + n que utilizamos anteriormente, para utilizarlos usamos el comando += (es igual a x = x + n;) o -= (es igual a x = x - n;), estos mismos accesos directos aplican para los otros operadores de multiplicación, división y módulo.

```
int main()
{
    int x = 5;
    x += 5; // Equivalente a x = x + 5.
    x -= 2; // Equivalente a x = x -2.
    x *= 10; // Equivalente a x = x * 10.
    x /= 1; // Equivalente a x = x / 1.
    x %= 2; // Equivalente a x = x % 2.
    return 0;
}
```

Otro punto a destacar es la existencia de los **operadores de incremento**, usualmente utilizados en ciclos, a diferencia de los operadores de asignación, estos solo pueden incrementar o decrementar una unidad.

Estos operadores de incremento poseen una característica, y es que pueden ser de cierto tipo:

- **Prefijo**: van dos $-$ o $+$ antes del nombre de la variable. El tipo prefijo incrementa el valor de la variable, luego continúa con el código o la expresión donde se esté trabajando.
- **Posfijo**: van dos $-$ o $+$ después del nombre de la variable. El tipo posfijo evalúa todo el código o expresión, luego incrementa la variable.

```
int main()
{
    int x = 5, y;
    y = ++x;
    cout << y; // x vale 5 inicialmente, en su siguiente linea, x incrementa mas uno su
        valor y finalmente se le asigna a y su valor. Resultado: y = 6, x = 6.

    int a = 6, b;
    b = a++;
    cout << b; // a vale 6 inicialmente, en su siguiente linea, b se le es asignado lo que
        vale a, luego esta incrementa mas uno su valor. Resultado: a = 7, b = 6.

    return 0;
}
```

Nota: este tipo de operadores suelen verse en ciclos, para esos casos, el ciclo puede dar una primera vuelta, luego el valor del contador que este posee incrementa, o primero incrementa el valor del contador, luego se da la primer vuelta al ciclo.

7 Condicionales

La declaración **if** sirve para poner condicionales en código, si la condición **true** se cumple, se ejecuta un bloque de código, sino, se ejecuta otro bloque o se ignora, continuando con el código. Se utilizan **operadores relacionales** para algunas sentencias condicionales funcionales:

Table 3: Operadores relacionales

Operador	Descripción
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
==	Igual a
!=	Distinto que

```
if (condición){
    // Código.
}
else{
    // Código.
}
```

C++ da la posibilidad de poder **anidar sentencias** if, esto quiere decir que podemos insertar un if dentro de otro, en su estructura completa, puede ser un número ilimitado de condicionales. Por otro lado, si solo se le da una instrucción al if o al else, pueden omitirse las llaves, para hacer menos volumen de código.

```
int x = 15;
if (x >= 18){
    if (x >= 21)
        cout << "Legal en el mundo";
    else
        cout << "Legal el México";
}
else{
    if (x == 15){
        cout << "Puede tener quinceañera";
    }
    else{
        cout << "Es menor de edad";
    }
}
```

Nota: si no se ponen las llaves y se escribe más de una instrucción, C++ lanzará un error.

8 Ciclos

Un ciclo es un bloque de código que se ejecuta hasta que una condición es cumplida. Pueden ser usados para obtener datos de los usuarios repetidas veces, desplegar información de un arreglo o lista, entre otras cosas.

8.1 Ciclo while

Un ciclo **while** es un ciclo tal que se ejecutará siempre y cuando su condición sea verdadera, es decir, un ciclo while con condición 5>4 puede ser infinito solo hasta que 5 no sea mayor a 4. Debe de existir un contador o variable que logre hacer que este ciclo termine, ya que sin este, el ciclo será infinito.

```
while (condición){
    // Código.
    // Contador de salida.
}
```

Un **contador** es una variable que va adquiriendo valor uno más uno, suele ser utilizado en este tipo de ciclos para darles fin o para desplegarlos dentro del mismo ciclo, pero también son usados en los ciclos for, do-while u otras circunstancias. Podemos combinar los contadores con las operadores de incremento.

```
int contador;
contador = contador + 1; // Forma de incrementar el contador.
contador++; // Forma de incrementar el contador con operadores de incremento.
```

Un **acumulador** es una variable que va sumando valores distintos uno tras otro dentro de un ciclo u otra circunstancia, puede que sea un único valor o todos distintos, suele ser inicializada con valor igual a cero. Se pueden usar los operadores de asignación para hacer esta operación un poco más rápida.

```
int acumulador;
acumulador = acumulador + 5; // Forma de acumular.
acumulador = acumulador + 10;
acumulador += 5; // Forma de acumulador con operadores de asignación.
acumulador += 10;
```

```
int main()
{
    int x = 0; //Variables

    while (x<10){ // Condicional, este ciclo imprime el contador que se esta evaluando
        hasta que sea 9, el contador va incrementando uno a uno, pero se puede modificar
        este incremento.
        cout << "El contador vale " << x; // Código.
        x++; // Contador.
    }

    return 0;
}
```

8.2 Ciclo for

El ciclo **for** es aquel que permite controlar mejor la cantidad de ciclos que este dará, siendo más preciso, dentro de su sintaxis, nos encontramos con que primero requiere de una variable con un valor que será evaluada, punto y coma después, la condición con la que se trabajará, punto y coma después, el control de incremento del ciclo; las llaves son obligatorias.

```
for (variable; condición; incremento){
    // Código.
}
```

La lógica de este ciclo es que, se lee la variable y su valor con la que trabajará el ciclo primero, después evalúa la condición para verificar que sea cierta, y ejecuta el bloque de código insertado dentro de las llaves, finalmente incrementa o decrementa la variable (pueden ser usados los operadores de incremento o asignación) y así sucesivamente hasta que la condición sea falsa.

```
for (int x = 0; x < 10; x++){
    cout << x; // Imprime de 0 a 9 los valores de x, luego se sale del ciclo.
}
```

8.3 Ciclo do-while

A diferencia del ciclo *while*, el ciclo **do-while** evalúa la condición que se le da al final del bloque de código con sus instrucciones, dando como resultado un ciclo en donde al menos una vez, se ejecuta su bloque de código, a diferencia de los otros dos, donde primero se evalúa y luego se ejecuta.

```
do {
    // Código.
} while (condición);
```

Aquí pasa como con el ciclo while, debe tener un contador que haga que el ciclo termine, si esta no existe, el ciclo será infinito.

```
int contador;

do {
    cout << "Esto es un ciclo do-while.";
    contador++
} while (contador <= 15);
```

9 Condicional switch

Como vimos previamente, podemos tener tantas condicionales y condicionales anidadas como queramos, pero esto puede causar gran volumen de código, para ello, existe una alternativa llamada la condicional **switch**, la cual evalúa un valor, condición o expresión y, dentro de sus llaves, propone varios resultados para varias evaluaciones de esta variable, si ninguno de los resultados coincide con la evaluación dada, puede caer en un resultado por defecto. Sintaxis:

```
switch (condición){
    case valor1:
        // Código.
        break;
    case valor2:
        // Código.
        break;
    ...
    case valorN:
        // Código.
        break;
    default:
        // Código.
}
```

Todo **case** requiere que, después de todas las instrucciones que se le den, se ponga un **break;**, ya que la condicional *switch* seguirá ejecutando los otros *case* hasta que encuentre un *break*. Esto no aplica para **default**, ya que este, por defecto, es el último case dentro de switch.

```
int x = 0;
switch (x) {
    case 1:
        cout << "Es uno";
        break;
    case 10:
        cout << "Es diez";
        break;
    case 100:
        cout << "Es cien";
        break;
    default:
        cout "Es cero";
}
```

10 Operadores lógicos

Los operadores lógicos usados para condicionales y ciclos son los siguientes:

Table 4: Operadores lógicos en C++

Operador	Nombre	Ejemplo
&&	AND (y que)	x && y
||	OR (o que)	x || y
!	NOT (no es igual)	! x

La forma en la que funciona la evaluación del operador lógico **AND** es similar a como interactúan los símbolos matemáticos de + y -:

- falso y falso = **falso**
- falso y verdadero = **falso**
- verdadero y falso = **falso**
- verdadero y verdadero = **verdadero**

Este operador tiene este comportamiento especial, el operador **OR** debe tener solamente un valor **verdadero** para que toda la evaluación resulte en verdadera.

10.1 Operador ternario

El **operador ternario** es un tipo de operador especial que evalúa una comparación y, si la expresión o variable del lado izquierdo del operador es verdadera, regresa dicha expresión, sino, regresa la expresión o variable de lado derecho. Es un operador sencillo que reduce a una línea de código una condicional.

```
#include <iostream>
using namespace std;

int main(){
    int a, b;
    string m; // Declaración de variables.
    cin >> a >> b; // Asignación de valores a las variables.
    m = a > b ? " a es mayor" : "a es menor"; // Evalúa a > b; si a es mayor, a m se le
        asigna una cadena, sino, se le asigna otra.
    cout << m << "\n";
}
```

11 Tipos de datos

Los tipos de datos son un identificador de las variables que se crean en C++, con estas, se puede dar un uso apropiado a cada variable, qué puede ser almacenado en estas y qué operaciones se pueden realizar. Existe una forma legal e ilegal de utilizar los tipos de datos, y es simplemente que no pueden realizar operaciones entre tipos de datos distintos, no se puede sumar la palabra texto al número 5. El tamaño de cada tipo de dato depende de la arquitectura del sistema en la cual el programa se ejecuta, por lo general, 4 bytes es el tamaño mínimo en las arquitecturas modernas.

11.1 Tipo numérico

Este tipo de dato incluye:

- **Enteros (integer)**: son todos aquellos números no fraccionarios, los cuales pueden ser positivos o negativos, para utilizar un valor entero se usa la palabra abreviada **int**, el tamaño utilizado para *int* por defecto es de 4 a 8 bytes. También podemos modificar el tipo int, haciendo que reciba solamente valores positivos, ambos, que pueda contener valores más pequeños o más grandes, como lo muestran los siguientes puntos:
  - **signed**: entero que acepta positivos y negativos $(-\infty, 0, \infty)$.
  - **unsigned**: entero que solo acepta positivos (0, ..., ∞).
  - **short**: la mitad del tamaño por defecto (2 bytes).
  - **long**: el doble del tamaño por defecto (8 bytes).

```
int a = 5;
unsigned short int b = -1;
long int c = 555666777;
```

- **Punto flotante (float)**: son todos aquellos números que poseen un punto decimal, suelen ser del tipo *signed*, lo cual significa que automáticamente acepta valores positivos y negativos, su tamaño por defecto es de 4 bytes, pero también existen otros dos tipo de punto flotante:
  - **double**: una versión más grande de *float*, de 8 bytes de tamaño
  - **long**: una versión más grande de *double*, para ser exactos, *long* equivale a dos doubles, por lo cual, long tiene un tamaño de 16 bytes.

```
float var1 = 420.0;
double var2 = -3.33;
long var3 = 0.035546;
```

11.2 Tipo cadena y caracteres

Este tipo está conformado por números, caracteres o símbolos:

- **Cadena (string)**: consiste en una una serie de símbolos, letras, caracteres o números uno tras otro, formando frases o palabras. Estas cadenas se ponen entre dos comillas (" "). Este tipo de dato se encuentra dentro del cabecera <**string**>, que a su vez esta se encuentra dentro de *iostream*, por lo que si ya se trabaja con iostream, no es necesario volver a escribir la cabecera string.

```
    string nombre = "Luis";
```

- **Un carácter (character)**: consiste en un solo símbolo o letra de 1 byte de tamaño, representando como un carácter ASCII, encerrado entre una sola comilla (' ').

```
    char letra = 'a';
```

11.3 Tipo booleano

Este tipo simplemente puede tener dos valores: **true** (verdadero, 1) o **false** (falso, 0).

```
bool online = true;
bool loggeado = 0 // Adquiere valor de false.
```

12 Arreglos

Un **arreglo** (también llamado **vector**) es una colección de datos del mismo tipo bajo el mismo nombre, podemos pensar a los arreglos como un grupo de variables con el mismo nombre, pero distinto valor, en vez de crear 10 variables con el mismo nombre, pero diferenciándolos con un contador, se puede crear un arreglo de 10 espacios para almacenar 10 valores. Cuando se crea un arreglo, debes darle un tipo de dato, un nombre significativo y un número entero de valores que puede recibir encerrado entre corchetes ([]).

```
int a[5]; // Arreglo de tipo entero llamado a con 5 espacios para recibir valores.
```

Así como una variable puede ser inicializada, un arreglo también, después de crearlo, podemos darle valores (dependiendo del tipo de dato), separados por comas y encerrados entre llaves. Sin embargo, también podemos inicializar un arreglo como se acaba de describir, pero sin indicar el número de espacios que van a recibir valores dentro de los corchetes, a continuación se muestran ejemplos de como se inicializan los arreglos, ambos son el mismo pero inicializados de distinta manera:

```
int b[3] = {1, 22, 333};
int b[] = {1, 22, 333};
```

Nota: si el arreglo tiene 3 espacios para valores y se desea iniciar dicho arreglo, a la hora de escribir valores entre las llaves, no debe superar el número de espacios que se indicó en la creación del arreglo, sino, causará un error.

```
int c[2] = {32, 101, 4, 5} // Esto está mal.
```

Cuando se crea un arreglo, supongamos, de 10 espacios, cada uno tiene un **índice**, con este índice, podemos utilizar los valores contenidos en el arreglo. Este índice comienza con el valor 0 (un arreglo de 10 valores tiene valores de índice del 0 al 9). Para acceder a los valores contenidos dentro del arreglo, utilizamos su nombre, seguido de corchetes, y dentro de estos, el índice del valor que estamos buscando.

```
int main(){
    int arreglo[3] = {1, 2, 3}; // Índices: {0, 1, 2}.
    cout << arreglo[2] // Despliega: 3.
    return 0;
}
```

12.1 Arreglos en ciclos

Podemos utilizar un ciclo para asignarle valores a cada índice un arreglo. Hay que ser precavido con esto, ya que, si el ciclo se va a repetir 5 veces, y nuestro arreglo tiene 5 espacios, la variable del ciclo debe comenzar en 0 y la condición que evalúa el ciclo debe ser igual a 5 menos 1 (para el operador <=) o 5, pero el operador tiene que ser obligatoriamente >, ya que los índices de un arreglo también comienzan en 0, si la variable del ciclo tiene un valor de 5, esto estaría fuera de los límites del índice del arreglo (5>4), lo cual lanzaría un error.

```
int (){
    int x;
    int arreglo[5];
    for (x = 0; x <= 4; x++){
        arreglo[x] = x;
        cout >> arreglo[x];
    }

    return 0;
}
```

12.2 Arreglos multidimensionales

Los arreglos pueden tener **más de una dimensión, o más de un índice**, a esto se le llama **arreglos multidimensionales**. Los más comunes son los de dos (como una tabla) y tres dimensiones (como un cubo), la sintaxis a continuación:

[tipo de dato][nombre] [n. de espacios][n. de espacios]...[n. de espacios];
string nombres[2][4]; // Ejemplo.

Para visualizar como es un arreglo bidimensional, tenemos la siguiente tabla: está compuesta de dos filas y cuatro columnas.

Table 5: Representación de un arreglo bidimensional

	Columna1	**Columna2**	**Columna3**	**Columna4**
Fila1	Índice[0][0]	Índice[0][1]	Índice[0][2]	Índice[0][3]
Fila2	Índice[1][0]	Índice[1][1]	Índice[2][2]	Índice[3][3]

Para inicializar un arreglo bidimensional (y de aquí podemos tomar como ejemplo otros arreglos multidimensionales), debemos seguir lo siguiente:

```
int inicializacion[3][3] = {
    {1, 2, 3}, // 1ra fila.
    {4, 5, 6}, // 2da fila.
    {7, 8, 9} // 3ra fila.
};
// 0 se puede hacer así también.
int inicializacion2[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

Si queremos acceder a los valores dentro de un arreglo multidimensional, se aplica la misma filosofía de utilizar índices que con un arreglo de una sola dimensión:

```
int x[2][2] = {{1, 2}{3, 4}};
cout << x[0][1];
```

13 Introducción a los Punteros

Un **puntero** es una forma de referirse a la dirección de memoria física definida que tiene una variable, para mostrarla en pantalla, se debe poner el símbolo & seguido del nombre de la variable en cuestión. Un puntero puede ser tratado como una variable con ciertos atributos, los cuales son: **la dirección** y **el contenido** del puntero.

Para contemplar qué es un puntero, podemos pensar que tenemos una caja, esta caja está ubicada en alguna parte de un cuarto, y por supuesto que esta caja tiene algo contenido adentro, puede ser cualquier cosa, un *puntero* puede ser una persona que sabe donde está esa caja y cual es su contenido, es así que, si buscamos saber qué hay dentro de la caja y dónde está ubicada, podríamos ir nosotros directamente, o podríamos preguntarle a esta persona dónde está la caja y qué tiene adentro; este pequeño pensamiento nos sirve para poder diferencia entre variables y punteros.

Un puntero puede crearse como una variable, es decir, dándole un tipo de dato y un nombre, para hacerlo, debemos escribir un **asterisco (*)** después del tipo de dato, entre el tipo de dato y el nombre del puntero o después del nombre de la variable. El tipo de dato de un puntero es un *long* y, si un puntero apunta a una variable, ambos deben de ser del mismo tipo, un puntero *char* no puede apuntar a una variable *string*.

```
int* puntero1;
int *puntero1;
int puntero3*; // Tres formas de declarar un puntero.
```

Para poder asignarle la referencia de una variable a un puntero, se utiliza el operador **&** previo al nombre de una variable en una asignación de puntero, como lo muestra el siguiente ejemplo:

```
int main(){
    int* aPuntero; // Puntero tipo int.
    int a; // Declaración de variable int.

    aPuntero = &a; // Al puntero aPuntero se le asigna la dirección de memoria física de la
        variable a (referencia).

    return 0;
}
```

Además del símbolo & para acceder a la dirección de memoria de una variable (**Dirección-de**), el símbolo * da acceso al valor contenido en la variable puntero (**Contenido-de**). El siguiente código explica un poco estos conceptos:

```
#include <iostream>
using namespace std;

int main(){
    int* puntero1;
    int *puntero1;
    int puntero3*; // Tres formas de declarar un puntero.

    int a; // Declara variable int.

    puntero = &a; // Crea puntero del tipo int que apunta a la variable a.

    cout << "Contenido de a: " << *puntero1 << "\n"; // Despliega el contenido de la
        variable a la que apunta.
    cout << "Dirección de memoria: " << puntero1 << "\n"; // Despliega la dirección de
        memoria física de la variable a la que apunta.

    return 0;
}
```

14 Memoria dinámica

C++ administra su memoria en dos partes: **El stack (pila o memoria estática)** es aquel espacio de memoria de donde las variables comunes (variables dentro de funciones o procedimientos, dentro de main y globales) son guardadas, este espacio de memoria suele ser bastante reducido, por lo que aquí se suelen presentar las excepciones por *desbordamiento de pila*, y **El heap (montón)**, que es una sección de memoria sin utilizar que se puede tomar cuando el programa es ejecutado, así asignando dinámicamente la memoria. El conocer estas dos categorías nos permite pensar en como es que utilizamos la memoria para nuestro programa, y el impacto que esto tiene a la hora de ejecutarlo (tiempo de carga). Por otra parte, la memoria heap (ahora llamada dinámica) nos permite darle un valor dinámico a nuestras variables, en especial a los arreglos, ya que un problema concurrente con estos

últimos es que no podemos cambiar el valor de los espacios que puede utilizar (se establece en la compilación y durante la ejecución no se puede alterar).

La memoria dinámica es trabajada por medio de **punteros**, cuando a una variable se asigna a la memoria dinámica, lo que recibe esta variable es una dirección de memoria física, por lo que debemos tener un puntero a la mano para manipular esta variable. Las palabras reservadas dedicadas a esta memoria son **new** y **delete**. Podemos verlo distinto definiendo los tipos de memorias con otras palabras:

- Memoria stack, pila o estática: es aquella que se establece durante el compilado y no se puede alterar en la ejecución.

```
int vector[50];
```

- Memoria pseudo estática: es aquella que se establece por medio de la entrada de un dato por parte del usuario, se compila y no se puede alterar durante la ejecución.

```
int N;
cout << "N: ";
int vector[N];
```

- Memoria heap o dinámica: es utilizada mediante *punteros*, se establece durante el compilado y puede ser alterado durante la ejecución.

```
int* vector = new int[N];
// Código.
delete[] vector;
vector = null;
```

Una buena practica es crear punteros e inicializarlos a valor *null*, utilizar el la instrucción *delete* después de utilizarlos (con o sin el uso de memoria dinámica) y volver a asignales un valor *null*.

15 El operador sizeof()

Como se dijo previamente, el tamaño de cada tipo de dato varia de la arquitectura de la computadora en la que se está trabajando, pero C++ ofrece un mínimo de tamaño para cada tipo de dato que ofrece, este operador se puede utilizar también para variables de cierto tipo de dato.

```
sizeof(tipo de dato)
```

La siguiente tabla muestra los el mínimo tamaño ofrecido por C++:

Table 6: Tamaños mínimos de los tipos de datos

Categoría	Tipo	Tamaño mínimo
integer	short	2 bytes
	int	2 bytes
	long	4 bytes
	long long	8 bytes
floating point	float	4 bytes
	double	8 bytes
	long double	8 bytes
character	char	1 byte
boolean	bool	1 byte

Este mismo operador es útil para conocer el **tamaño de bytes de un arreglo**, simplemente lo declaramos (no es necesario asignarle valores) del tipo que deseemos, y dentro del operador sizeof, escribimos el nombre del arreglo. También podemos conocer el número de elementos que tiene un arreglo utilizando sizeof del arreglo entre el número de bytes que representa el tipo de dato del arreglo:

```
int a[10]; // Arreglo int, tiene un tamaño de 2 bytes.
double b[10]; // Arreglo double, tiene un tamaño de 8 bytes.

cout << sizeof(a) << endl; // Despliega 20, que es 2 bytes por los 10 espacios del arreglo
    .
cout << sizeof(b) << endl; // Despliega 80, que es 8 bytes por los 10 espacios del arreglo
    .
cout << sizeof(b) / 8 << endl; // Despliega 10, que es el tamaño de los bytes del arreglo
    (80) entre los 10 espacios.
```

16 Funciones

Una **función** es un bloque de código que puede ser llamado dentro de otra sección del código para realizar una tarea en específico. Las ventajas de las funciones es que puede crear las tuyas que hagan una determinada tarea que requieras, además de que estas funciones son reutilizables y puedes probarlas cada una individualmente.

Toda función tiene la siguiente estructura:

```
[Tipo de dato] [Nombre] ([Parámetro1], [Parámetro2], ..., [ParámetroN])
{
    // Código.
    return [Variable o expresión del mismo tipo de dato de la función]
}
```

Los parámetros no son precisamente obligatorios, se ponen para fines de establecer la estructura de una función, lo que si es requerido es que la instrucción **return** regrese un resultado, variable o expresión del mismo tipo que sea la función, sino había un error.

En ocasiones, las funciones no regresarían un valor, simplemente cumplen con una tarea, es decir, regresan un vacío, a este tipo de funciones se le llaman **procedimientos** y su estructura es la siguiente:

```
void [Tipo de dato] [Nombre] ([Parámetro1], [Parámetro2], ..., [ParámetroN])
{
    // Código.
}
```

Para llamar y ejecutar una función o procedimiento se tiene que haber creado previo a su utilización, debemos simplemente escribir el nombre de la función o procedimiento, seguido del inicio y cierre de paréntesis, y punto y coma.

```
void Despliegue()
{
    cout << "Hola mundo";
}

int main()
{
    Despliegue(); // Despliega "Hola mundo".

    return 0;
}
```

Una **función prototipo** consiste en la declaración simple de una función o procedimiento, pero sin darle un bloque de código a ejecutar, es como declarar una variable y no inicializarla, con esto, podemos crear *funciones prototipo*, llamarlas en el *main*, y darles un cuerpo o bloque a dichas funciones por debajo de main o en otra ubicación que deseemos.

```
void Despliegue1(); // Prototipo de procedimientos.
void Despliegue2();

int main()
{
    Despliegue1(); // Despliega "Hola mundo".
```

```
    Despliegue2(); // Despliega "Hola mundo".

    return 0;
}

void Despliegue1()
{
    cout << "Hola mundo1";
}

void Despliegue2()
{
    cout << "Hola mundo2";
}
```

16.1 Con parámetros

Si somos más específicos, una función puede tener **argumentos**, y los valores que le pasamos a dichos argumentos se llaman **parámetros**, entonces, para que una función o procedimiento tenga argumentos, debemos seguir la siguiente estructura:

```
void Despliegue([tipo de dato] [nombre])
{
    // Código.
}
```

El siguiente ejemplo muestra un procedimiento que acepta un argumento y, en su llamada, le pasamos un parámetro:

```
void Despliegue(string mensaje)
{
    cout << mensaje;
}

int main()
{
    string mensaje = "que tal";
    Despliegue(mensaje); // Despliega "que tal".

    return 0;
}
```

Si bien un *procedimiento* no regresa un valor y realiza una tarea sin más, una función si regresa un valor, y este valor puede ser asignado a una variable, como si se tratase de una asignación cualquiera, siempre y cuando sean del mismo tipo de dato. Además, la instrucción *return* puede modificar el argumento que recibe la función para devolverlo alterado:

```
int Multiplicacion(int x)
{
    return x*10;
}

int main()
```

```
{
    int numero;
    numero = Multiplicacion(10); // La función le asigna a la variable el valor 10
        multiplicado por 10.

    return 0;
}
```

16.1.1 Argumentos predeterminados

Una función o procedimiento puede argumentos con **valores predeterminados**, esto quiere decir que, si le hablamos a la función o procedimiento, y a esta no le pasamos ningún parámetro que corresponda con su argumento en la estructura de la función, esta misma asumirá que dicho argumento tiene asignado su valor por defecto, esto aplica para funciones y procedimientos con n argumentos.

```
int Suma(int x, int y, int z = 5) // El valor predeterminado del argumento z es 5.
{
    return x + y + z;
}

int main()
{
    int resultado1 = Suma (1, 2, 3); // Llama a Suma y le pasa tres parámetros.
    int resultado2 = Suma(2, 2); // Llama a Suma y le pasa dos parámetros.

    cout << "Resultado 1: " << resultado1 << endl; // Despliega 6.
    cout << "Resultado 2: " << resultado2 << endl; // Despliega 9.
}
```

16.2 Con múltiples parámetros

Le puedes poner tantos argumentos como uno desee, separados por comas , siempre y cuando cada argumento tenga su tipo de dato y nombre. Para llamar a la función o procedimiento, se sigue la misma lógica que tiene el llamar una función con un solo parámetro, si tiene varios, se separan por una coma.

```
int Multiplicacion(int x, int y, int z)
{
    return x+y+z;
}

int main()
{
    int numero;
    numero = Multiplicacion(10, 20, 30); // La función le asigna a la variable el valor 10
        más 20 más 30.
    cout << numero;

    return 0;
}
```

16.3 Arreglos como parámetros

Una función o procedimiento también **arreglos** como argumento, basta con indicarle al argumento un tipo de dato, un nombre identificativo y los corchetes (sin ningún contenido dentro de ellos).

```
void DespliegueArreglo(int arreglo[], int size) {}
```

Para llamar a la función o procedimiento y pasarle como parámetro un arreglo, debemos escribir (en su correspondiente argumento) el nombre del arreglo sin los corchetes, C++ reconoce que es un arreglo y asume todos sus valores sin poner los corchetes. A continuación se presenta un ejemplo en código:

```
void DesplegarArreglo(int arreglo[], int size)
{
    int i;

    for (i = 0; i < size; i++)
    {
        cout << arreglo[i] << endl;
    }
}

int main()
{
    int tam, i;

    cout << "Tamaño del arreglo: ";
    cin >> tam;

    int arr[tam];

    for (i = 0; i < tam; i++)
    {
        arreglo[i] = i;
    }

    DesplegarArreglo(arr, tam);
}
```

16.4 Punteros como referencias

Por defecto, C++ tiene por defecto el pase de valores por parámetro hacías las funciones y procedimientos, pero existen otra forma de pasarle un parámetro a una función o procedimiento:

- **Por valor**: lo que pasa cuando se le manda valor por parámetro a la función o procedimiento es que, desde donde se llama, se copia el valor del argumento (variable) local y dicha copia se le pasa como parámetro a la función o procedimiento, por lo que ambos quedan totalmente separados y dentro de la función, se le puede hacer lo que quiera al valor obtenido por parámetro (es como una copia de un archivo, el orinal queda intacto, mientras que la copia la puedes modificar como desees).

- **Por referencia**: contrario a lo expuesto anteriormente, en vez de realizar una copia del argumento (variable) local, se crea una referencia de él la cual pasa como parámetro a la función o procedimiento, dentro de esta, lo que se le haga a la referencia afectará a su origen (es como el acceso directo a un archivo).

El pase por valor ya lo conocemos, es que usualmente se utiliza. El **pase por referencia funciona** debido a que esta referencia es la dirección de memoria del argumento que se le pase a la función o procedimiento por parámetro, recordemos que una dirección de una variable contiene su valor también, es por ello que, cuando se modifica un valor pasado por referencia dentro de la función, este también se ve modificado desde su origen.

Creando un procedimiento que acepte parámetros por referencia, debemos utilizar *punteros* para ello, estos deben estar indicados en la estructura del procedimiento o función, con su tipo de dato y el operador * (que indica la declaración de un puntero). Cuando llamamos a la función, simplemente le pasamos la dirección de la variable a la que apunta el puntero (con el símbolo &). Refresquemos la memoria y lo leído con el siguiente ejemplo:

```
void Funcion(int *x) // El parámetro de esta función es un puntero llamado x del tipo int.
{
    *x = 100; // El * antes del nombre del puntero establece el valor del mismo.
}

int main()
{
    int variable = 25; // Variable local del tipo int.
    Funcion(&variable); // Le pasamos la referencia de variable local, la función le asigna
         el valor de 100 y después se despliega.
    cout << variable;

    return 0;
}
```

De manera general, el uso de parámetros por valor es más efectivo, rápido y requiere menos memoria, a diferencia del pase por referencia, además, evitamos tener que pensar en alterar el valor original de la variable que pasó su valor por referencia con el pase por valor.

16.5 Sobrecarga de funciones

Sobrecargar una función o procedimiento permite duplicar dichas funciones y procedimientos, pero con distintos parámetros, para ejecutar las mismas tareas pero con ligeras variaciones. Es preferible que estos procedimientos sobrecargados se diferencien por medio de distintos tipos de datos para los argumentos, la cantidad de los mismos, y la adición o recorte de alguna instrucción en su bloque de código.

```
void Suma(int x, int y)
{
    int resultado = x + y;
    cout << resultado;
}

void Suma(float x, float y)
{
```

```
    float resultado = x + y;
    cout << resultado;
}

int main()
{
    Suma(1, 2);
    Suma (3.14, 5.66);
}
```

Un error casual a la hora de ejecutar esto, es que se quieran sobrecargar funciones solamente cambiando su tipo de dato que devuelve, pero manteniendo tipos de argumentos y cantidad de los mismos similares.

```
int Despliegue (int a) { }
float Despliegue (int b) { }
double Despliegue (int c) { }
```

16.6 La función rand()

Generar números aleatorios nos permite probar distintas funciones o códigos, C++ requiere de la cabecera **cstdlib** para poder acceder a la función **rand()**.

```
#include <cstdlib>;
```

Algo que se debe de saber es que estos números aleatorios son pseudoaleatorios, esto quiere decir, en C++, una vez se termine la ejecución y se vuelva a ejecutar, los números que anteriormente aparecieron, volverán a aparecer, además. estos números son enteros.

Podemos hacer que los números generados por *rand()* estén contenidos dentro de un rango, para ello, utilizamos el operador **modulo (%)** y la siguiente estructura:

```
#include <cstdlib> // Cabecera para utilizar la función rand().

int main()
{
    int numero = rand(); // Se le asigna a la variable un número aleatorio.
    int numerorango = 1 + (rand() % 6); //Se le asigna a la variable un número aleatorio
        entre el 1 y el 6.

    cout << numero << ", " << numero rango << endl;
}
```

16.6.1 La función srand()

Esta función te permite **generar auténticos** números aleatorios. Esta función permite que se le pase un valor especial como parámetro, el cual es utilizado por la función *rand()*, este valor es un tipo de "fuente", de la cual rand toma valores aleatorios; si se pone un número fijo, realmente no pasa nada y no se generan números aleatorios auténticos, en cambio, si ponemos una fuente que esté constantemente cambiando su valor, podríamos utilizar y considerar esto como valores aleatorios reales, por ejemplo, la función **time()**, el siguiente ejemplo explica lo escrito: ç

```
#include <ctime> // Cabecera que acepta el tiempo actual del sistema.

int main()
{
    srand(time(0)); // El tiempo actual del sistema con parámetro 0 (segundos), es el valor
         especial del cual rand() obtendrá sus valores.

    for (int i = 1; i <= 10; i++)
    {
        cout << 1 + (rand() % 6) << endl;
    }

    return 0;
}
```

16.7 Funciones recursivas

Una **función recursiva** es aquella que se llama a sí misma, al igual que con los *ciclos while*, se debe escribir una condición o expresión de salida, para evitar un recursión indefinida.

La condición o expresión de salida suele llamarse **el caso base**, el cual hace que se evite una recursión infinita. Existen muchos ejemplos donde podemos utilizar estas funciones, como lo puede ser la famosa función matemática **factorial**, ejemplificaremos las funciones recursivas con este caso:

```
int factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}

int main()
{
    cout << factorial(5); //Despliega 5 * 4, 4 * 3, 3 * 2, 2 * 1, 1
}
```

17 Clases

17.1 Concepto de clases

Una **clase** es la base u origen de los objetos, si bien sabemos que cada objeto tiene características, conductas y son distintos cada objeto de otro, aunque consideremos que son "iguales" (recordando el ejemplo de las manzanas más arriba), las clases son lo que le dan dichas características y conductas a los objetos, podemos pensar a las clases como *planos* de los objetos, con el plano de una cosa, podemos crear dicha cosa múltiples veces, lo mismo pasa con las clases, una clase tiene un *nombre*, le agregas atributos y métodos (funciones y procedimientos), y de esa clase puedes crear objetos y darles valores a dichos atributos y métodos, los cuales definirán el estado de cada objeto.

Los **métodos** son el comportamiento de cada objeto, un método dentro de una clase puede compararse a lo que son las funciones o procedimientos, es decir, ambos son bloques de código que realizan una tarea en específico y pueden ser llamados, pero las funciones están contenidas en una clase.

Sabemos que una clase es la base de los objetos, las clases le proporcionan a cada objeto su identidad, atributos y comportamientos, podemos verlo un poco más claro con el siguiente ejemplo:

Queremos una clase para crear distintos objetos que son automóviles de distinto tipo, sabemos que los autos pueden variar de pequeños, medianos, grandes, camiones, motos, etcétera; además de que todos poseen un color, marca, modelo, entre más cosas, y su comportamiento regular es que se muevan y detengan; con esta información, podemos crear una clase como la siguiente:

Nombre: Auto
Atributos: Tipo, Marca, Modelo, Color
Comportamiento: Mover(), Detener()

Con esta clase hecha, podemos crear distintos objetos que representen una motocicleta, un automóvil para dos personas, una camioneta, un camión, entre otros. El proceso de creación de un objeto a partir de una clase se le llama **instanciación**, y una vez que el objeto es creado, a este también se le puede llamar **instancia**.

Ahora que tenemos una idea sobre las clases, toca crear una en código, su estructura básica es la siguiente:

```
class [nombre de clase]{
    // Atributos y métodos.
};
```

Para crear atributos y métodos dentro de una clase, es básicamente lo mismo a declarar variables y crear funciones o procedimientos en *main()*, pero podemos agregar un **especificador de acceso** a cada atributo y método, para limitar quienes pueden acceder o modificar la clase. Crearemos el ejemplo anterior de la clase Auto en código a continuación

```
#include <iostream>
using namespace std;

class Auto{
```

```
    public: // Especificador de acceso público, si no se indica, public es el especificador
         por defecto.
        string Tipo; // 4 atributos.
        string Marca;
        string Modelo;
        string Color;

        void Mover (){ // Método 1.
            cout << "Hacía adelante";
        }

        void Detener(){ // Método 2.
            cout << "Detenido";
        }
};
```

Las clases deben de crearse antes de la función *main()*, como las funciones o procedimientos, una vez son creadas, para instanciarlas dentro de main, debemos escribir primero el nombre de la clase, y luego el nombre de la instancia u objeto que queramos.

```
Auto camioneta;
```

Si queremos asignarle valores a los atributos del objeto, o llamar a alguno de sus métodos, debemos utilizar le nombre del objeto creado, punto, y el nombre del atributo o método que queramos usar.

```
camioneta.Tipo = "grande";
camioneta.Color = "Rojo";
camioneta.Mover();
```

17.2 Conceptos de POO

17.2.1 Objetos

La **Programación Orientada a Objetos** (OOP por sus siglas en inglés) es una forma de programara la cual busca que sea más cercana a la realidad. En este paradigma, cada cosa en el mundo real es llamado **objeto** en la programación, es decir, una manzana verde es un objeto, una manzana roja podría ser ese mismo objeto, pero es distinto ya que esta tiene la propiedad o característica de ser color rojo, por lo tanto, son dos objetos independientes con una **identidad** propia.

A estas características o propiedades se les llaman **atributos**, estos atributos pueden tener algún valor o no, los atributos representan el estado del objeto, una curiosidad es que los objetos pueden llegar a contener más objetos, pero aún así cada uno dentro de este será distinto a su contenedor.

Los objetos, como en el mundo real, tienen un **comportamiento**, no solo características, este comportamiento también puede definir al objeto, así como un auto se mueve y un celular vibra, un objeto puede realizar una acción.

Hasta ahora, sabemos que un objeto está compuesto por su identidad, atributos y comportamiento; un objeto siempre está contenido en su propia identidad, es por eso que entre objetos, ninguno es igual; los objetos *no siempre* representan un objeto en el mundo real,

una cuenta bancaria existe en la realidad, pero la puedes tocar, un objeto en programación puede representar dicha cuenta bancaria.

Table 7: Representación simple y visual de un objeto

Objeto: Persona
Nombre: "Luis" Edad: 21
Caminar() *Hablar()*

17.2.2 Abstracción

La **abstracción** es el pensamiento fundamental de la Programación Orientada a Objetos, así como pensábamos en una clase, la cual es un modelo o plano para los objetos, la abstracción es ese pensamiento que nos permite comprender como funciona una clase y objeto. La abstracción es la idea de como funciona algo, pero sin dar detalles de cómo o qué es ese algo, sabemos qué es un libro, pero no tenemos idea de cuántas páginas tiene, cómo es la portada, su autor, editorial, entre otras cosas, pero sabemos como funciona un libro, eso es la abstracción; lo mismo pasa con el ejemplo del automóvil, sabemos que los autos tienen marca, modelo, color, número de llantas, número de serie, tipo, etcétera, nosotros no sabemos toda esa información cuando hablamos de un auto, pero sabemos que se poseen todas esas características al instante, ese es el pensamiento que se sigue a la hora de crear una clase y su objeto. Esto nos funciona para crear una sola clase para muchos objetos, en vez de crear muchas clases para sus respectivos objetos. Más adelante se verán ejemplos de clases abstractas.

17.2.3 Encapsulamiento

Como mencionamos anteriormente, podemos especificar un acceso a los atributos y métodos de una clase, esto por medio de palabras reservadas, este proceso es llamado **encapsulamiento**, el cual consiste en mantener unidas y protegidas una serie de entidades dentro de otra. Los **especificadores de acceso** o **modificadores de acceso** presentes en C++ son:

- **public**: da acceso a cualquier parte de código fuera de la clase.
- **private**: no da acceso a nada fuera de la clase, solo a la clase y su contenido.
- **protected**: no da acceso a nada fuera de la clase, solo a su clase, su contenido y a una clase derivada de esta.

El encapsulamiento lo podemos interpretar como el hecho de que no queremos que los atributos de nuestros objetos sea accedidos por otros objetos o partes del código, solo pueden ser accedidos por medio de algún método que lo permita (otro ejemplo es que, teniendo un una clase CuentadeBanco, con un objeto que representa la cuenta bancaria de un usuario, no deseemos que cualquiera pueda acceder al saldo de dicha cuenta, solo el propietario de la

cuenta por medio de una autenticación). Si seguimos con el ejemplo del Auto, no queremos que otro objeto o sección del código modifique el tipo de Auto que tenga dicho objeto, para ello, podemos realizar unas modificaciones a nuestro código ejemplo anterior:

```
#include <iostream>
using namespace std;

class Auto{
    private: // Especificador de acceso público, si no se indica, public es el
        especificador por defecto.
        string Tipo; // 4 atributos.
        string Marca;
        string Modelo;
        string Color;
    public
        void Info(string T){
            Tipo = T;
        }
        void Despliegue(){
            cout << Tipo;
        }
        void Mover (){ // Método 1.
            cout << "Hacía adelante";
        }

        void Detener(){ // Método 2.
            cout << "Detenido";
        }
};
```

En nuestro ejemplo, hicimos privados todos los atributos para que nadie pueda asignarles valor o leer su contenido, y creamos dos procedimientos, uno para escritura y lectura respectivamente, así restringimos su acceso.

Una combinación de procedimiento y función para asignar y regresar el valor de un atributo que es privado es **set** y **get**, el primero obtiene el valor del atributo y se lo asigna, y el segundo es una función que regresa el valor del atributo que acaba de ser asignado, se ve mas claro con el siguiente ejemplo:

```
#include <iostream>
using namespace std;

class Nombre{ // Se crea una clase.
    private:
        string nombre; // Se crea un atributo privado.
    public:
        void setNombre(string n){ // Se crea procedimiento público que le asigna un valor
            al atributo privado.
            nombre = n;
        }
        string getNombre(){ // Se crea una función pública que regresa el valor del
            atributo privado.
            return nombre;
        }
};
```

```
int main(){
    string nom; // Se crea una variable string.
    cin >> nom; // Se le asigna un valor a la variable.
    Nombre obj; // Se instancia un objeto de la clase Nombre.
    obj.getNombre(nom); // Se utiliza su procedimiento público.
    cout << obj.setNombre() << endl; // Se despliega el nombre del atributo privado sin
        modificarlo o acceder a él directamente.

    return 0;
}
```

17.2.4 Herencia

La **herencia** consiste básicamente en crear clases a partir de otras, esto conlleva el pasarle sus atributos y métodos. La clase que recibe la herencia es llamada **clase derivada**, mientras que la clase que hereda su contenido es llamada **clase base** (p.e: la clase Padre (base) puede heredar su contenido a la clase Hijo (derivada)); la clase derivada hereda todo el contenido de la clase base, a su vez, la clase derivada puede contener su propio contenido. Para derivar una clase de otra, se sigue la siguiente sintaxis:

```
class [nombre clase derivada] : [especificador de acceso] [nombre clase base] {}
```

Se utiliza un **especificador de acceso** debido a que, todos los miembros públicos de la clase base pasarán a ser públicos en la clase derivada, esto deja de lado entonces todos aquellos miembro que sean privados o protegidos.

```
#include <iostream>
using namespace std;

class Madre{ // Clase base.
    public:
        Madre(){} // onstructor de la clase base.
        void MensajeMadre(){ // Procedimiento público propio y heredado a la clase derivada
            .
            cout << "Hola";
        }
    private:
        void MensajePrivadoM(){ // Procedimiento propio debido a que es privado.
            cout << "Hola privado";
        }
};

class Hija : public Madre{ // Clase derivada.
    public:
        Hija(){} // Constructor de la clase derivada.
        void MensajeHija(){ // Procedimiento propio de esta clase.
            cout << "Que tal";
        }
};
```

Al hacer esta herencia, la clase Hija puede acceder al procedimiento público de la clase Madre, esto se hace al crear un objeto Hija. Una clase derivada también puede nacer de dos o más clases bases.

```
class Nieta : public Madre, public Padre, public Abuela, public Abuelo {}
```

Las clases derivadas **no pueden heredad** lo siguiente:

- Funciones **friend**.
- Operadores sobrecargados o redefinidos.
- Constructores y destructores.

17.2.5 Especificadores de acceso de clases derivadas (miembros protegidos)

Aquí simplemente vamos a repasar el uso de los **especificadores de acceso** a la hora de heredar.

- **public**: todos los atributos y métodos públicos de la clase base pasan públicos a la clase derivada. Los atributos y métodos privados de la clase base continúan privados y la clase derivada no puede acceder a ellos, pero puede asignarles valores o desplegarlos por medio de métodos públicos.
- **private**: todos los atributos y métodos públicos y protegidos de la clase base pasan privados a la clase derivada.
- **protected**: todos los atributos y métodos públicos protegidos de la clase base pasan protegidos a la clase derivada.

Hay que recalcar que el especificador *public* es el más común; si una clase derivada no se le especifica un acceso, la herencia automáticamente es **privada**.

```
class Hija : Madre {} // La clase Hija tiene un especificador privado por defecto.
```

Constructor y destructor de clases derivadas

Cuando se crea una clase que hereda a otra, el constructor y destructor de la clase base no es heredada, sin embargo, cuando un objeto de la clase derivada es creado, se le habla al constructor de la clase base, lo mismo pasa con el destructor cuando el objeto es eliminado. El siguiente código explica un poco el llamado de constructores y destructores de clases base y derivadas.

```
#include <iostream>
using namespace std;

class Madre{
    public: // Constructor y destructor de Madre despliegan mensaje.
        Madre(){
            cout << "Constructor de Madre";
        }
        ~Madre(){
            cout << "Destructor de Madre"
```

```
        }
};

class Hija : public Madre{
    public: // Constructor y destructor de Hija despliegan mensaje.
        Hija(){
            cout << "Constructor de Hija";
        }
        ~Hija(){
            cout << "Destructor de Hija";
        }
};

int main(){
    Madre madre; // Se despliega el mensaje del constructor de Madre cuando el objeto se
        crea, despliega el mensaje del destructor cuando se termina el programa.
    Hija hija; // Se despliega el mensaje del constructor Madre y después el de Hija cuando
         el objeto se crea, despliega el mensaje del destructor de Hija y luego el de Madre
         cuando se termina el programa
    return 0;.
}
```

17.2.6 Polimorfismo

Polimorfismo significa **tener varias formas**, para explicar el polimorfismo en C++, haremos unas analogías:

Cuando nosotros vamos al doctor, nosotros nos convertimos en un *paciente* con características y comportamientos, cuando vamos a la escuela, nos convertimos en un *estudiante* con rasgos y acciones distintas, cuando vamos a una tienda, nos convertimos en un *cliente*, esto significa que, dependiendo de la circunstancia, en programación, podemos tener una clase "Vehiculo" con un método *Andar*, si heredamos esa clase a dos clases derivadas Bicicleta y Automovil, ambas tendrán la función Andar, pero si nos ponemos a pensar, un vehículo en sí, puede ir hacía enfrente, atrás, izquierda y derecha, un automóvil y bicicleta siguen esa lógica, pero un auto tiene cuatro llantas y un motor, una bici tiene dos llantas y pedales, esto significa que, dependiendo el objeto y la circunstancia, Vehiculo adopta una forma u otra, de eso se trata el polimorfismo. Un ejemplo con código

```
#include <iostream>
using namespace std;

class Enemigo{ // Clase base.
    protected: // Atributo protegido, las clases derivadas de esta tendrán acceso a este
        atributo.
        int PoderAtaque;
    public:
        void setPoderAtaque(int a){ // Método público que le asigna un valor al atributo.
            PoderAtaque = a;
        }
};

class Monstruo : public Enemigo{ // Clase derivada 1.
```

```
    public:
        void Atacar(){ // Monstruo tiene su método público Atacar con cuerpo propio.
            cout << "Monstruo - " << PoderAtaque << endl;
        }
};

class Ninja : public Enemigo{ // Clase derivada 2.
    public:
        void Atacar(){ // Ninja tiene su método público Atacar con cuerpo propio.
            cout << "Ninja - " << PoderAtaque << endl;
        }
};

int main(){
    Ninja n;
    Monstruo m; // Se crean objetos de cada clase derivada.

    Enemigo *enemigo1 = &n;
    Enemigo *enemigo2 = &m; // Se crean dos punteros Enemigo que apuntan a la dirección de
        cada objeto Ninja y Monstruo creados anteriormente.

    enemigo1 -> setPoderAtaque(30);
    enemigo2 -> setPoderAtaque(40);// Se le asigna un valor de Ataque a cada objeto Ninja y
         Monstruo por medio del punto y la clase base.

    n.Atacar();
    m.Atacar(); // Se llama al método que despliega el ataque de cada enemigo.
}
```

Lo que ocurre en el código de arriba, es que buscamos crear un juego en el cual habrá una variedad de enemigos que atacaran al jugador, este ataque que tendría cada enemigo tiene un valor de daño que le harán al jugador y un tipo de ataque, para ello, creamos una clase base Enemigo, de la cual nacerán todos los demás tipos de enemigos, esta clase base tiene dos clases derivadas: Ninja y Monstruo, entonces, cuando creamos las clases derivadas, estas heredan el método de asignar el poder de ataque de cada enemigo, además, a cada clase le creamos un método donde se despliega el poder de ataque que tiene el enemigo y el tipo de ataque que tiene; volteamos a ver a *main()*, aquí creamos un objeto Ninja y Monstruo, el puntero Enemigo que apunta hacía la dirección de memoria de estos objetos recientemente creados funcionan para asignarle un valor de poder de ataque a cada objeto por medio de la clase base Enemigo y finalmente se despliega el poder y tipo de ataque de cada enemigo.

Podría pensarse que está de más utilizar punteros en el ejemplo anterior, pero es normal y usual utilizar punteros para trabajar con polimorfismo, también podría pensarse que con crearse un objeto de cada clase derivada y llamar a las funciones heredadas es suficiente, pero el punto del polimorfismo es que los objetos de las clases derivadas nazcan de la clase base y que los métodos se utilicen igual por medio de la clase base, eso explica la aparición de punteros en el previo ejemplo, dejaremos otro para reforzar:

```
#include <iostream>
using namespace std;

class Persona {
```

```
    private:
        string nombre; // Atributos privados.
        int edad;
    public:
        Persona(string n, int e) :nombre(n), edad(e) {} // Constructor que inicializa los
            atributos.
        ~Persona(){} // Destructor.
        void InfoPersona() { // Procedimiento público que despliega el valor de los
            atributos.
            cout << "Me llamo " << nombre << "y tengo " << edad << " años\n";
        }
        virtual void Caminar() { // Método virtual que despliega un mensaje predeterminado.
            cout << "Persona camina";
        }
};

class Nino : public Persona { // Clase heredada1.
    private:
        string dulcefav; // Atributos privados.
    public:
        Nino(string n, int e, string df) : Persona(n, e) { dulcefav = df; } // Constructor
            que inicializa el atributo de esta clase y los de la clase base.
        ~Nino(){} // Destructor.
        void Caminar() { // Procedimiento virtual sobrecargado para esta clase derivada.
            InfoPersona(); // Llama al procedimiento de la clase base.
            cout << "Soy un niño y camino mientras como " << dulcefav << "\n";
        }
};

class Adolecente : public Persona {
    private:
        string estudios; // Atributo privado.
    public:
        Adolecente(string n, int e, string es) : Persona(n, e) { estudios = es; } //
            Constructor que inicializa el atributo de esta calse y los de la clase base.
        ~Adolecente() {} // Destructor.
        void Caminar() { // Procedimiento virtual sobrecargado para esta clase derivada.
            InfoPersona(); // Llama al procedimiento de la clase.
            cout << "Soy un adolecente que camina mientras estudia " << estudios << "\n";
        }
};

int main() {
    Persona* nin = new Nino("alan", 10, "chocolate"); / Crea un puntero Persona hacía la
        clase Nino pasándole parámetros.
    Persona* ado = new Adolecente("david", 16, "matematicas"); // Crea puntero Persona hací
        a la clase Adolecente pasándole parámetros.
    ado->Caminar();
    nin->Caminar(); // Invoca al procedimiento virtual sobrecargado de cada clase derivada.
    system("pause");
    return 0;
}
```

17.3 Creando una clase en un archivo separado

Como hemos visto en ejemplos más arriba, podemos crear clases en nuestro archivo de código, pero también podemos **crear clases en archivos separados** a nuestro fichero principal, y referenciar nuestra clase separada en este fichero. Dependiendo del programa donde estés desarrollando código C++, la forma de generar la clase separada es distinta, lo que debes de saber es que una clase separada está constituida por dos archivos:

- **[nombre de la clase].h**: es la cabecera del archivo. Este contiene la declaración de la clase (*prototipo de la clase*) y declaración de variables, si es necesario. En este archivo también se encuentra el constructor de la clase.

```
// Archivo clase.h.

class ClaseP{
    public:
        ClaseP(){ } // Constructor.

    // Atributos.
}
```

- **[nombre de la clase].cpp**: es el código fuente del archivo. Aquí se escribe la implementación del los atributos, variables y métodos de la clase.

```
// Archivo clase.cpp.

#include "ClaseP"

ClaseP::ClaseP() {}
```

Sabemos que en el archivo **.h** de la clase, se declaran los atributos, constructores y métodos, pero solo eso, no se implementan o trabajan todavía, en el archivo **.ccp** debemos incluir en la cabecera al archivo *.h*, y con ello, ya podemos trabajar los atributos, constructores y métodos, sin embargo, no podemos empezar a codificar simplemente escribiendo el nombre de un método y dándole un cuerpo a este, sino que debemos de referenciar del archivo .h al .ccp todo de atributo, constructor y método que en ese archivo haya, es por eso que en el ejemplo de arriba, hay dos puntos (**::**) entre el nombre de la clase y su constructor, estos dos puntos son llamados **operadores de resolución de alcance**, debemos indicar estos dos puntos cada que se dé cuerpo a un método o constructor.

```
// Referencia de un método o constructor de .h a .cpp.
[fuente]::[referencia]() {}
[padre]::[hijo] {}
Auto::Mover() {}
```

Finalmente, para utilizar la clase en archivo separado en nuestro archivo principal con el main(), debemos incluir el fichero .h en el código principal.

```
#include <iostream>
#include "ClaseP.h"
using namespace std;
```

```
int main(){
    ClaseP obj;

    return 0;
}
```

17.4 Miembros especiales

17.4.1 Constructores

Los **constructores** son un método especial de cada clase, no pertenecen al tipo de funciones ni procedimientos, no regresan nada. Estos métodos son invocados cada que se crea una nueva clase y, dentro de su código, puede mostrar o hacer lo que sea. A continuación su estructura:

```
#include <iostream>
using namespace std;

class miClase{
    miClase(){
        // Código.
    }
};
```

Los constructores son útiles para inicializar atributos de la clase, por lo general, los constructores no tienen argumentos, pero podemos ponérselos, y con ello, asignar valores de inicio a los atributos de nuestra clase, como por ejemplo:

```
#include <iostream>
using namespace std;

class Nombre{ // Se crea una clase.
    Nombre(string nom){ // Constructor de la clase con un argumento.
        setNombre(nom); // Se llama al procedimiento de asignación de valor al atributo.
    }

    private:
        string nombre; // Se crea un atributo privado.

    public:
        void setNombre(string n){ // Se crea procedimiento público que le asigna un valor
            al atributo privado.
        nombre = n;
        }
        string getNombre(){ // Se crea una función pública que regresa el valor del
            atributo privado.
        return nombre;
        }
};
```

Entonces, si podemos crear argumentos en los constructores, cuando una objeto se crea a partir de una clase, debemos pasarle algún valor como parámetro para que el constructor lo reciba y haga lo que tenga que hacer, como se muestra a continuación:

```
#include <iostream>
using namespace std;

class Nombre{ // Se crea una clase.
    Nombre(string nom){
        setNombre(nom);
    }

    private:
        string nombre; // Se crea un atributo privado.

    public:
        void setNombre(string n){ // Se crea procedimiento público que le asigna un valor
            al atributo privado.
        nombre = n;
        }
        string getNombre(){ // Se crea una función pública que regresa el valor del
            atributo privado.
        return nombre;
        }
};

int main(){
    string nom; // Se crea una variable string.
    cin >> nom; // Se le asigna un valor a la variable.
    Nombre obj(nom); // Se crea el objeto y se le pasa como parámetro la variable local.
    cout << obj.setNombre() << endl; // Se despliega el nombre del atributo privado sin
        modificarlo o acceder a él directamente.

    return 0;
}
```

Nota: podemos tener múltiples constructores dentro de una clase con cantidad variada de argumentos para parámetros.

17.4.2 Destructores

Contrario a los constructores, los **destructores** son llamados cuando un objeto es destruido o eliminado, esta función es utilizada comúnmente para eliminar espacio de los punteros, liberar memoria, cerrar archivos que se estaban utilizando, etcétera. La estructura de un destructor es la siguiente:

```
#include <iostream>
using namespace std;

class Prueba{
    public:
        Prueba(){
            // Código.
```

```
        }

        ~Prueba(){
            // Código.
        }
};
```

Podemos ver que al inicio del nombre de la clase, se debe poner una virguilla o tilde ( ), los destructores deben declararse en el archivo *.h* de la clase y debe ser referenciado en el archivo *.ccp* con el *operador de resolución de alcance ::*. Los destructores no reciben parámetros, no puedes tener más de un destructor, no se pueden sobrecargar y no son obligatorios. Aquí un ejemplo:

```
// Archivo .h.

class Auto{ // Creando una nueva clase.
    public:
        Auto(); // Prototipo del constructor público.

        ~Auto(); // Prototipo del destructor público.
};


//Archivo .ccp.

#include <iostream> // Se incluye en la cabecera las entradas y salidas. de iostream
#include "Auto.h" // Se incluye en la cabecera el archivo .h de la clase.
using namespace std;

Auto::Auto(){ // Referenciado del constructor de .h a .ccp.
    cout << "Se ha creado un objeto de la clase Auto" << endl; // Contenido del constructor
        .
}

Auto::~Auto(){ // Referenciado del destructor de .h a .ccp.
    cout << "Se ha destruido un objeto de la clase Auto" << endl; // Contenido del
        destructor.
}


// Archivo principal.
#include <iostream>
#include "Auto.h" // Se incluye en la cabecera el archivo .h de la clase.
using namespace std;

int main(){
    Auto obj; // Despliega el contenido del constructor, al ser creado el objeto y el
        contenido del destructor, cuando el programa termina, el destructor es invocado.

    return 0;
}
```

17.5 Palabras reservadas

17.5.1 Friend

Si hacemos memoria, sabemos que gracias a el **especificador de acceso private** los atributos solo puede ser accedidos por miembros o métodos de la misma clase, no al resto del código, sin embargo, existe una palabra reservada que permite que un procedimiento o función fuera de la clase acceda a un atributo de la misma en específico, como se puede ver en el siguiente ejemplo:

```
#include <iostream>
using namespace std;

clase Persona{
    public:
        Persona(){
            edad = 0;
        }
    private:
        int edad;

    friend void AsignarEdad(Persona &persona); // Procedimiento amigo que accede al
        atributo privado edad.
};

void AsignarEdad(Persona &per){ // Procedimiento externo a la clase Persona que accede y
    modifica el valor del atributo privado edad.
    per.edad = 21;
    cout << per.edad << endl;
}

int main(){
    Persona persona; // Se crea un objeto Persona.
    AsignarEdad(persona); // Se le pasa por parámetro el objeto persona al procedimiento.

    return 0;
}
```

Podemos notar que, dentro de la clase, hay un atributo privado llamado *edad*, el constructor lo inicializa en 0, fuera de la clase, hay un procedimiento llamado *AsignarEdad*, su argumento es un objeto *Persona* al cual le pasamos un parámetro por referencia (obligatoriamente) utilizando el símbolo **&**, dentro de dicho procedimiento, se le asigna un valor al atributo privado y se despliega su contenido, esto es posible gracias a la palabra reservada **friend**, esta instrucción permite el acceso de procedimientos o funciones externas de una clase a un atributo de la misma, dentro de Persona está el prototipo del mismo procedimiento externo, solo que al principio está la palabra reservada, y dentro de sus argumentos se pasan parámetros por referencia.

```
friend [void o tipo de dato de retorno] [nombre de la función] (parámetros)
```

La instrucción *friend* es utilizado para acceder a atributos o modificar sus valores de dos o más clases que necesitan interactuar brevemente entre ellas. Se puede ir más allá y crear una clase completa friend.

17.5.2 This

Supongamos que tenemos nuestra clase *MiClase*, la cual tiene un atributo privado del tipo string llamado "mensaje", también tenemos un método que tiene una variable local string llamada "mensaje", dentro del método, se despliega el contenido de mensaje. Esta situación puede ocurrir con variables locales de un método y atributos de una clase que tienen el mismo nombre, podríamos evadir el problema cambiando nombres y listo, pero lo que pretendemos hacer es mantener los mismos nombres, pero acceder a cada variables respectivamente. Si desplegamos con *cout* el contenido de la variable local no hay problema, pero no podríamos saber cual es el contenido del atributo con el mismo nombre ya que el método prioriza sus variables locales, utilizaremos la palabra reservada **this** para acceder al contenido del atributo. *this* es una palabra reservada que representa un puntero que apunta hacía los atributos de una clase desde sus métodos, esto se utiliza por lo general para solucionar la situación presentada hace algunas líneas y para la sobrecarga de operadores.

```
#include <iostream>
using namespace std;

class MiClase{
    public:
        MiClase(int a) : var(a) {}
        void Imprimir(){
            int var;
            cin >> var;
            cout << var << endl; // Imprime la variable local.
            cout << this->var << endl; // Imprime el atributo de la clase.
            cout << (*this).var << endl; // Imprime el atributo de la clase.
        }
    private:
        int var;
};

int main(){
    MiClase obj(50);
    obj.Imprimir();

    return 0;
}
```

Al ser *this* un puntero, para acceder a su contenido es necesario utilizar la **flecha operadora de operación (-¿)** o la instrucción especial **(*this).** Hay que aclarar también que solo los métodos (funciones miembro) tienen un puntero *this*, y la palabra *friend* no posee un puntero de este tipo.

17.6 Operadores de selección

Vamos a plantear una situación que puede ocurrir cuando se crear clases en archivos separados: supongamos que creamos tres clases llamadas "abuelo", "padre" e "hijo", abuelo tiene en su prototipo una estructura llamada "poo", ahora, queremos incluir en la cabecera de padre al archivo *.h* de abuelo, finalmente, incluimos en la cabecera de hijo los archivos

.h de abuelo y padre, una vez sea compilado el programa, tendremos como resultado dos estructuras poo en nuestro programa, causando un error de compilado.

```
// Archivo .h de abuelo.
struct poo{
    int atri;
}


// Archivo .h de padre.
#include "abuelo.h"


// Archivo .h de hijo.
#include "abuelo.h"
#include "padre.h"

// Resultado de la compilación.
struct poo{
    int atri;
}

struct poo{ // Doble estructura creada a causa de doble inicialización de la misma
    estructura en dos archivos distintos.
    int atri;
}
```

El error se da porque, como bien sabemos, al programar en C o C++, el código que estamos creando no va dirigido a la computadora, sino al compilador, el compilador tiene una serie de **directrices** o **centinelas (guard)** que le sirven para no repetir código y así evitar errores, tal es el caso de **#include**, *include* lo que hace es decirle al compilador que agregue un fichero a otro ya existente, para su posterior uso, si el fichero a referenciar contiene una estructura que pueda repetirse y, como consecuencia, duplicarse, genera un error de compilación.

Para evitar el error del código de arriba, utilizaremos una directriz o centinela llamada **#ifndef [nombre]- #endif** y **#define [nombre]**, el cual asegura dentro del compilador, una vez hemos mandado el código, que no exista una estructura repetida en diversos archivos, para evitar un duplicado, *#ifndef [nombre]* actua como un if cualquiera, el nombre puede ser definido por el usuario como este desee, algunos escriben el nombre de la clase, guión bajo, y una H, haciendo referencia al archivo .h de la clase en cuestión, es recomendable utilizar mayúsculas para estos nombres. *#define* se encarga de declarar la estructura que se procura no duplicar o que se procura evaluar su existencia. Finalmente, *#endif* cierra la directriz o centinela. Corregiremos el código de arriba:

```
// Archivo .h de abuelo.
#ifndef CABUELO_H // Directriz para corroborar existencia de la estructura.
#define CABUELO_H // Declara la estructura.

struct poo{
    int atri;
}
```

```
#endif // Cierra la directirz.


// Archivo .h de padre.
#include "abuelo.h"


// Archivo .h de hijo.
#include "abuelo.h"
#include "padre.h"

// Resultado de la compilación.
struct poo{
    int atri; // Como se creó inicialmente la estructura en abuelo.h, padre.h no la
        referencia gracias a la directriz, finalmente, la compilación en hijo.h crea una
        sola estructura sin duplicado.
}
```

Una vez más, podemos utilizar **punteros** del tipo de una clase y para acceder a la dirección de memoria de un objeto creado a partir de una clase. Si queremos acceder a un método de la clase de la cual se hizo un puntero objeto, debemos usar el **flecha operadora de selección (-¿)**.

```
Auto obj;
Auto* puntero = &obj;
obj -> Imprimir();
```

17.7 Objetos Const

Una **constante** es una expresión la cual debe ser inicializada con un valor a la hora de su declaración, la cual no se puede modificar durante la ejecución del programa, es una variable que funciona como una constante matemática o física (Pi, Euler, la gravedad). Debemos usar la palabra reservada **const**, seguido del tipo de dato, nombre de la variable y su valor.

```
const [tipo de dato] [nombre variable] = [valor];
```

Se pueden crear objetos de clases que sean constantes, es decir, una vez creado el objeto, no se puede modificar ninguno de sus parámetros o métodos durante su tiempo de vida. Cuando se crea, debe ser inicializado por medio de su constructor; si tiene un constructor con parámetros, se le asignan valores a dichos parámetros, si no tiene parámetros el constructor y esté no tiene argumentos, se crea el objeto simplemente sin paréntesis ni valores, y si la clase de origen del objeto no tiene constructor, genera un error de compilación.

```
const Auto obj;
```

```
#include <iostream>
using namespace std;

class Auto{
    public:
        Auto(){ } // Constructor vacío.
```

```
        Auto(string m){ // Constructor que despliega mensaje.
            cout << m;
        }
        ~Auto(){ // Destructor.
            cout << "Eliminado";
        }
};

int main(){
    const Auto obj; // Objeto constante de la clase Auto inicializado por medio del
        constructor vacío.
    const Auto obj2("Este mensaje no cambiará nunca."); // Objeto constante de la clase
        auto inicializado por parámetro con el constructor con un argumento.
}
```

Algo que se debe tomar en cuenta, es el hecho de que los objetos no constantes pueden llamar a las funciones o métodos no constantes de una clase, esto quiere decir que, un objeto constante no puede acceder a las funciones y métodos no constantes, si un objeto constante invoca a un método o función no constante, ocurre un error de compilación; es entonces que debemos declarar funciones constantes dentro de una clase, para ello, simplemente agregamos la palabra reservada *const* al final de cada prototipo y definición de método o función que tenga una clase.

```
//Archivo .h.
#ifndef CLASEPRUEBA_H // Si no existe esta clase, entonces.
#define

class Prueba{
    public:
        void Despliegue() const; // Método constante.
        void Mensaje(); // Método no constante.
};

#endif // Fin de centinela.


// Archivo .ccp.
#include "Prueba.h"
#include <iostream>
using namespace std;

void Prueba::Despliegue() const { // Método constante.
    cout << "Mensaje 1" << endl;
}
void Prueba::Mensaje(){ // Método no constante.
    cout << "Mensaje 2" << endl;
}


// Archivo principal.
#include "Prueba.h"
#include <iostream>
using namespace std;
```

```
int main(){
    Prueba obj; // Objeto no constante.
    const Prueba obj2; // Objeto constante.

    obj.Mensaje(); // El objeto no constante solo muestra el método no constante.
    obj2.Despliegue(); // El objeto constante solo muestra el método constante.

    return 0;
}
```

17.8 Miembros inicializador

En ocasiones, necesitaremos de variables constantes dentro de una clase, pero en el momento en el que estamos escribiendo el código de la clase, nos damos cuenta que requerimos que el usuario ingrese el valor para la variable constante, si acudimos a ejemplos pasados donde asignamos valores a las variables de una clase por medio de un método o el constructor, nos daremos cuenta que esto genera un error de compilación, precisamente porque la constante no fue inicializada y porque asignarle un valor por entrada va en contra de la filosofía de las constantes.

Es entonces que debemos recurrir a ciertos tipos de **miembros inicializadores**, estos miembros son otra forma de asignar valores a variables, entonces sus ejemplos nos topamos con las **listas inicializadoras**, estas van al lado del constructor y se separan de este por medio de dos puntos (**:**), como lo muestra su estructura:

```
[constructor de clase]() : atributo1(valor), atributo2(valor) {}
```

Con este miembro lista, podemos resolver el problema que se presentó algunas líneas arriba, debemos recalcar que estos miembros lista no terminan con punto y coma. Podemos ver esto más claro con el siguiente ejemplo:

```
// Archivo Persona.h.
#ifndef PERSONA_H
#define
class Persona{
    public:
        Persona(int a, string b); // Constructor con dos argumentos.
    private:
        int edad; // Atributos.
        const string nombre;
};
#endif


// Archivo Persona.ccp.
#include "Persona.h"
#include <iostream>
using namespace std;

Persona::Persona(int a, int b) : edad(a), nombre(b) // Se le pasan los parámetros a y b al
     constructor por medio de una lista inicializadora.
```

```
{
    cout << nombre << ", " << edad << endl; // Despliega los atributos.
}


// Archivo principal.
#include "Persona.h"
#include <iostream>
using namespace std;

int main(){
    Persona obj(21, "Luis"); // Crea objeto de Persona y le pasa parámetros al constructor.

    return 0;
}
```

Nota: las listas inicializadoras pueden ser utilizadas para variables o miembros no constantes, depende de uno utilizar este método de asignación.

17.9 Composición

La **composición** consiste en armar un objeto completo a partir de objetos más pequeños y simples, por ejemplo, un automóvil está compuesto de piezas de metal, un motor, llantas, engranajes, ventanas y demás piezas, en programación, dicho auto podría estar compuesto de una sola clase que contenga atributos del tipo de otras clases (clase Auto tiene el atributo llanta, la cual es del tipo Llanta). Podemos apreciarlo más claramente con el siguiente ejemplo:

```
#include <iostream>
using namespace std;

class Cumpleanos{
    public:
        Cumpleanos(int m, int d, int a) : mes(m), dia(d), anio(a) {} // Constructor con
            argumentos a que inicializan los atributos.
        void ImprimirCumple(){ // Método que imprime la fecha de cumpleaños.
            cout << dia << "/" << mes << "/" << anio << endl;
        }
    private: // Atributos privados de una fecha de cumpleaños.
        int mes;
        int dia;
        int anio;
};

class Persona{
    public:
        Persona(string n, Cumpleanos c) : nombre(n), cumple(c) {} // Constructor con
            argumentos que inicializa los atributos.
        void ImprimeDatos(){ // Método que imprime los datos de la persona.
            cout << nombre << endl;
            cumple.ImprimirCumple(); // El atributo tipo Cumpleanos invoca al método de
                imprimir la fecha de cumpleaños.
        }
```

```
    private: // Atributos privados de una persona.
        string nombre;
        Cumpleanos cumple; // Podemos hacer que un atributo sea del tipo de una clase,
            automáticamente, dicho atributo obtiene todos los atributos contenidos de la
            clase tipo que se le asignó, volviendo esto una estructura más compleja.
};

int main(){
    Cumpleanios cumpleanios(2, 19, 2001); // Crea un objeto Cumpleanios y se le asignan
        valores al objeto.
    Persona persona("Luis", cumpleanios); // Se crea un objeto Persona y se le pasa un
        valor string normal y el objeto creado previamente.
    persona.ImprimirInfo(); // Se llama al método de impresión de Persona.

    return 0;
}
```

En resumen, si un objeto de la vida real que estamos intentando imitar es muy complejo, podemos dividirlo en pequeños objetos y más simples para componerlo, volviendo esta tarea algo más simple y sencillo de entender, solo no hay que perder el hilo de todos los objetos piezas que compondrán a uno complejo mayor.

17.10 Sobrecarga de operadores

Los operadores en C++ funcionan únicamente con los tipos de datos integrados en el lenguaje y con expresiones lógicas que reconoce el mismo, pero a la hora de estar creando constantemente clases y objetos, quizás queramos compararlos o realizar alguna acción aritmética con ellos, los operadores no podrían funciona, pero C++ cuenta con la opción de **sobrecargar o redefinir operadores**, para que adopten una nueva forma de trabajar. Los operadores que pueden ser redefinidos o sobrecargados son:

Table 8: Operadores que aceptan sobrecarga

+	-	*	/	%	∧
&	|	∼	!	,	=
<	>	<=	>=	++	–
<<	>>	==	!=	\	||
+=	-=	/=	%=	∧=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]

En el caso de clases, sobrecargar un operador funciona para sumar objetos, cosa que con los operadores convencionales no se podría. Para **sobrecargar** un operador lo podemos hacer desde la clase donde se busque utilizarlo, se escribe el nombre de la clase, la palabra reservada **operator**, el operador que se busca redefinir y paréntesis con los argumentos que se deseen.

```
[nombre clase] operator[operador a sobrecargar](parámetros) {}
```

Tenemos el siguiente ejemplo:

```
#include <iostream>
using namespace std;

class Persona {
    public:
        int edad; // Atributo público edad.

        Persona() {} // Constructor vacío.
        Persona(int e) : edad(e){} // Constructor con un argumento al que se le pasa un
            valor por parámetro.
        Persona operator+(Persona& per) { // Sobrecarga del operador +, se le pasa un
            objeto Persona por referencia.
            Persona p; // Se crea un objeto Persona p.
            p.edad = this->edad + per.edad; // Se suma la edad el objeto Persona que se cree
                 más otro objeto persona y se le asigna a la edad de otro objeto Persona.
            return p; // El objeto Persona es retornado.
        }
};

int main() {
    Persona persona1(20), persona2(40); // Se crean dos objetos Persona y se les asigna un
        valor de edad. La sobrecarga del operador no se ejecuta automáticamente, solo
        cuando el compilador detecte una suma con un objeto Persona.
    Persona personaC = persona1 + persona2; // Se crea un objeto Persona y se le asigna la
        suma de dos objetos Persona, se llama a la sobrecarga del operador +.
    cout << personaC.edad; // Despliega 60.

    return 0;
}
```

17.11 Clases abstractas

Si recordamos el significado de abstracción visto más arriba, tenemos que lograr que cada objeto tenga un comportamiento distinto dada una situación; para la programación, cada objeto debe tener un código distinto para el mismo método, para ello, utilizamos las **funciones virtuales**. Una *función virtual* es método que hace aparición en una clase base, este método, al heredar a otras clases, aparece y puede ser redefinida por las clases derivadas. Su estructura es la siguiente:

```
virtual [void o tipo retorno] [nombre] ([parámetros]) {}
```

Para la instrucción de arriba, la clase que estamos creando es abstracta, pero podemos crear objetos a partir de esta, heredarla a otras clases y crear objetos con la clase derivada. Un ejemplo: tenemos una clase base llamada *Persona*, la cual tiene de atributos su *nombre* y *edad*, con un método *Caminar()* el cual despliega un mensaje, este método es virtual; ahora, crearemos dos clases derivadas de Persona las cuales tendrán los atributos de su clase base, pero desplegarán otro mensaje distinto al presentado en Caminar de Persona:

```
#include <iostream>
using namespace std;
```

```
class Persona {
    private:
        string nombre; // Atributos privados.
        int edad;
    public:
        Persona(string n, int e) :nombre(n), edad(e) {} // Constructor que inicializa los
            atributos.
        ~Persona(){} // Destructor.
        void InfoPersona() { // Procedimiento público que despliega el valor de los
            atributos.
            cout << "Me llamo " << nombre << "y tengo " << edad << " años\n";
        }
        virtual void Caminar() { // Método virtual que despliega un mensaje predeterminado.
            cout << "Persona camina";
        }
};

class Nino : public Persona { // Clase heredada1.
    private:
        string dulcefav; // Atributos privados.
    public:
        Nino(string n, int e, string df) : Persona(n, e) { dulcefav = df; } // Constructor
            que inicializa el atributo de esta clase y los de la clase base.
        ~Nino(){} // Destructor.
        void Caminar() { // Procedimiento virtual sobrecargado para esta clase derivada.
            InfoPersona(); // Llama al procedimiento de la clase base.
            cout << "Soy un niño y camino mientras como " << dulcefav << "\n";
        }
};

class Adolecente : public Persona {
    private:
        string estudios; // Atributo privado.
    public:
        Adolecente(string n, int e, string es) : Persona(n, e) { estudios = es; } //
            Constructor que inicializa el atributo de esta calse y los de la clase base.
        ~Adolecente() {} // Destructor.
        void Caminar() { // Procedimiento virtual sobrecargado para esta clase derivada.
            InfoPersona(); // Llama al procedimiento de la clase.
            cout << "Soy un adolecente que camina mientras estudia " << estudios << "\n";
        }
};

int main() {
    Persona* nin = new Nino("alan", 10, "chocolate"); // Crea un puntero Persona hacía la
        clase Nino pasándole parámetros.
    Persona* ado = new Adolecente("david", 16, "matematicas"); // Crea puntero Persona hací
        a la clase Adolecente pasándole parámetros.
    ado->Caminar();
    nin->Caminar(); // Invoca al procedimiento virtual sobrecargado de cada clase derivada.
    system("pause");
    return 0;
}
```

```
#include <iostream>
using namespace std;

class Madre{ // Clase base.
    public:
        Madre(){} // Constructor de la clase base.
        virtual void Mensaje(){ // Procedimiento virtual que despliega un mensaje
            predeterminado.
            cout << "Mensaje predeterminado";
        }
    private:
        void MensajePrivadoM(){ // Procedimiento propio debido a que es privado.
            cout << "Hola privado";
        }
};

class Hija : public Madre{ // Clase derivada.
    public:
        Hija(){} // Constructor de la clase derivada.
        void Mensaje(){ // Procedimiento propio de esta clase.
            cout << "Que tal";
        }
};

int main() {
    Madre m = new Madre();
    Hija h = new Hija();
    m.Mensaje();
    h.Mensaje();

    return 0;
}
```

Con esto logramos hacer que, bajo ciertas circunstancias o dependiendo de la situación, una función virtual en una clase heredada tenga un bloque de código distinto al de la clase base.

Existe también las llamadas **funciones virtuales puras**, las cuales sus prototipos están igualadas a 0, esto significa que la clase donde está esta función pura se convierte completamente a una **clase abstracta**, dando como resultado una clase la cual no puede tener objetos instanciados, solo clases derivadas. Hay que aclarar que, si una clase base tiene una función virtual, todas las clases que son derivadas de esta, deberán *anular* (redefinir la función virtual pura en la clase derivada) a la función virtual pura de la clase base, si no se hace esto, ocurrirá un error de código y surgirán problemas cuando se instancie un objeto de la clase derivada donde no se anuló la función. Otro ejemplo de abstracción:

```
#include <iostream>
using namespace std;

class Enemigo{ // Clase base.
    protected: // Atributo protegido, las clases derivadas de esta tendrán acceso a este
        atributo.
        int PoderAtaque;
    public:
```

```
        void setPoderAtaque(int a){ // Método público que le asigna un valor al atributo.
            PoderAtaque = a;
        }
        virtual void Atacar(){ // Método virtual que todas las clases heredadas deben
            anular o redefinir.
            cout << "Ataque de enemigo";
        }
};

class Monstruo : public Enemigo{ // Clase derivada 1.
    public:
        void Atacar(){ // Método anulado o redefinido de la clase base.
            cout << "Monstruo - " << PoderAtaque << endl;
        }
};

class Ninja : public Enemigo{ // Clase derivada 2.
    public:
        void Atacar(){ // Método anulado o redefinido de la clase base.
            cout << "Ninja - " << PoderAtaque << endl;
        }
};

int main(){
    Ninja n;
    Mounstro m; // Se crean objetos de cada clase derivada.

    m.setPoderAtaque(30);
    n.setPoderAtaque(40); // Se le asigna un valor de Ataque a cada objeto Ninja y Monstruo
        por medio del punto y la clase base.

    m.Atacar();
    n.Atacar(); // Se llaman al método virtual y todas sus redefiniciones por medio de
        punteros.
}
```

18 Plantillas (Templates)

18.1 Funciones plantilla

Las **funciones plantilla** son utilizadas como base para repetir una tarea básica sin importar las variables y los tipos de estas. Esto aplica para casos donde debemos realizar operaciones matemáticas básicas, por ejemplo, una suma, podríamos crear una función que regrese la suma de x valores, pero dichos valores deben ser de un único tipo de dato, si quisiéramos utilizar esa función para otro tipo distinto de dato, no podríamos, la solución seria sobrecargar la función pero cambiando los tipos de datos a los que se necesiten.

```
int Suma(int x, int y){
    return x + y;
}

double Suma(dobule x, double y){
    return x + y;
}
```

Para crear una función plantilla que represente la operación Suma para todos los tipos de datos numéricos que posee C++, utilizamos la palabra reservada **template**, seguido del **tipo de definición**:

```
template <class T>
```

Lo que significa el contenido entre los <> es que se creará una clase (un tipo de dato) del tipo **T**, el tipo T es un tipo de dato genérico, es decir que, no representa ningún tipo de dato alguno, los representa a todos en general; se usa el nombre T porque podemos interpretar el funcionamiento de este como una variable, el tipo de dato genérico que creemos puede tener el nombre que gustemos, para efectos de este documento y de la fuente donde se estudió, se usa T.

```
#include <iostream>
using namespace std;

// Declaración de la plantilla de función con su tipo genérico.
template <class T> T Suma (T x, T y){
    return x + y;
}

int main(){
    cout << Suma(1,1) << endl; // Despliega 2.
    cout << Suma(3.14, 2.56) ; // Despliega 5.7.

    return 0;
}
```

El ejemplo anterior logra que podamos usar la misma función Suma para cualquier tipo de dato. Una alternativa a escribir una declaración de plantilla, es usando la palabra **typename**:

```
template <typename T>
```

18.2 Plantillas de funciones con múltiples parámetros

En caso de que se quieran utilizar más de un tipo genérico en la plantilla, se puede lograr por medio de utilizar la misma lógica que crear argumentos de una función con distinto tipos de datos:

```
int Operaciones(int x, double y) - T Suma(A x, B y)
```

Con lo anterior, resolvemos el problema de querer sumar dos tipos de datos numéricos distintos (con la plantilla de función con un único tipo genérico, ocurriría un error de compilación)

```
#include <iostream>
using namespace std;

template <typename A, typename B> A Suma (A x, B y){ / /Declaración de la plantilla de
    función con dos tipos genéricos.
    return a + b;
}

B Menor(A a, B b){ // Declaración de la función con tipos genéricos; regresa un resultado
    tipo B.
    return (a < b ? a : b);
}

int main(){
    int a = 5;
    double b = 4.666; // Declaración de variables.
    cout << Suma(1,1) << "\n"; // Despliega 2.
    cout << Suma(3, 2.56) << "\n"; // Despliega 5.56.
    cout << Menor(a, b) << << "\n"; // Despliega a.

    return 0;
}
```

18.3 Plantillas de clases

Así como hay funciones, se pueden crear clases plantilla con funciones o atributos de un tipo genérico.

```
template <typename A>
class MiClase {};
```

El siguiente ejemplo es una clase que tiene atributos y una función de un tipo genérico A en el mismo archivo donde está **main()**:

```
#include <iostream>
using namespace std;

template<typename A> class Numeros{ // Declaración de la plantilla de clase con un tipo
    genérico.
    private:
        A x, y; // Atributos privados del tipo genérico A.
```

```
    public:
        Numeros(A x, A y) : first(a), second(b){} // Constructor público que inicializa los
            atributos.

        A Mayor(){ // Función que regresa un tipo de dato genérico.
            return (x > y ? x : y);
        }
};
```

Sin embargo, podemos recordar que las clases pueden crearse en archivos separados (**.h, .cpp**); en el archivo cabecera, se declaran todos los atributos, procedimientos y funciones, en el archivo fuente, se desarrollan dichas declaraciones o prototipos, recordemos que se utiliza una sintaxis especial para desarrollar el prototipo de una función o procedimiento:

```
void Auto::Mover() {}
int Auto::Kilometraje() {}
```

Entonces, para crear una clase plantilla con más de un dato genérico, debemos declarar la plantilla en ambos archivos, y especificar el dato genérico en las funciones miembros al inicio y después del nombre de la clase, encerrado entre <>, de la clase.

```
// Archivo .h.
#ifndef NUMEROS_H
#declare

template<typename A> class Numeros{ // Declaración de la plantilla de clase con un tipo
    genérico.
    private:
        A x, y; // Atributos privados del tipo genérico A.
    public:
        Numeros(A a, A b){} // Declaración del constructor público.
        A Mayor(){} // Prototipo de una función que regresa un tipo de dato genérico.
};

#endif


// Archivo .ccp.
#include <iostream>
#include "Numeros.h"
using namespace std;

template<typename A> // Declaración de la plantilla de clase con un tipo genérico.

Numeros::Numeros(A x, A y) : first(a), second(b){} // Constructor público que inicializa
    los atributos.
A Numeros<A>::Mayor(){ // Función que regresa un tipo de dato genérico.
    return (x > y ? x : y);
}
```

El tipo de dato genérico encerrado entre <> corresponde a que, cuando crees un objeto de la clase plantilla, debemos indicarle el tipo de dato que se le va a pasar a los tipos de datos genéricos; es como pasarle un tipo de dato como parámetro a una clase.

```
#include <iostream>
#include "Numeros.h"
using namespace std;

int main(){
    Numeros<int> obj1(10, 100); // Crea objeto Numeros que recibe el tipo int.
    obj1.Mayor(); // Despliega 100.

    Numeros<double> obj2(3.14, 2.56); // Crea objeto Numeros que recibe el tipo double.
    obj2.Mayor(); // Despliega 3.14.
}
```

18.4 Plantillas especializadas

Otro enfoque que se le dan a las plantillas, es que podemos hacer que nuestra clase o función plantilla, reaccione distinto si le pasa un tipo de dato específico, ya que como bien sabemos, las funciones y clases plantilla funcionan para todos los tipos de datos. El siguiente ejemplo corresponde a una clase plantilla que despliega el parámetro que se le pasa al constructor:

```
#include <iostream>
using namespace std;

template <typename A> class MiClase{ // Declaración de la plantilla de clase con un tipo
    genérico.
    public:
        MiClase(A x){ // Constructor público que despliega el parámetro genérico que se le
            pase.
            cout << x << " no es un char.\n";
        }
};
```

Para hacer que una clase plantilla se comporte distinto con un tipo de dato específico, debemos de escribir **template** <> después del bloque de código de la declaración de la clase plantilla, después tenemos que escribir de vuelta todo el bloque de código de la clase, pero con unos cambios:

```
#include <iostream>
using namespace std;

template <typename A> class MiClase{ // Declaración de la plantilla de clase con un tipo
    genérico.
    public:
        MiClase(A x){ // Constructor público que despliega el parámetro genérico que se le
            pase.
            cout << x << " no es un char.\n";
        }
};

template < > class MiClase<char>{
// Declaración de una plantilla especializada de clase, debemos dejar un espacio entre los
     < >.
// Declaración de la clase plantilla, pero con un tipo de dato específico como parámetro.
```

```
    public:
        MiClase(char x){ // Constructor público de la plantilla de clase especializada que
            despliega el parámetro char que se le pase.
            cout << x << " es un char.\n";
        }
}; // Lleva punto y coma al finalizar la declaración.
```

Si queremos que nuestras clases o funciones plantilla se comporten distinto con un tipo de dato, debemos dejar ese espacio entre los <> y en la declaración de la clase, debemos darle como parámetro el tipo que buscamos. Podemos pensar en las plantillas especializadas como una **sobrecarga de plantillas**.

Plantillas especializadas == Sobrecarga de plantillas

Las plantillas genéricas **no heredan** sus miembros, por lo que, cada plantilla especializada debe volver a declarar constructores, funciones, atributos y el destructor. En el main, se crean los objetos de las clases plantilla:

```
#include <iostream>
using namespace std;

int main(){
    MiClase<int> entero(10); // Despliega 10 no es char.
    MiClase<string> cadena("hola"); // Despliega hola no es char.
    MiClase<dobule> flotante(3.1416); // Despliega 3.1416 no es char.
    MiClase<char> caracter('1'); // Despliega 1 es un char.

    return 0;
};
```

19 Excepciones

Una **excepción** es un problema que ocurre durante la ejecución de un programa, es un error que no estaba contemplado en el código pero que el compilador no es capaz de detectar, como dividir entre cero, insertar un valor en la posición 20 de un arreglo cuando este tiene solo 5 posiciones, ingresar una cadena a un tipo entero, etc. Se utilizan las siguientes palabras reservadas:

- **try**: funciona para detectar, en un bloque de código un error que pueda ocurrir durante la ejecución de un programa.
- **catch**: se encarga de detectar las excepciones que puedan ocurrir durante la ejecución, en caso de detectar un rango fuera de un índice de arreglo, lanza un mensaje, en caso de detectar error de formato en una variable, lanza otro mensaje, esto quiere decir que una instrucción *try* puede tener más de un catc*catch*.
- **throw**: funciona para lanzar algún mensaje de error en caso de que una condición o expresión se dé durante la ejecución del programa.

Ahora un ejemplo:

```
#include <iostream>
using namespace std;

int main{
    int EdadMadre, EdadHijo; // Declaración de variables.
    // Instrucción try que detecta excepciones.
    try{
        cin >> EdadMadre >> EdadHijo; // Asignación de valores a las variables.
        if (EdadHijo > EdadMadre){ // Si la edad del hijo es mayor a la de la madre.
            throw 99; // Lanza una excepción en valor entero que recibe. catch
        }
    }
    catch(int x){ // La excepción en valor entero la recibe catch, y la despliega.
        cout << "Imposible que el hijo sea mayor que la madre. Error " << x << "\n"; //
            Despliega Imposible que el hijo sea mayor que la madre. Error 99.
    }

    return 0;
};
```

20 Trabajando con archivos

El lenguaje C++ también da la posibilidad de escribir y leer archivos externos, al igual que C#, debemos utilizar una librería adicional, en este caso, llamada **fstream**.

```
#include <fstream>
```

Esta librería genera tres tipos de datos:

- **ofstream**: permite crear y escribir información en variables.
- **ifstream**: permite la lectura de información de archivos.
- **fstream**: variable general, permite las características de las dos variables anteriores.

Para leer o escribir un archivo, se debe primero abrirlo, si no existe en la ubicación dada, se crea uno automáticamente utilizando la función **open()**, y si se deja de trabajar con este, se utiliza el comando **close()**, como lo muestra el siguiente ejemplo:

```
#include <iostream>
#include <fstream> // Cabecera para escribir y leer archivos.
using namespace std;

int main(){
    ofstream Prueba; // Declaración de variable para crear y escribir un archivo.
    Prueba.open("prueba.txt"); // Se abre el archivo a trabajar.
    Prueba << "Algún texto\n"; // Se escribe en el archivo.
    Prueba.close(); // Se cierra el archivo.

    return 0;
}
```

De igual forma, podemos instanciar el objeto *ofstream* pasándole directamente el nombre y dirección del archivo a abrir:

ofstream Prueba("prueba.txt")

Para corroborar que un archivo está abierto o si se tiene el permiso para acceder a este, podemos utilizar la función **is_open()**, la cual se asegura de que si podamos usar el archivo. Algo más que podemos agregar a esto, es el hecho de que la función *open()* puede recibir parámetros especiales para el manejo del archivo que se va a abrir, si se requiere más de un parámetro, se pueden separar por medio de |, como lo muestra la siguiente tabla:

Table 9: Parámetros especiales de open()

Parámetro	**Definición**
ios::app	Realiza operaciones al final del archivo
ios::ate	Va al final del archivo al abrirlo
ios::binary	Abre el archivo en **modo binario**
ios::in	Abre el archivo para solo lectura
ios::out	Abre el archivo para escritura solamente
ios::trunc	Borra el contenido del archivo si es que existe

```
#include <iostream>
#include <fstream> // Cabecera para escribir y leer archivos.
using namespace std;

int main(){
    ofstream Prueba("Prueba2.txt", ios::out | ios::trunc); // Declaración de variable para
        crear y escribir un archivo con parámetros de solo escritura y truncado.
    if (Prueba.is_open()){ // Verifica si existe el archivo.
        Prueba << "Soy el archivo, estoy abierto y escribes en mi\n"; // Se escribe en el
            archivo.
    }
    else{
        cout << "Algo salió mal\n"; // Mensaje de error.
    }

    return 0;
}
```

Para leer un archivo existente, se utiliza la función **getline** junto al siguiente código:

```
#include <iostream>
#include <fstream> // Cabecera para escribir y leer archivos.
using namespace std;

int main(){
    string linea; // Declaración de variable para ir escribiendo linea a linea el contenido
         del archivo.
    ofstream Prueba; // Declaración de variable para crear y escribir un archivo.
    Prueba.open("prueba3.txt"); // Se abre el archivo a trabajar.
```

```
    while (getline(Prueba, linea)){ // Mientras a linea se le asignen lineas de texto de
        Prueba, el ciclo continuará escribiendo el valor de linea.
        cout << linea << "\n";
    }
    Prueba.close(); // Se cierra el archivo.

    return 0;
}
```