

# Estructuras de datos y algoritmos en C++

Mg. Ing. Patricia Calvo - Esp. Lic. Andrés Juárez



# Índice general

<b>I C++ como herramienta de programación</b>	<b>7</b>
<b>1. Comenzando con C++</b>	<b>9</b>
1.1. Proceso de compilación . . . . .	9
1.2. Integrated Development Environment . . . . .	10
1.3. Nociones básicas del lenguaje C++ . . . . .	11
1.4. Módulos . . . . .	35
<b>2. Memoria dinámica</b>	<b>37</b>
2.1. División de la memoria . . . . .	37
2.2. Punteros . . . . .	39
2.3. Punteros y vectores . . . . .	42
2.4. Pedidos de memoria y liberación . . . . .	44
2.5. Punteros a estructuras complejas . . . . .	45
2.6. Doble punteros . . . . .	45
2.7. Operaciones con punteros . . . . .	46
2.8. Persistencia de datos . . . . .	48
<b>3. Programación Orientada a Objetos</b>	<b>51</b>
3.1. Paradigmas de programación . . . . .	51
3.2. Tipo Abstracto de Datos . . . . .	54
3.3. Diseño de un TDA . . . . .	58
3.4. Clases . . . . .	60
3.5. Objetos . . . . .	63
3.6. Algunas características de la POO . . . . .	68
3.7. Acceso a los atributos . . . . .	71
3.8. Constructores . . . . .	72
3.9. Sobrecarga de métodos . . . . .	72
3.10. Parámetros por defecto . . . . .	73
3.11. El puntero <i>this</i> . . . . .	74
3.12. Atributos y métodos estáticos . . . . .	74
3.13. Destructores . . . . .	77

3.14. Constructor de copia . . . . .	79
3.15. Sobrecarga de operadores . . . . .	82
3.16. Pre y pos condiciones . . . . .	83
<b>II Estructuras de datos lineales</b>	<b>87</b>
<b>4. Genericidad</b>	<b>89</b>
4.1. TDA Vector . . . . .	89
4.2. Punteros void . . . . .	96
4.3. Tipos genéricos . . . . .	99
4.4. Herencia . . . . .	101
4.5. Polimorfismo . . . . .	105
<b>5. Listas</b>	<b>107</b>
5.1. Propiedades y representación . . . . .	107
5.2. Listas propiamente dichas . . . . .	108
5.3. Pilas . . . . .	111
5.4. Colas . . . . .	111
5.5. Implementaciones . . . . .	113
5.6. Listas con templates . . . . .	126
<b>III Estrategias y análisis</b>	<b>127</b>
<b>6. Recursividad</b>	<b>129</b>
6.1. Principios de recursividad . . . . .	129
6.2. Funcionamiento interno . . . . .	131
6.3. Algoritmo divide y vencerás . . . . .	138
6.4. Métodos de ordenamiento usando D y V . . . . .	144
<b>7. Complejidad</b>	<b>149</b>
7.1. Complejidad temporal algorítmica . . . . .	149
7.2. Conteo . . . . .	154
7.3. Medidas asintóticas . . . . .	160
7.4. Análisis de complejidad en algoritmos recursivos . . . . .	166
7.5. Balance entre tiempo y espacio . . . . .	173
<b>IV Estructuras no lineales</b>	<b>177</b>
<b>8. Árboles</b>	<b>179</b>

<b>ÍNDICE GENERAL</b>	<b>5</b>
-----------------------	----------

8.1. Árboles binarios . . . . .	180
8.2. Árboles binarios de búsqueda . . . . .	183
8.3. Recorridos en un ABB . . . . .	194
8.4. Árboles balanceados . . . . .	198
8.5. Árboles multivías . . . . .	208
8.6. Array de bits . . . . .	212
8.7. Trie . . . . .	215
8.8. Cola con prioridad . . . . .	217
8.9. Heap . . . . .	218
<b>9. Grafos</b>	<b>229</b>
9.1. Definición matemática de grafo . . . . .	230
9.2. TDA Grafo . . . . .	232
9.3. Implementaciones . . . . .	234
9.4. Recorridos . . . . .	234
9.5. Caminos en un grafo . . . . .	250
9.6. Árbol de expansión de coste mínimo . . . . .	257
<b>V Temas complementarios</b>	<b>263</b>
<b>10. Hashing</b>	<b>265</b>
10.1. Introducción . . . . .	265
10.2. Funciones de hashing . . . . .	266
10.3. Colisiones . . . . .	269
10.4. Borrado de un elemento . . . . .	273
10.5. Funciones de hash perfectas . . . . .	273
10.6. Funciones de hash para archivos extensibles . . . . .	274
<b>A. Código</b>	<b>275</b>
A.1. Implementación y uso de Vector con templates . . . . .	275
A.2. Implementación y uso pila estática . . . . .	279
A.3. Implementación dinámica y uso de Pila . . . . .	282
A.4. Implementación dinámica y uso de Cola . . . . .	286
A.5. Implementación y uso lista simplemente enlazada . . . . .	289



## **Parte I**

# **C++ como herramienta de programación**



# Capítulo 1

## Comenzando con C++

El lenguaje de programación C++ es un superconjunto del lenguaje C, es decir, incluye todas las características de C y agrega otras nuevas. El lenguaje C está diseñado para escribir programas bajo el paradigma de programación estructurada. Este paradigma enfoca los problemas manteniendo dos conceptos por separado: datos y funciones. Por un lado tenemos los datos y por otro las funciones que los manipulan. C++ evoluciona a partir de C consiguiendo que el lenguaje esté orientado a objetos. La Programación Orientada a Objetos (POO) ya no analiza por separado los datos y funciones, sino que los encapsula o engloba en una entidad que llamamos *clase*. Si bien C++ está desarrollado para construir soluciones con el enfoque POO, al incluir el lenguaje C, también nos permite hacer desarrollos solo de manera estructurada, o una mezcla entre los dos paradigmas que llamamos híbrido.

### 1.1. Proceso de compilación

El lenguaje C++ es un lenguaje compilado, es decir, se necesita de un compilador que entre otras cosas traduzca el código fuente que escribimos a lenguaje máquina y genere un archivo ejecutable. Si bien hay muchos compiladores de C++ recomendamos utilizar el GNU GCC de software libre que corre en sistemas operativos como Windows, Linux y Mac OS X.

Cuando escribimos un código y lo compilamos pasa por tres etapas:

- Preproceso. En esta etapa lo que el compilador realiza es eliminar los comentarios e incluir los archivos indicados con la directiva `#include`.
- Compilación propiamente dicha. En esta fase traduce el código a lenguaje máquina. El compilador nos indicará los errores que encuentre, como variables que no están declaradas, instrucciones mal escritas, falta

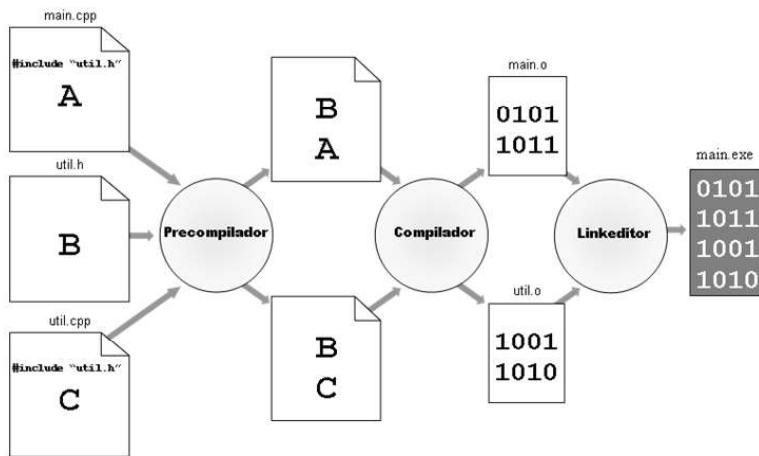


Gráfico 1.1.1: Proceso de compilación

de un punto y coma, etc. Si el código está escrito correctamente genera los archivos objeto que son archivos binarios.

- Enlazado o linkedicion. Finalmente, en esta última etapa el compilador junta (enlaza) todos los archivos traducidos a lenguaje máquina para generar el archivo ejecutable. Si estamos bajo una plataforma de Windows, el archivo tendrá extensión .exe. En Linux no agrega ninguna extensión.

En el gráfico 1.1.1 mostramos un ejemplo de cómo funciona el proceso de compilación para un proyecto con tres archivos: uno de cabecera *util.h* con su correspondiente archivo de definición *util.cpp*, junto con el archivo *main.cpp*.

## 1.2. Integrated Development Environment

En principio, para escribir nuestros programas solo necesitamos un compilador de C++. Por ejemplo, bajamos e instalamos el compilador gratuito GNU GCC que nos sirve tanto para Linux como para Windows mediante MinGW. Luego escribimos nuestro código fuente en cualquier editor de texto y lo guardamos en un archivo de extensión cpp. Finalmente, por línea de comando podemos compilarlo y luego ejecutarlo.

Sin embargo, esto no es recomendable salvo que nuestro código sea muy simple. ¿Por qué? Porque los simples editores de texto no nos marcarían las palabras reservadas con colores, tampoco nos brindarían una lista de los

métodos asociados a cierto objeto, ni nos indicarían palabras mal escritas, entre otras cosas. Se complicaría aún más si necesitáramos trabajar con varios archivos en un mismo proyecto. Tampoco tendríamos ninguna herramienta para seguir nuestro código, en el caso de que haya compilado bien pero tenga algún error a la hora de su ejecución o de tipo lógico, como una salida que no es la esperada. En estos casos estaríamos deseosos de poder seguir el código paso a paso, observando los valores de las distintas variables. Todos estos problemas los soluciona un Integrated Development Environment (IDE) o, en español, Entorno de Desarrollo Integrado. Los IDEs son aplicaciones que nos permiten el desarrollo de aplicaciones.

Un IDE, entre otras cosas viene con:

- Un editor de código que nos indica con distintos colores si la palabra es reservada o no.
- Nos despliega todos los métodos que se pueden llamar desde un objeto determinado.
- Un depurador o debugger para seguir el código paso a paso.

Hay varios IDEs que se pueden utilizar. Nosotros recomendamos utilizar Eclipse bajo plataforma Linux, ya que además podemos agregarle una herramienta para el control del manejo de la memoria dinámica que es Valgrind. Otros dos IDEs que recomendamos son Code::Blocks, ya que es muy sencilla su instalación y manejo, y Dev-C++. Para los usuarios de Mac, el IDE más utilizado es Xcode.

### 1.3. Nociones básicas del lenguaje C++

La mejor forma de aprender un lenguaje es comenzar a utilizarlo con ejemplos sencillos, los que irán incrementando su complejidad a medida que avancemos. Por tradición en todas las enseñanzas de lenguajes de programación el primer programa a realizar es un programa que imprime “Hola mundo” en nuestro monitor.

**Ejemplo 1.1.** Primer programa.

Abrimos el IDE que hemos instalado, generamos un nuevo proyecto y, dependiendo del IDE que usemos, nos genera algunas sentencias de manera automática. De lo contrario, escribimos:

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hola mundo!" << endl;
8     return 0;
9 }
```

Al ejecutarlo observamos una ventana con un cartel que dice “Hola mundo!”.

Si analizamos este código, línea por línea, tenemos lo siguiente:

1. `#include <iostream>`

Esta línea indica que se está incluyendo una librería que se llama *iostream*. La palabra *stream* se utilizará como canal o corriente que se refiere a flujo de datos. En este caso, el flujo serán los caracteres de entrada / salida. Podemos pensarla como un *canal* en donde van ingresando los caracteres a imprimirse. La *io* significa input / output, por lo tanto, esta librería tiene funciones para el manejo de la entrada y la salida.

La entrada estándar será el teclado y la salida estándar la consola o pantalla donde vemos los mensajes.

Luego, las líneas 2 y 4, están en blanco. Toda línea en blanco, el compilador la obviará, al igual que los comentarios. Se utilizan para una mayor claridad del código, dividiendo las distintas secciones de un programa.

3. `using namespace std;`

Con esta instrucción, estamos diciendo que utilice un espacio de nombres llamado *std*. Más adelante veremos mejor qué significa esto, pero se adelanta que, para utilizar la instrucción *cout* y *endl*, debemos indicarle que están definidas dentro del espacio de nombres estándar, es decir, en *std*.

5. `int main()`

Esta instrucción es la cabecera de una función. Toda función se indica de la siguiente forma:

`tipo_de_devolución | nombre_de_la_función | parámetros`

En este caso, el tipo de devolución es un *int*, es decir, un entero (luego se verá la utilidad de esto). El nombre de la función es *main*. Esta es una función especial, que dice que el programa se empezará a ejecutar a partir de esa función y no de otra. Por esta razón todo proyecto debe tener una función *main* para que comience a ejecutarse el código desde ahí.

6. {  
9. }

Las líneas 6 y 9 son llaves que abren y cierran, respectivamente, el cuerpo de un bloque. En este caso, abren y cierran la función *main*.

Representa	Tipo	Tamaño (bytes)
Enteros	char	1
	int	entre 2 y 8
Booleano	bool	1
	float	entre 4 y 8
Flotantes	double	entre 8 y 16
	void	
Vacio		

Tabla 1.3.1: Tipos básicos

```
7. cout << "Hello world!" << endl;
```

Esta línea contiene la primera instrucción propiamente dicha del programa. Indica que imprima lo que se encuentra entre comillas. El *endl* se refiere a un fin de línea, lo que hace que el cursor avance al siguiente renglón o línea.

Lo que va luego de los dos símbolos de menores, se coloca en el *stream* (flujo) de datos. *cout* indica que el flujo que viene detrás debe enviarse a la salida estándar, es decir, la consola. Los distintos flujos se concatenan con los dos símbolos de menores. Aquí vemos que se concatenan los caracteres que forman la frase “Hola mundo!” y, luego, el carácter especial de fin de línea.

```
8. return 0;
```

Finalmente, dijimos que la función debe retornar un entero. Esto se realiza con la instrucción *return*, indicando que se devolverá el entero cero. Podríamos haber devuelto otro entero cualquiera, por ejemplo, uno o menos treinta. Pero, en general, se utiliza una devolución de cero para indicar que todo funcionó bien, sin ningún error. Los valores distintos de cero indican algún error en la ejecución.

Si el lector nunca programó en C o C++ y lo anterior le resulta un tanto extraño no se debe preocupar, ya que a medida que vayamos construyendo otros programas / ejemplos se irá comprendiendo el funcionamiento.

### 1.3.1. Tipos de variables básicas

En el cuadro 1.3.1 se muestran los tipos básicos:

Analizando un poco la tabla anterior nos encontramos que el tipo *char* está dentro de los tipos enteros. Si bien se utiliza para almacenar caracteres, también se puede utilizar para almacenar números no demasiado grandes, como una edad, etcétera. Cada carácter se representa según su código ASCII, en donde, por ejemplo, la A lleva la misma codificación binaria que el número 65, la B es 66, etcétera. Por lo tanto, el dato almacenado, se puede interpretar como un carácter o como un entero.

El tamaño de un char es de un byte, por lo tanto, se pueden almacenar  $2^8 = 256$  símbolos diferentes, ya sean números o caracteres.

El tamaño del resto de los tipos depende de la plataforma, es decir, depende de la arquitectura de la máquina, del sistema operativo y del compilador que se utilice. En general, un *int* ocupará 4 bytes, por lo que puede almacenar  $2^{32}$  números, más de 4 mil millones.

Si bien un dato booleano se podría almacenar en solo un bit, la menor medida posible de almacenamiento es un byte, por lo tanto, una variable de este tipo ocupará un byte. Nos sirve para almacenar dos tipos de valores: *true* o *false*. Tanto C, como C++, representan un valor falso como el entero 0 y, el verdadero, como cualquier valor distinto de cero (en el caso de los booleanos, como el uno). Por lo tanto, si imprimiéramos el valor de una variable booleana, tomándola como un entero (cosa que no tendría demasiado sentido a excepción de tener en cuenta el funcionamiento interno), veríamos que imprime un cero si la variable tiene valor falso y un uno si es verdadero.

Los datos de tipo flotantes almacenan números racionales, pero no todos, ya que entre dos números racionales hay infinitos racionales, por lo que se necesitaría una cantidad de memoria infinita para guardarlos. La diferencia entre dos números racionales se llama “ancho de paso”. Este ancho de paso no es fijo, es decir, no es constante entre todos los números. La idea es buscar una mayor precisión en números chicos, cercanos al cero, que en números grandes dado que interesa minimizar los errores relativos. Por ejemplo, un error de 60cm en una medición puede ser demasiado o insignificante. Si se afirma que algo mide 0,25mts en lugar de 0,85mts la equivocación es importante. En cambio, decir que algo mide 4563,25mts en lugar de 4563,85mts, la diferencia no es significativa. Por lo tanto, el ancho de paso es muy chico para los números cercanos a cero, y se va agrandando a medida que los números son más grandes en valor absoluto. Este ancho de paso puede llegar a representar varios miles, incluso billones de números enteros. Es por este motivo que está el tipo *double* para lograr una mayor precisión, ya sea en números chicos o en grandes. Al tener mayor capacidad, el ancho de paso es menor.

Finalmente, tenemos el tipo *void*, que indica ausencia de valor. Se utiliza en ciertos casos, por ejemplo, cuando una función no necesita devolver nada. Más adelante veremos más sobre este tipo de dato.

### 1.3.2. Identificadores

Los identificadores son palabras que definiremos para utilizarlas como nombres de variables, constantes o etiquetas. Pueden formarse por la combinación de letras, dígitos (números), guión bajo “\_” o el signo \$. Pero no pueden comenzar con números. Por ejemplo *x1*, como nombre de variable es

válido pero no lo es *1x*. Estos nombres no pueden ser palabras reservadas del lenguaje. El lenguaje es *case sensitive*, es decir, sensible a las mayúsculas y minúsculas, por lo que el identificador *NUMERO* será diferente de *Numero* y de *numero*.

Estos identificadores pueden tener la cantidad de caracteres que uno desee pero, nunca se debe perder de vista el buen criterio. Los identificadores demasiado abreviados o demasiado largos pueden oscurecer el código.

### 1.3.3. Variables

Una variable la podemos pensar como un lugar donde guardar datos que pueden variar a lo largo del programa o pueden no estar definidos al principio del mismo. Este lugar estará en la memoria ocupando una o más celdas según sea su tipo. Las variables que vayamos a utilizar debemos declararlas. Si bien podemos declararlas con anterioridad, se recomienda siempre que se pueda declararlas en el mismo momento de su inicialización. ¿Cómo se declara una variable? Simplemente indicando su tipo y luego un nombre. Por ejemplo:

```
int numero;
```

Al finalizar cada sentencia se debe cerrar con un punto y coma.

Las variables viven en el ámbito donde se las declara, es decir, dentro del bloque donde se manifiestan. Fuera de estos bloques no pueden ser accedidas. Por ejemplo, si declaramos una variable dentro de un ciclo while no estará disponible una vez que el ciclo termine.

Las buenas prácticas de programación aconsejan escribir las variables con minúsculas. En caso de necesitar una variable con un nombre compuesto por más de una palabra es aconsejable separarlas con un guión bajo o con una mayúscula al comienzo de las diferentes palabras. Por ejemplo, si necesitáramos almacenar el importe neto de una factura, se recomienda utilizar alguna de estas dos opciones:

- `importeNeto`
- `importe_netto`

La segunda opción es más clara que la primera a la hora de leer el código, sin embargo la primera opción es la más utilizada. Este criterio de diferenciar las palabras se llama *camelCase*.

### 1.3.4. Constantes

Si bien la idea de constante es similar al de variable en el sentido en que también tendrá cierto tipo y ocupará cierto lugar en memoria, la diferencia

es que conocemos desde el principio su valor, el cual no mutará a lo largo del programa. Por ejemplo el valor de un impuesto o una cantidad máxima de clientes, etcétera.

Para definir una constante lo hacemos igual que una variable pero anteponiendo la palabra *const*. Las buenas prácticas aconsejan utilizar letras mayúsculas para diferenciarlas de las variables. En este caso, si tenemos un nombre compuesto no nos queda otra forma de separar las palabras mediante un guión bajo.

Algunos ejemplos:

- `const int IVA = 21;`
- `const float PI = 3.141593;`
- `const int MAX_CLIENTES = 5000;`

### 1.3.5. Comentarios

Los comentarios no son tenidos en cuenta por el compilador pero son importantes para la persona que mira el código. Debemos pensar que cualquier modificación que haya que hacer al código fuente de un programa implica en primer lugar su entendimiento, por lo que los comentarios nos ayudarán a comprender ciertos pasajes dudosos.

Hay dos tipos de comentarios:

- De una línea: con doble barra al principio del comentario.
- ```
// es un comentario de una línea que no debe cerrarse
```
- De más de una línea: con barra asterisco para abrir y asterisco barra para cerrar.

```
/*
    es un comentario
    de varias líneas
    que necesita cerrarse
*/
```

### 1.3.6. Modificadores

Los modificadores son palabras reservadas que se anteponen a los tipos de variables, con el fin de adaptar el tipo según los datos que se almacenarán. Por ejemplo, si sabemos que utilizaremos números enteros, mayores o iguales a cero, podríamos utilizar el modificador *unsigned*. Por lo tanto, definimos

```
unsigned int variable;
```

Es decir, *variable* almacenará números de tipo entero, pero solo positivos. Los modificadores de los enteros son *signed* / *unsigned*. Por defecto, al no indicar nada, los enteros son de tipo *signed*, es decir, con signo, a excepción del tipo *char* que depende de la plataforma.

Además, al tipo *int* se le puede agregar el modificador *short* o *long*, según sea si necesitamos un entero corto o largo. Las longitudes varían según la plataforma. La regla general es la siguiente:

$$\text{size(short int)} \leq \text{size(int)} \leq \text{size(long int)}$$

Es decir, el tamaño de un *short int* será menor o igual que el de un *int*, el que, a su vez, es menor o igual que el de un *long*. En mi plataforma los tamaños son 2, 4, 4, respectivamente. El tipo *double* también puede llevar el modificador *long*, aunque creemos que no tiene demasiado sentido, salvo que se esté trabajando con datos muy específicos.

Para declarar variables de los tipos vistos anteriormente, se debe, simplemente, indicar el tipo con un identificador (el nombre de la variable). A continuación se dan algunos ejemplos:

|                                   |                                                        |
|-----------------------------------|--------------------------------------------------------|
| <code>int x;</code>               | // Variable de tipo entera con signo                   |
| <code>char c;</code>              | // Variable de tipo char, puede ser con signo o sin él |
| <code>unsigned int a;</code>      | // Variable entera sin signo (solo positivos y cero)   |
| <code>unsigned long int b;</code> | // Variable de tipo entero largo sin signo             |
| <code>bool b;</code>              | // Variable booleana                                   |

### 1.3.7. Tipos de datos derivados

Son los tipos de datos que crea el usuario. Un tipo de dato primordial es el de *clase*, pero lo veremos más adelante cuando se vean *Tipos de Datos Abstractos* y Programación Orientada a Objetos. Los otros tipos que veremos a continuación son los vectores, las estructuras y los enumerados.

#### Vectores

Los vectores son arreglos lineales de elementos del mismo tipo. Por ejemplo, si queremos definir un vector de 10 enteros indicamos:

```
int vector[10];
```

Esto nos genera un vector de 10 posiciones, en donde cada posición alberga un entero. La primera posición es la posición cero y, la última, es  $N - 1$ . En este caso, nueve. Entonces, para referirse a la posición *i*, simplemente, hay que indicar el nombre de la variable general, en este ejemplo es *vector* y, entre corchetes, la posición *i*.

```
vector[i]
```

Si necesitáramos una matriz, simplemente se indica con otro corchete la segunda dimensión. Por ejemplo:

```
int matriz [4] [3];
```

declara una matriz de enteros de cuatro filas por tres columnas.

## Estructuras

Una estructura define un tipo compuesto (aunque podría no serlo). Por ejemplo, necesitamos guardar los datos de un empleado. Los mismos consisten en, número de legajo, nombre y sueldo. Entonces armamos una estructura de la siguiente forma:

```
struct Empleado
{
    int legajo;
    string nombre;
    float sueldo;
};
```

De esta forma creamos un nuevo tipo de variable: *Empleado*. Si la queremos utilizar, simplemente indicamos:

```
Empleado empleado;
```



Nota: a diferencia de C, en donde hay que indicar *struct* en la declaración de la variable o utilizar *typedef*, en C++ no es necesaria esta inclusión. Nótese que se utilizó el tipo *string* para el nombre, más adelante se profundizará sobre este tipo.

La variable *empleado* es una estructura (también se puede llamar registro), que tiene tres campos: *legajo*, *nombre* y *sueldo*. Para acceder a cada uno de ellos se debe utilizar el operador “.” Ejemplo:

```
empleado.legajo = 234;
```

Por supuesto, un campo de una estructura puede, a su vez, ser otra estructura. También se podrán crear vectores de estructuras y viceversa: estructuras con vectores, aunque esta última variante no sea recomendada.

## Enumerados

Los tipos enumerados son enteros constantes especiales que se determinan en conjunto. Por ejemplo, si tenemos un *partido\_de\_futbol* que puede tener entre otras cosas un *estado*, y ese estado puede ser *NO\_COMENZADO*, *INICIADO*, *FINALIZADO*, podemos crear un tipo *enum* con dichos estados. El código será:

```
enum Estado {
    NO_COMENZADO;
    INICIADO;
```

| Tipo        | Operación                | Operador | Cantidad de operandos |
|-------------|--------------------------|----------|-----------------------|
| Aritméticos | Suma                     | +        | Binarios              |
|             | Resta                    | -        |                       |
|             | Producto                 | *        |                       |
|             | División                 | /        |                       |
|             | Resto en división entera | %        |                       |
| Asignación  | Asignación simple        | =        | Binario               |
|             | Post incremento          | var++    | Unitario              |
|             | Pre incremento           | ++var    | Unitario              |
|             | Post decremento          | var-     | Unitario              |
|             | Pre decremento           | -var     | Unitario              |
| Comparación | Igualdad                 | ==       | Binarios              |
|             | Distinto                 | !=       |                       |
|             | Mayor                    | >        |                       |
|             | Mayor o igual            | >=       |                       |
|             | Menor                    | <        |                       |
|             | Menor o igual            | <=       |                       |
| Lógicos     | Y (and)                  | &&       | Binario               |
|             | O (or)                   |          | Binario               |
|             | No (not)                 | !        | Unitario              |
| Tamaño      | Tamaño de una variable   | sizeof   | Unitario              |

Tabla 1.3.2: Operadores

```
FINALIZADO;
};
```

Internamente, *NO\_COMENZADO* tendrá el valor 0, *INICIADO* el valor 1, etc. Pero esto no nos debe interesar, el uso que le daremos no es en base a los valores que toma, sino utilizando dichas constantes. Un ejemplo de uso sería:

```
Estado estado;
estado = INICIADO;
...
if (estado == INICIADO)
    cout << "El partido se está jugando";
```

### 1.3.8. Operadores

Los operadores son símbolos que indican al compilador que debe realizar determinadas operaciones. En la tabla 1.3.2 se da un listado de los mismos.

Los operadores, en general, se utilizan combinados. Por ejemplo:

```
a = c + d;
```

Estamos utilizando el operador suma, el cual suma las variables *c* y *d* y, luego, el resultado, lo asigna (operador de asignación) a la variable *a*.

El operador / (división), realiza la división entera si los dos operandos son enteros. En cambio, si uno solo es de tipo flotante o double realiza la operación flotante. Algunos ejemplos:

**Ejemplo 1.2.** División

```
int x = 8, y = 5, z;
z = x / y;
```

*z* queda con valor 1, ya que realiza la división entera

**Ejemplo 1.3.** Más sobre la división

```
int x = 8, y = 5;
double z;
z = x / y;
```

A diferencia de lo que uno podría esperar, *z* queda con valor 1.0, dado que realiza la división entera (tanto *x* como *y* son variables de tipo *int*). Luego, asigna el valor a *z*, por eso le agrega el ,0 para indicar que es un flotante. Sin embargo, la división quedó con un número entero.

**Ejemplo 1.4.** División con decimales

```
int x = 8;
double y = 5, z;
z = x / y;
```

Ahora *z* queda con valor 1.6, ya que al intervenir una variable de tipo *double* en la división, realiza la operación con flotantes.

**Ejemplo 1.5.** El operador % guarda el resto de una división entera.

```
int x = 8 % 5;
```

*x* guarda el valor 3 ya que la división entera entre 8 y 5 es 1, quedando como resto 3.

**Ejemplo 1.6.** Los operadores de incremento (o decremento), son una combinación resumida de operaciones:

```
int x = 3;
x++;
```

*x* queda con el valor 4. *x++* es equivalente a hacer *x* = *x* + 1.

**Ejemplo 1.7.** Operador de post incremento

```
int x = 3, y;
y = x++;
```

*x* queda con el valor 4, pero *y* con el valor 3, por ser de tipo “post” (primero asigna y luego incrementa).

Las sentencias anteriores equivalen a lo siguiente:

```
int x = 3, y;
y = x;
x = x + 1;
```

#### Ejemplo 1.8. Operador de pre incremento

```
int x = 3, y;
y = ++x;
```

En este caso, tanto *x* como *y* quedan con el valor 4, por ser de tipo “pre” (primero incrementa y luego asigna).

Las sentencias anteriores equivalen a lo siguiente:

```
int x = 3, y;
x = x + 1;
y = x;
```

Lo mismo sucede con los operadores de decremento.

Con respecto a los operadores de comparación, se debe tener cuidado ya que la consulta sobre la igualdad de elementos es con un doble igual, dado que uno solo produciría una asignación, en lugar de una comparación.

#### Ejemplo 1.9. Operador *sizeof*. Indica el tamaño en bytes de una variable o de un tipo.

```
cout << sizeof (int) << endl;
```

imprime la cantidad de bytes que ocupa un entero.

### Precedencia de los operadores

Cabe destacar que los operadores tienen diferentes niveles de precedencia, cuanto más alto sea el nivel de precedencia, antes se ejecutará ese operador. Por ejemplo, el operador \* (multiplicación) tiene mayor nivel de precedencia que el + (suma). Por lo tanto:

```
int a = 4, b = 3, c = 2, d;
d = a + b * c;
```

*d* guarda el valor 10, ya que en primer lugar resuelve la multiplicación entre las variables *b* y *c*, dado que la multiplicación se encuentra con mayor prioridad que la suma. Luego, a ese resultado, le practica la suma al valor que contiene *a*. Finalmente, lo asigna a la variable *d*. Es decir, respeta la operación matemática al dividir en términos y realizar la cuenta en forma manual. De todos modos, ante la duda, es aconsejable utilizar paréntesis para asegurarse de obtener los resultados que uno desea. Por ejemplo:

```
int a = 4, b = 3, c = 2, d;
d = a + ( b * c );
```

```
// d guarda el valor 10
int a = 4, b = 3, c = 2, d;
d = ( a + b ) * c;
// d guarda el valor 14
```

Si bien hay más operadores, algunos los veremos más adelante, como los operadores *new* y *delete*, que se utilizan para el manejo de memoria dinámica. Otros se pasarán por alto, por ejemplo los operadores de bits o los de abreviaturas de operaciones, con la finalidad de no cargar al lector con demasiada información que por el momento no es relevante.

#### Ejemplo 1.10. Formas abreviadas de expresiones de asignación

| Operación                         | Tipo               |
|-----------------------------------|--------------------|
| acumulador = acumulador + cuenta; | // Forma expandida |
| acumulador += cuenta;             | // Forma abreviada |

### 1.3.9. Conversiones de tipo

Cuando una variable se asigna a otra siendo de diferente tipo se debe hacer una conversión o casteo de tipo. Este casteo puede ser implícito o explícito. Por ejemplo:

```
char x = 5;
int y = x; // conversión implícita o automática
```

En el código anterior se asigna una variable de tipo char a otra de tipo int. Como el tipo char es un tipo de dato que está incluido en uno de tipo int se realiza una conversión automática o implícita. En cambio, si el código fuera al revés:

```
int x = 5;
char y = x; // da error
```

Aunque sepamos que el valor 5 se puede guardar en una variable de tipo char, da un error porque una variable de tipo int no está incluida en una de tipo char, por lo tanto debemos hacer un casteo explícito de la siguiente forma:

```
char y = (char) x; // correcto
```

En la línea anterior indicamos que tome la variable *x* que es de tipo *int* como si fuera de tipo *char*.

### 1.3.10. Funciones

Las funciones sirven para modularizar el código y hacerlo más legible y sencillo de corregir o modificar. Cada vez que veamos que una porción de código se repite en diferentes lugares, debemos pensar que esa porción debería conformar una función a la cual llamaremos cada vez que la necesitemos.

Las funciones como ya mencionamos constan de una cabecera o firma:

tipo\_de\_devolución | nombre\_de\_la\_función | parámetros

Luego vienen las llaves de apertura y cierre. Dentro estará el código necesario para dicha función.

Por ejemplo, si queremos hacer una función que devuelva el mínimo entre dos números enteros podemos escribir:

```
int minimo (int x, int y) {
    if (x > y)
        return y;
    else
        return x;
}
```

La cabecera consta de:

| tipo_de_retorno | nombre_de_la_funcion | parametro_1 | parametro_2 |
|-----------------|----------------------|-------------|-------------|
| int             | minimo               | int x       | int y       |

Si bien el código anterior es correcto, como la sentencia *return* corta la ejecución de la función, se suele no escribir el *else*:

```
int minimo (int x, int y) {
    if (x > y)
        return y;
    return x;
}
```

El llamado a la función se realiza con el nombre y los parámetros necesarios para su ejecución. Por supuesto, debemos tomar en otra variable o imprimir el resultado que devuelve, de lo contrario dicho valor se pierde. Por ejemplo:

```
int x = minimo (5, 8);
```

Si una función solo debe realizar por ejemplo una impresión por pantalla, sin necesidad de ningún retorno, se coloca la palabra *void* como tipo de devolución, lo que significa devolución vacía, por lo que no hay necesidad de finalizar la función con una sentencia *return*. Por ejemplo:

```
int imprimir (int x) {
    cout << "El valor de la variable es: " << x << endl;
}
```

## Parámetros

Una función puede recibir desde ninguno a la cantidad de parámetros que se necesiten. En el caso de no pasar ningún parámetro se dejan los paréntesis vacíos. En caso de recibir más de uno se listan separados por comas.

Los parámetros pueden ser por referencia o por valor. En el caso de tipos de datos simples, como un entero, un char, un float, etc, los parámetros se

---

**Algoritmo 1.1** Parámetros por valor y por referencia

---

```

void intercambiar(int x, int y) {
    int aux = x;
    x = y;
    y = aux;
}

void intercambiar2(int & x, int & y) {
    int aux = x;
    x = y;
    y = aux;
}

int main() {
    int x = 5, y = 8;
    cout << "Antes del llamado a las funciones" << endl;
    cout << "x = " << x << " y = " << y << endl;
    intercambiar(x, y);
    cout << "Despues del llamado a la funcion intercambiar" << endl;
    cout << "x = " << x << " y = " << y << endl;
    intercambiar2(x, y);
    cout << "Despues del llamado a la funcion intercambiar2" << endl;
    cout << "x = " << x << " y = " << y << endl;
    return 0;
}

```

---

pasan por valor, salvo que se indique lo contrario. Esto quiere decir que la función copia los parámetros en una nueva variable. Por lo tanto, cualquier modificación que se haga a estos parámetros dentro de la función no modifica al parámetro actual, es decir, a la variable original. En cambio, si queremos que estas modificaciones afecten a la variable que se pasa por parámetro debemos pasarla por referencia anteponiendo el símbolo `&`. En el código que figura en 1.1 vemos dos funciones `intercambiar` e `intercambiar2` con el mismo código, la única diferencia es que en la primera los parámetros no se modifican ya que están pasados por valor, en cambio en la segunda están pasados por referencia.

Al ejecutar el código, lo que imprime es lo siguiente:

```

Antes del llamado a las funciones
x = 5 y = 8
Despues del llamado a la funcion intercambiar
x = 5 y = 8
Despues del llamado a la funcion intercambiar2
x = 8 y = 5

```

### Vectores y matrices por parámetro

Los vectores y las matrices también son pasadas por valor, sin embargo, lo que se copia es solo la dirección donde se encuentran los datos por lo que a los efectos prácticos se puede pensar que están pasadas por referencia ya que este tipo de variables conservan las modificaciones cuando se alteran dentro de las funciones. La explicación se entenderá mejor cuando veamos punteros en el siguiente capítulo. Por ejemplo, el siguiente código carga en una función un vector de tamaño  $n$  con los cuadrados de los números desde 1 hasta  $n$ .

```
void cargar(int vec[], int n) {
    for (int i = 0; i < n; i++)
        vec[i] = (i+1)*(i+1);
}
```

Si el código anterior no se comprende totalmente no hay que preocuparse ya que el ciclo *for* lo veremos más adelante. Lo importante a observar son dos cosas:

- El vector no lo estamos pasando por referencia, no tiene el símbolo `&` delante. Sin embargo, el vector queda cargado.
- No es necesario pasarle el tamaño del vector entre corchetes aunque podríamos hacerlo. Si fuera una matriz deberíamos indicarle la segunda dimensión porque internamente lo guarda en celdas contiguas, es decir al igual que un vector.

¿Qué sucede si la variable  $n$  del segundo parámetro no coincide con el tamaño real del vector?

- Si  $n$  es menor que el tamaño del vector estaremos desperdiциando memoria. Es una práctica habitual tener cierto porcentaje de desperdicio de memoria. En estos casos manejamos dos tamaños distintos:
  - Tamaño físico: es el espacio que realmente ocupa el vector en la memoria.
  - Tamaño lógico: es el que utilizamos para los cálculos, impresiones, etc.
- Si  $n$  es mayor que el tamaño real del vector nos sucederá que intentaremos escribir en una zona de la memoria que no hemos reservado, cosa que en general nos dará un error en tiempo de ejecución.

En el gráfico 1.3.1 vemos la representación de una matriz de 3 filas por 5 columnas. En la parte *a*) el gráfico muestra cómo lo interpretamos, y en la

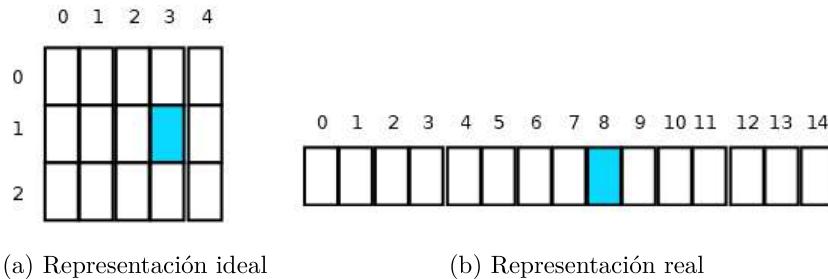


Gráfico 1.3.1: Representación de las matrices

parte *b*) cómo realmente la almacena. Tenemos que tener en cuenta que el almacenamiento es en la memoria que es una tira de *bytes*. Por lo tanto, si quisiéramos dirigirnos a la celda que está marcada en celeste deberíamos escribir el nombre de la variable, supongamos que se llama *matriz*, y en un corchete el número de fila y en un segundo el de la columna, de la siguiente manera:

```
matriz[1][3]
```

Sin embargo, el compilador debe ir a la celda número 8 que vemos en la figura *b*). Para hacer este cálculo necesita multiplicar el número de fila por la cantidad de columnas y sumar el número de columna:

$$\text{numeroFila} * \text{columnas} + \text{numeroColumna} = 1 * 5 + 3 = 8$$

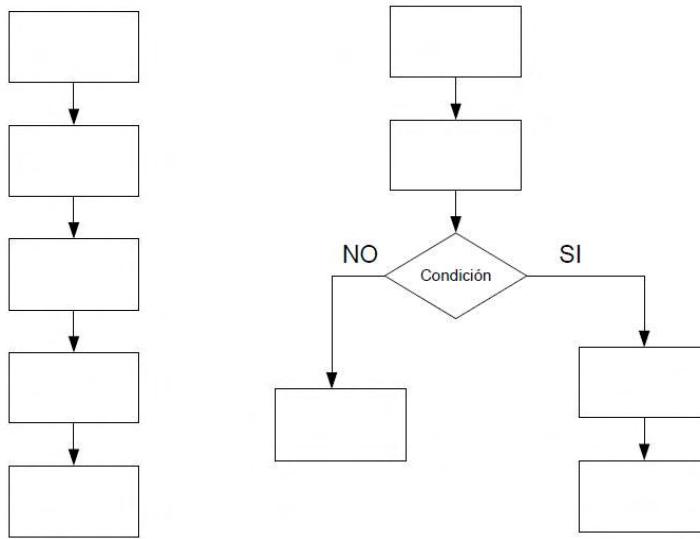
Es por este motivo que el compilador necesita conocer la segunda dimensión de la matriz. Si fuera una matriz *n-dimensional* donde *n* puede ser 3, 4, etc, sucede lo mismo, solo la primera dimensión puede quedar sin definir.

### 1.3.11. Control de flujo

Para poder realizar programas útiles debemos poder controlar el flujo de las sentencias. Es decir, algunas sentencias se ejecutarán solo si se cumple determinada condición, aquí aparecen las estructuras condicionales. En otras situaciones, podríamos desear que se ejecutaran *n* veces algunas sentencias, siendo *n* algún número muy grande o un número variable. Por este motivo necesitamos de estructuras repetitivas. Veamos las distintas estructuras.

#### Sentencias secuenciales

Las sentencias secuenciales son las que hemos visto hasta el momento: sentencias de asignación, de impresión, etc. Al ejecutarse una detrás de la



(a) Estructura secuencial

(b) Estructura condicional

Gráfico 1.3.2: Diagramas de flujo

Otra decimos que estamos en una estructura de tipo secuencial. Los bloques se delimitan con llaves:

```
{ // Se abre un bloque
// Sentencias del bloque
} // Se cierra el bloque
```

### Sentencias condicionales o selectivas

Las estructuras condicionales hacen que el flujo de ejecución se ramifique. O, pensado de otra manera, si la ejecución de sentencias es siempre secuencial (una detrás de la otra), con una estructura condicional podremos decidir que una sentencia o más se ejecuten solo si se cumple determinada condición.

En el gráfico 1.3.2 vemos ejemplos de ambas estructuras: secuenciales y condicionales. Por supuesto que las estructuras condicionales están, a su vez, formadas por estructuras secuenciales.

**if – else** Funciona al igual que en C. La sintaxis es la siguiente:

```
if (condición) sentencia1;
else sentencia2;
```

El *if* verifica que la condición se cumpla, en ese caso ejecuta la sentencia1. El *else* es optativo ya que podríamos desechar no ejecutar nada si la condición

no se cumple. Por otro lado, si necesitáramos ejecutar más de una sentencia deberíamos abrir bloques con llaves. De todas formas podría ser una buena práctica abrirlos siempre, aunque fuera solo una sentencia por una cuestión de claridad en el código. Veamos algunos ejemplos en los algoritmos que figuran en 1.2.

En el ejemplo 3 hay dos sentencias dentro del *if*, por este motivo está la necesidad de usar las llaves de apertura y cierre. El ejemplo 4 utiliza más de una condición para entrar al *if*, como están concatenadas con un *and*, ambas deben ser verdaderas para que ingrese en ese bloque. Es conveniente, cuando se anidan sentencias, dejar sangrías para ubicar rápidamente a qué bloque corresponde cada sentencia. En el ejemplo 5 vemos el uso de condiciones anidadas, es decir, una dentro de otra.

**switch** Cuando se necesitan anidar varios *if* y las condiciones son evaluaciones de expresiones que devuelven un entero, se puede y se recomienda utilizar *switch*. La sintaxis es:

```
switch (expresion) {
    case valor1: sentencia_1;
    case valor2: sentencia_2;
    ...
    default: sentencia_n; // El default es optativo
}
```

Por ejemplo, si queremos tomar un número del uno al cinco y escribirlo con letras podríamos hacer lo siguiente:

```
// Ejemplo sin switch.
if (x == 1) cout << "uno";
else if (x == 2) cout << "dos";
else if (x == 3) cout << "tres";
else if (x == 4) cout << "cuatro";
else if (x == 5) cout << "cinco";
else cout << "Opción no válida";

// Ejemplo utilizando switch.
switch (x) {
    case 1: cout << "uno";
    case 2: cout << "dos";
    case 3: cout << "tres";
    case 4: cout << "cuatro";
    case 5: cout << "cinco";
    default: cout << "Opción no válida";
}
```

---

**Algoritmo 1.2 Ejemplos if - else**

---

```
// Ejemplo1.  
if (x > 5)  
    y = x + 2;  
else  
    y = x * x;  
  
// Ejemplo2.  
if (x > 5)  
    y = x + 2;  
  
// Ejemplo3.  
if (x > 5) {  
    y = x + 2;  
    y = x * x;  
}  
else {  
    y = x + 5;  
}  
  
// Ejemplo4.  
if ((x > 5) && (y < 3)) {  
    y = x + 2;  
    x = x * x;  
}  
else {  
    y = x + 5;  
}  
  
// Ejemplo5.  
if (x > 5) {  
    if (y < 4) // x es mayor a 5 e y menor a 4  
        y = x * x;  
    else // x es mayor a 5 e y no es menor a 4  
        y = x - 2;  
}  
else // x no es mayor a 5, el valor de y no interesa {  
    y = x + 5;  
}
```

---

Sin embargo, si por ejemplo *x* valiera 3, la porción del código anterior arrojaría lo siguiente:

```
tres
cuatro
cinco
Opción no válida
```

¿Por qué? Porque el case determina el punto de entrada pero no de salida. Por lo que deberíamos modificar nuestro programa de la siguiente forma:

```
// Ejemplo utilizando switch (corregido).
switch (x) {
    case 1: cout << "uno"; break;
    case 2: cout << "dos"; break;
    case 3: cout << "tres"; break;
    case 4: cout << "cuatro"; break;
    case 5: cout << "cinco"; break;
    default: cout << "Opción no válida";
}
```

La sentencia *break* que se detallará más adelante, hace una ruptura (pega un salto) hacia fuera del bloque.

En ciertos casos está la necesidad de ejecutar cierta sentencia con varios valores de una variable y no con uno solo. Por ejemplo, si se desea imprimir “malo”, “regular”, “bueno”, “distinguido”, “sobresaliente”, según cierta nota, podríamos escribir:

```
switch (nota) {
    case 1:
    case 2:
    case 3: cout << "malo"; break;
    case 4:
    case 5: cout << "regular"; break;
    case 6:
    case 7: cout << "bueno"; break;
    case 8:
    case 9: cout << "distinguido"; break;
    case 10: cout << "sobresaliente"; break;
    default: cout << "nota inválida";
}
```

En el ejemplo anterior imprime “malo” en el caso de que la nota fuera 1, 2 o 3. Regular con 4 y 5, etc.

El switch se puede utilizar también con variables de tipo *char*, en ese caso, los valores hay que encerrarlos entre comillas simples:

```
switch (letra) {
    case 'A':
    case 'E':
    case 'I':
```

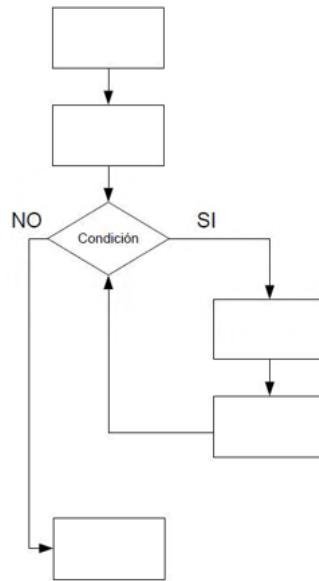


Gráfico 1.3.3: Diagrama de estructura repetitiva

```

case '0':
case 'U': cout << "vocal"; break;
default: cout << "consonante";
}
  
```

### Sentencias iterativas o repetitivas

Muchas veces necesitaremos ejecutar una sentencia de manera repetida. Por ejemplo, imprimir un registro hasta que no haya más registros en el archivo o listar todos los números primos desde 2 hasta cierto valor N. Para resolver estos problemas debemos utilizar estructuras de tipo repetitivo o, también llamado, iterativo. En el gráfico 1.3.3 se ve el diagrama de flujo de una estructura repetitiva.

**while** Es idéntico al uso del while en C. La estructura es:

```

while (condicion)
    sentencia
  
```

Es decir, mientras la condición sea verdadera se ejecuta la o las sentencias que hubieran en el bloque. Por ejemplo:

```

while (i < 10) {
    i++;
  
```

```

        cout << i << endl;
}

```

**do – while** Funciona de la misma manera que el while, solo que la condición se verifica al final. Ejemplo:

```

do {
    cout << x << endl;
    x--;
} while (x > 0);

```

La diferencia con el while es que el bloque se ejecuta por lo menos una vez. Por ejemplo, si x valiera -3 el bloque se ejecutaría igual porque la verificación de la condición es al final.

**for** Esta sentencia tiene múltiples usos, al igual que en C. Los códigos podrían llegar a ser poco legibles dependiendo de la forma en que lo utilicemos. La estructura es la siguiente:

```

for (sentencia_inicio; condicion; sentencias)
    otras_sentencias

```

La sentencia\_inicio se ejecuta solo una vez al principio, luego verifica la condición, en caso de que la condición sea verdadera ejecuta las otras\_sentencias y, luego, ejecuta las sentencias que figuran como último parámetro. Tanto las sentencias de inicio como las del tercer parámetro podrían ser varias, en este caso se deberían separar mediante una coma. Pero, también, podrían estar vacías al igual que la condición (una condición vacía genera un bucle infinito, a menos que haya alguna ruptura con *break* o *return*). Con ejemplos varios se entenderá mejor.

```

// Imprimir los números del 1 al 9
// Ejemplo1.
for (i = 1; i < 10; i++)
    cout << i << endl;

// Ejemplo2.
// Se ejecutan dos sentencias en el último parámetro
// El cuerpo del for es vacío
for (i = 1; i < 10; cout << i << endl, i++);

// Ejemplo3.
// El tercer parámetro está vacío
for (i = 1; i < 10; ) {
    cout << i << endl;
    i++;
}

```

```
// Ejemplo4.
// El primer parámetro está vacío
i = 1;
for ( ; i < 10; i++ ) {
    cout << i << endl;
}
```

Por supuesto que, si bien los ejemplos 2, 3 y 4 funcionan correctamente, no son recomendables, ya que el ejemplo 1 es más claro.

### Sentencias de ruptura

Son sentencias que cortan el hilo normal de ejecución, saliendo de los diferentes bloques.

**break** Sirve para salir de un bloque cualquiera. Por ejemplo:

```
for ( int i = 0 ; i < 10; i++ ) {
    cout << i << endl;
    if ( i == 4 ) break;
}
```

El código anterior empezará listando los números 0, 1, 2, 3 y, luego de listar el número 4 saldrá del bloque y continuará con la sentencia que haya fuera de él.



Nota: generalmente se lo utiliza para salir de un bloque *switch*.  
No se recomienda utilizar esta sentencia en otros casos.

**continue** La sentencia *continue* se utiliza dentro de un bucle *for*, *while* o *do-while* y sirve para transferir el control al final del cuerpo del bucle. Por ejemplo:

```
for ( int i = 0 ; i < 10; i++ ) {
    if ( i == 4 ) continue;
    cout << i << endl;
}
```

El código anterior no imprime el valor 4, ya que transfiere el control al final del bucle y continúa con el siguiente valor. Aconsejamos no utilizarlo.

**return** El *return* realiza varias cosas:

- Finaliza la ejecución de una función.
- Devuelve el control a la función que la invocó.

- Como vimos en la sección de funciones, si la función tiene algún tipo de retorno, el *return* debe ir acompañado por alguna expresión (valor, nombre de una variable o alguna expresión más compleja) que sea de un tipo compatible al tipo de retorno.

### 1.3.12. Espacio de nombres

En los grandes proyectos, realizados por varios desarrolladores, es común que el nombre de una variable o función se repita. Por ejemplo, la sección de RRHH de una empresa tendrá la necesidad de manejar una tabla con los datos de sus empleados en donde tendrán sus direcciones, fechas de nacimientos, etc. El nombre natural para asignarle a la tabla será *empleados*. Pero en la oficina de Sueldos también tendrán que manejar otros datos de estos empleados, como horas trabajadas, sueldo básico, etc. Es lógico que también llamen *empleados* a esa tabla. Si estos módulos están al mismo nivel, nos dará un error indicando que esa variable ya existe.

El lenguaje C no tiene forma de solucionar esto más que cambiando los nombres de las variables. C++ nos proporciona los espacios de nombres (*namespace*) para resolver este problema. Dentro de un espacio de nombre las variables no pueden repetirse pero sí lo pueden hacer en distintos espacios de nombres. En el ejemplo anterior tendríamos dos espacios de nombres: uno para RRHH y otro para Sueldos. De esta manera, cuando uno se refiera a la variable *empleados* deberá indicar si es la variable que corresponde a RRHH o a Sueldos.

Esto nos sucede con el objeto *cout* que corresponde al espacio de nombres *std* (estándar). Por eso debíamos incluir la sentencia

```
using namespace std;
```

De lo contrario hay que indicar *std::* antes de cada *cout* y *endl*.

¿Cómo creamos un espacio de nombres? Simplemente escribiendo *namespace nombre\_del\_espacio* y entre llaves irán las declaraciones deseadas. Por ejemplo:

```
namespace Uno {
    int x;
    ...
}

namespace Dos {
    int x;
    ...
}

// Uso de los espacios de nombres
Uno::x = 5; // Se utiliza la variable x del espacio Uno
```

```
Dos::x = 8; // Se utiliza la variable x del espacio Dos
```

Cuando veamos objetos, cada clase representará un espacio de nombres de manera implícita.

## 1.4. Módulos

Cuando los proyectos empiezan a crecer y dejan de ser algo que se resuelve en no más de 10 líneas de código es conveniente y una necesidad modularizar. La modularización en primer lugar consiste en definir algunas funciones que resuelvan las distintas tareas, las cuales serán llamadas desde la función principal (`main`).

### 1.4.1. Declaración y definición

Debemos distinguir entre declaración y definición de una función.

- Declaración. Consta de la cabecera o firma de la función, que es la primera línea con el tipo de devolución, el nombre de la función y el tipo de sus parámetros. Si la definición se hace aparte, hay que cerrarla con un punto y coma. Por ejemplo

```
int sumar (int, int);
```

Nótese que no hace falta poner el nombre de los parámetros, aunque en general se recomienda hacerlo ya que esclarece el código.

- Definición. Consiste en el código completo de la función, incluyendo la cabecera.

Para que la función `main` pueda llamar a las demás funciones deben estar declaradas antes. Si bien pueden también definirse alcanza solo con la declaración.

### 1.4.2. División de un proyecto en módulos

A medida que los proyectos se van haciendo más complejos se empiezan a acumular muchas declaraciones antes del `main` en conjunto con las definiciones luego del `main`. El crecimiento no es solo en cantidad de funciones sino en niveles. Es decir, el proyecto deja de ser un `main` que llama a 4 o 5 funciones simples, porque estas a su vez llamarán a otras, y esto seguirá expandiéndose según la necesidad del proyecto. Por otra parte, las funciones empiezan a relacionarse según determinadas características, por ejemplo, en un proyecto que toma datos, hace cálculos y luego en base a estos cálculos

emite reportes, tendremos funciones encargadas de tomar los datos de entrada y darles el formato necesario. Otras se encargarán de hacer cálculos, finalmente tendremos el grupo de funciones que harán los reportes. En este esquema es conveniente dividir el proyecto en tres módulos además del principal.

En general, cada módulo constará de

- Un archivo de cabecera o archivo .h (header). En este archivo se pondrán todas las cabeceras de las funciones, y si existiera la definición de alguna constante o tipo de datos, pero no debería tener ninguna definición.
- Un archivo de implementación o definición de las funciones declaradas en el archivo .h. Este archivo será de extensión .cpp y deberá incluir el archivo de cabecera con la directiva include y el nombre del archivo entre comillas. Por ejemplo:

```
#include "reportes.h"
```

Los nombres de estos archivos podrían ser cualquiera pero se recomienda que el archivo de implementación se llame igual que el de cabecera, obviamente cambiando su extensión de .h a .cpp. Algo muy importante a tener en cuenta es que las inclusiones son solo de los archivos .h, los archivos .cpp no se deben incluir.

## Ejercicios

### Ejercicio 1.1.