

Fundamentos de la programación

Fernando Diaz Urbano ¹

24 de septiembre de 2015

¹Agradecimientos a Dr. Juan Falgueras por el apoyo y la ayuda en la realización de este documento.

Índice general

1. Introducción	5
2. Historia y conceptos básicos	7
2.1. El compilador y el IDE. Fuente de muchos problemas.	7
2.1.1. Nuestro compilador en la nube	8
2.2. Nuestro primer programa, ¡Hola mundo!	10
2.2.1. using namespace std	13
2.2.2. Comentarios	13
2.3. Tipos de variables	14
2.3.1. Constantes	15
2.3.2. Operadores	15
2.3.3. Conversión de tipos ó <i>casting</i>	16
2.3.4. typedef	16
2.3.5. enum	17
2.4. Entrada de datos	17
3. Subprogramas	19
3.0.1. Prototipos	21
3.0.2. Paso de parámetros por referencia	21
3.0.3. Sobrecarga	22
4. Estructuras de control	25
4.1. Expresiones <i>booleanas</i>	25
4.2. Estructuras interativas	25
4.2.1. Bucle <i>for</i>	26
4.2.2. Bucle <i>while</i>	26
4.2.3. <i>do-while</i>	27
4.3. Estructuras de control	27
4.3.1. if-else	27
4.3.2. Bloques <i>if</i> anidados	28
4.3.3. <i>switch</i>	29

5. Arrays	31
5.1. Conceptos básicos	31
5.1.1. Cómo declarar <i>arrays</i> en C++	31
5.1.2. <i>Arrays</i> multidimensionales	32
5.1.3. Arrays en subprogramas	32
5.2. Cadenas de caracteres	33
5.2.1. <i>Strings</i>	34
6. Tratamientos secuenciales	37
6.1. Algoritmos de ordenación	37
6.1.1. <i>Bubble sort</i>	37
6.1.2. Select sort	38
6.1.3. Insert sort	40
6.2. Búsqueda de caracteres	41
7. Estructuras	43
7.1. Manipulando estructuras	43
7.1.1. Array de estructuras	44
8. Ficheros	47
8.1. ¿Qué es un fichero?	47
8.1.1. ¿Legible o no legible?	47
8.2. Tipos de ficheros	48
8.3. Manipulación de ficheros	48
9. Anexos	51
9.1. El compilador y el IDE. Fuente de muchos problemas	51
9.1.1. Instalación del compilador	52
9.1.2. Instalación del compilador en Linux.	61
9.2. Nuestro primer programa, ¡Hola mundo!	62
9.2.1. Comentarios	66
10. Cuestionarios	67
10.1. Lo básico	67
10.2. Estructuras de control	68
10.3. Estructuras iterativas	69
10.4. Funciones	70
10.5. Switch-case	71
10.6. Arrays	72
10.7. Estructuras	73
10.8. Punteros	74

Capítulo 1

Introducción

El objetivo de esta obra, es enseñar los conceptos básicos de la programación. Para ello, haremos uso del lenguaje C++, sin llegar a usar la orientación a objetos propiamente dicha, es por esto que se le puede considerar un C-; a lo largo del texto. Lo que trataremos, será también de utilidad para la asignatura "Fundamentos de la programación", que se cursa en los distintos grados de la Universidad.

Además, procuraré que en mayor medida se resuelvan cuestiones que compañeros míos me han preguntado a lo largo del curso, y que posiblemente tengan muchos estudiantes que cursen esta asignatura. Se presentarán los conceptos básicos de la programación, desde los distintos tipos de variables, pasando por estructuras de control, iterativas, *arrays* e incluso manipulación de ficheros. Incluiré además, un capítulo dedicado al uso de punteros, ya que suele dar muchos dolores de cabeza al principio.

Todo vendrá explicado con la mayor claridad y cercanía al lector posible, ya que se pretende enseñar desde *cero*. Sin más entretenimiento, damos paso al primer capítulo.

En caso de tener alguna duda, dejo disponible mi dirección de correo electrónico para responder las dudas que os encontreis o problemas, especialmente con la parte del compilador: ***fernandodiaz@uma.es***

¡Aprendamos!

Capítulo 2

Historia y conceptos básicos

Lo primero que haremos, será introducir el lenguaje de programación que utilizaremos a lo largo de este libro, y qué menos que conocer su historia. Si no estas interesado, puedes saltarte esta parte sin problemas.

La historia de *C++* se remonta a 1979, cuando *Bjarne Stroustrup* estaba haciendo su tesis doctoral. Tuvo la oportunidad de trabajar con el lenguaje *Simula*, diseñado para simulaciones como su nombre indica. Este lenguaje, introdujo el concepto de la programación orientada a objetos; sin embargo apenas tenía uso práctico este lenguaje. *Bjarne*, encontró este paradigma de programación muy útil para el desarrollo de software, por lo que pronto comenzó a trabajar en clases de *C*, cuyo objetivo residía en añadir la programación orientada a objetos a este lenguaje. Su lenguaje incluyó clases, herencia, *inlining*, argumentos de función por defecto y una fuerte comprobación de tipos al añadir todo este tipo de características al lenguaje *C*.

En 1983, pasó de llamarse *C* a *C++*, y este *++* es el operador utilizado para incrementar; de ahí que sea considerado una extensión.

Con el tiempo, se fueron añadiendo características, tales como las funciones virtuales, sobrecarga de funciones, referencias, constantes, comentarios en una sola línea y la herencia múltiple; hasta llegar al lenguaje que hoy en día tenemos y sigue actualizándose a día de hoy.

La última revisión del lenguaje, en la que este libro se base es en *C++11*, y se espera una revisión futura llamada *C++17*, cuya fecha de publicación es aún desconocida.

2.1. El compilador y el IDE. Fuente de muchos problemas.

Un problema muy frecuente a la hora de comenzar a estudiar la asignatura, es que en la mayoría de ocasiones no se es capaz de instalar con facilidad el compilador, ya sea por desconocimiento, o porque el propio sistema da errores.

Con el fin de eliminar de manera *sencilla y gratuita* todos estos problemas, haremos uso de una nube.

NOTA: En los anexos, encontraremos la forma de instalar el compilador sin hacer uso de la nube. Sin embargo, el método expuesto a continuación es mucho más sencillo y debería ofrecer menos problemas.

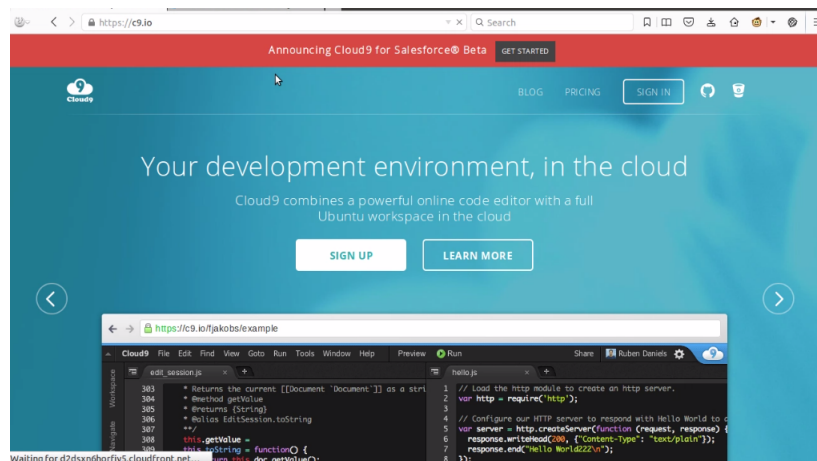
2.1.1. Nuestro compilador en la nube

Cloud9 es una *maquina virtual online* donde podremos desempeñar todas las tareas requeridas en la asignatura.

Anexo al documento, se encuentra un archivo en formato **.webm** donde se indican los pasos para registrar nuestra cuenta y crear nuestra máquina virtual, así como crear nuestro primer programa.

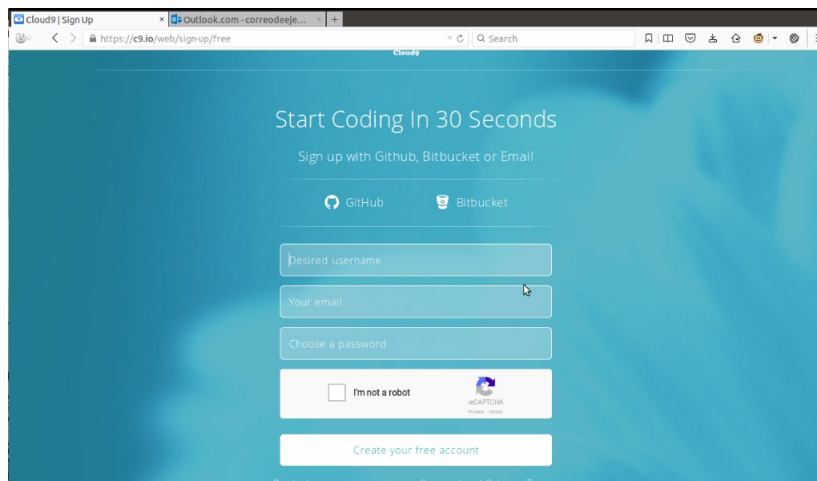
En caso de no poderse visualizar el documento en vídeo, se especificarán aquí los pasos (*extraídos del video*) sobre cómo realizar el proceso.

En primer lugar, debemos entrar en <https://c9.io> donde encontraremos una pantalla similar a la siguiente:



En ella, pulsamos en donde pone **Sign Up** y nos mostrará lo siguiente:

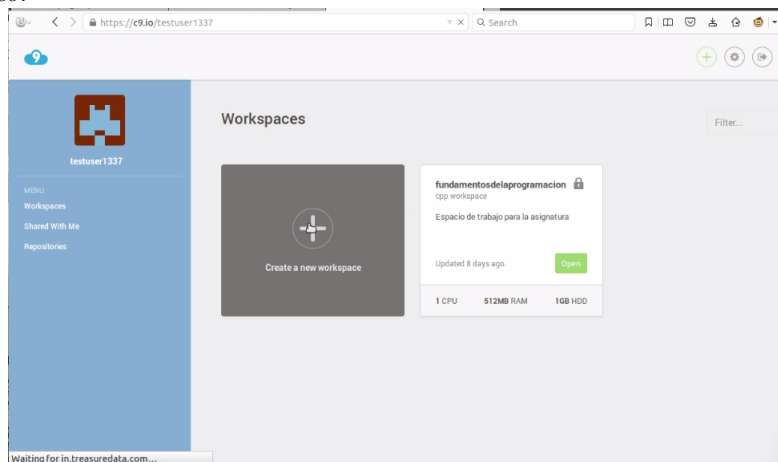
2.1. EL COMPILADOR Y EL IDE. FUENTE DE MUCHOS PROBLEMAS.9



En ella, nos pedirá nuestro nombre de usuario, nuestro correo electrónico y nuestra contraseña. También tendrá un **captcha** que os hará una pregunta para confirmar que sois humanos.

Una vez hecho esto, a vuestro correo llegará un e-mail de confirmación de cuenta. En este correo, os aparecerá un enlace que debéis pulsar para activar vuestra cuenta (viene indicado en el correo) y una vez dentro se habrá activado vuestra cuenta.

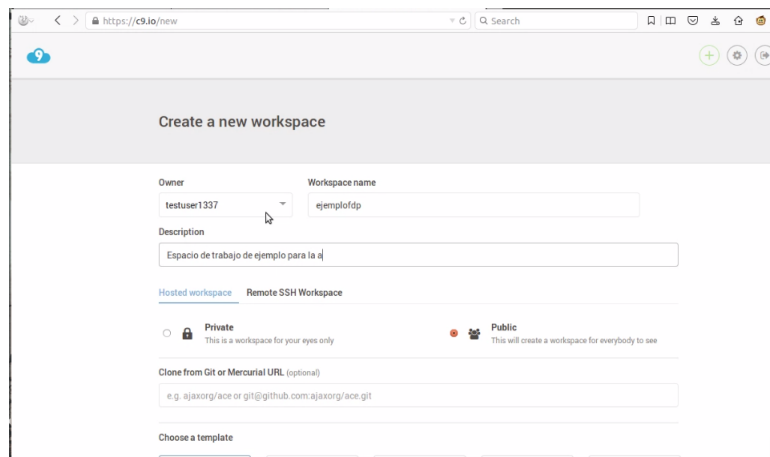
Hecho este proceso, pasamos a entrar en nuestra cuenta y veremos que tenemos la posibilidad de crear un nuevo *workspace* o espacio de trabajo. Yo etnía uno de antemano, pero crearemos uno nuevo pulsado en *Create a new Workspace*:



Nos pedirá una serie de datos, que hemos de rellenar, entre ellos:

- Workspace name: Elegimos el nombre que queremos ponerle al espacio.
- Description: Ponemos una breve descripción para guiarnos sobre su finalidad.

- Hosted workspace: Aquí escogeremos entre que nuestro espacio de trabajo sea público o privado, en nuestro caso escogeremos que sea **privado**. **Advertencia:** Sólomente se puede escoger un espacio privado.
- Choose a template: Escogeremos **C++** para tener el entorno preparado.

The image shows a web browser window at the URL https://c9.io/new. The page has a header with a C9.io logo and navigation icons. The main content area is titled 'Create a new workspace'. It contains several form fields: 'Owner' with a dropdown menu showing 'testuser1337', 'Workspace name' with a text input containing 'ejemplodp', and 'Description' with a text input containing 'Espacio de trabajo de ejemplo para la a'. Below these fields are two tabs: 'Hosted workspace' (selected) and 'Remote SSH Workspace'. Under the 'Hosted workspace' tab, there are two radio buttons: 'Private' (selected) and 'Public'. Below the radio buttons is a text input for 'Clone from Git or Mercurial URL (optional)' with the example 'e.g. ajaxorg/ace or git@github.com:ajaxorg/ace.git'. At the bottom, there is a 'Choose a template' section with several buttons, one of which is highlighted with a red border.

Y finalmente pulsamos en *Create workspace*. Una vez hecho esto, se nos abrirá nuestro espacio de trabajo donde podremos proceder a crear nuestro primer programa.

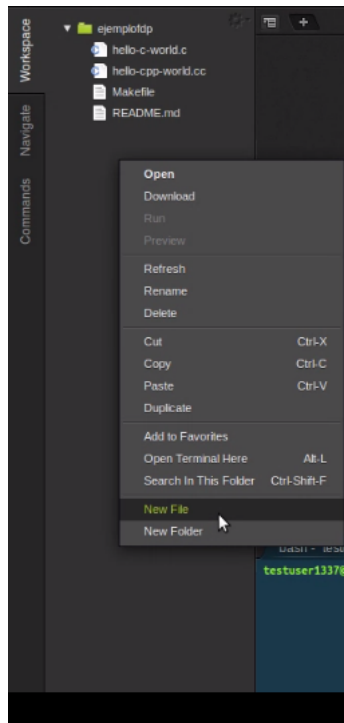
2.2. Nuestro primer programa, ¡Hola mundo!

Bien, ya tenemos instalado nuestro entorno de programación. Ahora, lo que nos queda es realizar nuestro primer programa.

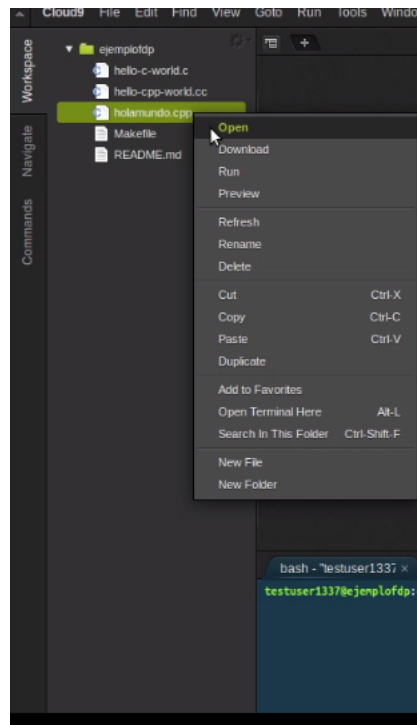
Para ello, anotar que necesitaremos crear nuestro archivo **.cpp** donde escribir nuestro código.

Por ello, pulsamos *click derecho* y pulsamos en **New File**. Nos pedirá un nombre, por lo que escribimos como ejemplo **holamundo.cpp**.

NOTA: Este nombre, '**holamundo.cpp**' puede ser cualquier cosa siempre que mantenga el '**.cpp**' al final. Ejemplos: `archivo.cpp`, `programa.cpp`, `hola.cpp`, `programa1.cpp` ...



Una vez hecho esto, damos dos clicks archivos o pulsamos click derecho y pulsamos **Open**.



Ahora, nos abrirá una pestaña donde podremos editar nuestro código. Escribiremos nuestro primer programa a continuación:

```
#include <iostream>

using namespace std;

int main(){

    cout << "Hola mundo!" << endl;
    return 0
}
```

- **#include** permite incluir una *librería*. **Librería:** Archivo que contiene un grupo de funciones predefinidas, que nos permitirá utilizar en nuestro programa sin saber si quiera cómo están hechas. Existen diferentes librerías, utilizaremos las básicas como *iostream* (Entrada y salida), *fstream* (Control de archivos), *cmath* (Funciones matemáticas)... Hay miles de librerías para casi cualquier cosa, lo que facilitará mucho el trabajo en proyectos grandes.
- **cout >>:** Imprime un mensaje en pantalla, que esté entre comillas.
- **endl:** Introduce una nueva línea, es necesario usarlo con *cout* y previo a él debe llevar el separador <<.

- **return 0:** Indica que todo ha acabado correctamente, esto es más a nivel interno.

2.2.1. using namespace std

```
#include <iostream>

int main() {
    std::cout << "Hola_mundo!";
    return 0;
}
```

Supongo que ahora os estaréis preguntando, ¿qué narices es eso de *std::*? Bien, como se explicó anteriormente, una librería es un conjunto de muchas funciones, entonces... ¿Qué pasaría si dos librerías diferentes tuvieran dos funciones que se llaman de la misma manera? ¿Cómo le decimos cual escoger? Para ello existe el *espacio de nombres*, pongamos un ejemplo práctico.

```
#include <archivos>
#include <libros>

int main()
{
    libros::leer();
    archivo::leer();
}
```

Como ves, tenemos dos funciones a leer, y de esta manera le indicamos al compilador cual coger. Esto ahora mismo no es importante, pero es conveniente conocerlo. A partir de ahora, nos ahorraremos escribir siempre *std::* utilizando justo después de incluir las librerías, la coletilla de todos nuestros programas: *using namespace std*; Tras escribir esta línea, podremos omitir *siempre* escribir *std::*, aunque hemos de tener siempre en cuenta las consecuencias de esto. Por ello, una vez aplicado esto nuestro programa quedara de la siguiente manera:

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

int main() {
    cout << "Hola_mundo!";
    return 0;
}
```

2.2.2. Comentarios

Vamos a ver de forma breve los comentarios. Mientras estamos haciendo nuestro programa, conforme se va haciendo más grande es más complicado luego

entender qué se esta haciendo, es fácil perderse. Para ello, C++ permite el uso de comentarios. Siempre que se haga un comentario, esa parte de código será ignorada por el compilador, por lo que realmente *no* formara parte del programa. Los comentarios se pueden poner en una única línea, precedida por `//` o de diversas líneas `/* */`. Pongamos un ejemplo:

```
// Esto es un comentario de una sola línea
/* Esto es un comentario
de varias líneas
y el compilador no lo vera nunca
*/
```

2.3. Tipos de variables

Bueno, por ahora no sabemos mas que imprimir un mensaje en pantalla y pararlo. Es hora de introducir los distintos tipos de variables. Cada tipo de dato tiene un límite de almacenamiento, es decir la cantidad de datos maxima que puede almacenar.

Tipo	Bytes	Rango
<i>int</i>	4	-2.147.483.648 a 2.147.483.647
<i>char</i>	1	-128 a 127
<i>double</i>	8	1.7E +/- 308
<i>long</i>	4	-2.147.483.648 a 2.147.483.647
<i>enum</i>	Varia	-
<i>short</i>	2	-32.768 a 32.767
<i>float</i>	4	3.4E +/- 38
<i>bool</i>	1	Verdadero o falso

A continuación se explican los distintos tipos de variables

- *int*: Se refiere a los enteros(integers), por lo que se podrán almacenar todos los numeros enteros que esten comprendidos en el rango expuesto en la tabla.
- *long*: Representa números grandes.
- *short*: Almacena numeros pequeños.
- *char*: Son los caracteres, en el se puede almacenar un caracter.
- *float*: Conocido como punto flotante; se refiere a todos los números decimales.
- *bool*: Son los tipos booleanos, y por tanto únicamente pueden ser verdaderos o falsos.
- *double*: Son números decimales, pero éstos son capaces de albergar una mayor cantidad de cifras.

- *enum*: Crea tipos de datos limitados a un rango específico.

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

int main()
{
    // El operador = asigna el valor 20 a la variable x,
    // tambien puede asignar el resultado
    // de una operacion matematica entre dos variables
    int x = 20;
    int y = 5;
    int suma = x + y;
    float f = 1.2;
    char c = 'a';
    bool b = true;
    cout << "Numeros enteros: " << x << " " << y << endl;
    cout << "Suma de x e y: " << suma << endl;
    cout << "Numero decimal: " << f << endl;
    cout << "Caracter: " << c << endl;
    cout << "Valor booleano: " << b;
    cin.get();
    return 0;
}
```

Una vez escrito el programa, probadlo y vereis el resultado que lanza. Ya ves como puedes asignar un valor a cada variable, imprimirlo en pantalla y sumarlos. El operador `=` es el operador de asignación. Este sirve para dar valor a una variable, o asignarle el valor de otra variable o constante.

2.3.1. Constantes

Como habrás podido observar, estas variables como *x* e *y* pueden cambiar a lo largo del programa ¿Pero y si quisiéramos que fueran valores constantes? Podemos definir las mediante la palabra clave **const** seguida del *tipo*. Ejemplo:

```
const float PI = 3.1415;
// Por convenio, las constantes iran en mayusculas
```

2.3.2. Operadores

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo(Resto)

Por ejemplo $10 \% 5$ es igual a 0 (Resto de la división).

Se muestran a continuación los operadores de *asignación compuesta*.

Operador	Equivalencia
<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>

2.3.3. Conversión de tipos ó *casting*

A veces, queremos cambiar el tipo de una variable ya declarada a otro tipo. Para esto, existe el mecanismo de la conversión de tipos que nos permite convertir de un tipo a otro. Por ejemplo, si tenemos un entero pequeño, podemos ahorrar memoria utilizando *short* en vez de *int*. ¿Pero si ya esta declarada como hacemos para convertirla? Veremos un ejemplo a continuación:

```
. . .
{
    int x = 321;
    int y = 20;
    int division = x / y;
    cout << (float) division;
. . .
}
```

Como podras observar, precedemos entre parentesis al tipo que queremos convertir la variable. En este caso, división es un número entero, por lo que al hacer la división no tendra números decimales. Para ello, la solución pasa por declarar división como *float* o aplicar *casting* sobre división. Puede ocurrir también lo siguiente:

```
int x = 9.9999999; // Imprimira 9
```

Si no se convierte x a un tipo decimal, truncará los decimales por lo que quedara como 9.

2.3.4. typedef

A veces, no nos es suficiente con los tipos que *C++* nos trae por defecto. Por ello, contamos con la herramienta **typedef**, a traves de la cual podremos crear nuestros propios tipos. Por ejemplo, si queremos ahorrarnos el escribir *unsigned int* podemos definir un tipo que defina una variable sin signo.

```
// Sintaxis: typedef <tipo de variable> <nombre del tipo>
typedef unsigned int UINT // Abreviacion de unsigned int
UINT x = 10; // Declaramos una variable sin signo negativo.
```

Como ves, **UINT** no viene por defecto en *C++*, pero hemos creado un tipo nuevo para poder crear variables sin signo.

2.3.5. enum

El tipo *enum* es un tipo distinto de datos, ya que esta limitado a unos valores constantes cuyos nombres asignaremos a voluntad. Por defecto, se numerarán empezando por el 0.

```
. . .
enum Colores {
    AMARILLO, // 0
    AZUL, // 1
    VERDE, // 2
    ROJO, // 3
    OCRE = 10; // 10;
};
Colores color; // Declaramos color, del tipo enumerado Colores
. . .
```

Por tanto, si queremos referirnos al amarillo sera el numero 0 mientras que el azul sera el 1, el verde el 2... También es posible asignar un valor distinto, con el operador de asignación `=`. Otro ejemplo de tipo enumerado:

```
. . .
enum Colores {LUN, MAR, MIE, JUE, VIE, SAB, DOM};
. . .
```

IMPORTANTE: Los tipos enumerados *no se pueden escribir ni leer*.

2.4. Entrada de datos

Hasta ahora, mediante *cout* hemos podido imprimir mensajes en pantalla. No obstante, también nos gustaria poder pedir al usuario que introdujera sus datos. Para esto, dentro de la libreria *iostream* existe **cin**. **cin** en realidad solamente lee letras, sin embargo C++ lo convertirá al tipo de dato que esté pidiendo, un entero por ejemplo. A diferencia de *cout* no utiliza el separador `<<`, sino `>>`. Pongamos un ejemplo práctico para ver como utilizarlo:

```
. . .
{
    int n = 0;
    cout << n << endl; // Aparecera 0 en pantalla
    cout << "Introduzca un numero en pantalla: "
    cin >> n; // El usuario introdujera un numero, yo pondre 100;
    cout << n; // En pantalla aparecera 10;
}
```

Como has podido ver, el usuario ha podido introducir el numero que ha querido, sin necesidad de nosotros tener que decirle a la variable **n** dentro del programa que cambie su valor. A continuación haremos un pequeño programa, que pida

al usuario un número entero, un flotante, y un caracter para posteriormente mostrarlos en pantalla.

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

int main() {
    int entero;
    float decimal;
    char caracter;
    cout << "Introduce un número entero: " << endl;
    cin >> entero;
    cout << "Introduce un número decimal: " << endl;
    cin >> decimal;
    cout << "Introduce un caracter: " << endl;
    cin >> caracter;
    cout << "El número entero que has introducido es " << entero
    << " el número decimal es "
    << decimal << " y el caracter que has introducido es " << caracter;
    cin.get();
    return 0;
}
```

Este programa, pedirá al usuario que introduzca un número entero, posteriormente un número decimal y un caracter. Posteriormente, mostrara en pantalla el valor que el usuario haya introducido en cada variable.

Capítulo 3

Subprogramas

Un *subprograma* o *subrutina* es una porción de código empaquetado que podemos llamar en otras zonas del programa, el cual puede devolver o no devolver un tipo de dato y podemos denominar como queramos. Siempre que queramos devolver un valor, debemos de hacerlo de la siguiente manera:

```
return <nombre de variable>
```

NOTA: Llamaremos *procedimiento* cuando el subprograma no devuelve nada, mientras que si devuelve algún tipo de dato se le llamara *función*. **IMPORTANTE:** Los subprogramas *nunca* se crearán **dentro** del programa principal, *siempre fuera*.

Cuando incluimos una librería, lo que realmente estamos haciendo es importar una serie de subprogramas que podemos utilizar posteriormente en nuestro programa. Para aclararlo, primero te mostraremos como ejecutar la función *seno* que esta definida dentro de la librería **cmath**. Es necesario importarla puesto que por defecto C++ no permite calcular el seno de un ángulo.

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

int main()
{
    cout << sin(30);
    cin.get();
    return 0;
}
```

La función *sin(angulo)* nos devolvera el valor del seno del angulo introducido entre paréntesis. La libreria **cmath** también nos permite calcular la raíz cuadrada, mediante *sqrt()*, el coseno mediante *cos...* Ahora, vamos a crear nuestra propia función. Ésta tendrá dos argumentos, que nosotros le daremos, y nos devolvera su valor al haber sido multiplicados.

```

#include <iostream>
#include <cmath>
using namespace std;
int multiplicar(int a, int b)
{
    return a*b; // Devuelve el resultado de multiplicar a por b.
}

int main()
{
    int x = 50;
    int y = 10;
    int resultado;
    resultado = multiplicar(x,y); // Llamamos a la funcion multiplicar
    // y multiplicara los valores x e y.
    cout << resultado;
    cin.get();
    return 0;
}

```

Como has podido ver, creamos nuestro subprograma de la siguiente manera:

```

<tipo de dato a devolver> <nombre del subprograma>(<tipo> argumento,
<tipo> argumento2, ...)
{
    . . .
    // Aqui estara el trozo de codigo que se ejecutara.
}

```

¿Y si no quisiéramos hacer una operación, ni devolver un tipo de dato? Para ello tenemos *void* que no devuelve *nada*. Un ejemplo, podría ser hacer un separador con guiones para cuando tengamos que escribir en pantalla.

```

void separador()
{
    cout << "-----" << endl;
}

int main()
{
    cout << "Hola!;
    separador();
    cout<<"Estoy separado por una cadena de guiones!";
    cin.get();
    return 0;
}

```

3.0.1. Prototipos

Hasta ahora, hemos escrito los subprogramas antes del programa principal, pero a fin de facilitar la legibilidad del código se *recomienda* poner el código del subprograma después del procedimiento principal; mientras que antes del código principal se deje constancia de que se va a escribir posteriormente el código de la función. Ésto es lo que denominamos *prototipo*. Para que se entienda más facilmente, pondremos un ejemplo:

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

void pintaAsteriscos(); // Esto se denomina prototipo.

int main()
{
    pintaAsteriscos();
    cin.get();
    return 0;
}
// Despues del codigo principal, ponemos el cuerpo del procedimiento en este caso
void pintaAsteriscos()
{
    // Esto es el cuerpo
    cout << "*****" << endl;
}
```

Es importante tener en cuenta, de que en caso de no colocarse el *prototipo* antes del procedimiento principal, no se podrá llamar al subprograma dentro de éste.

3.0.2. Paso de parámetros por referencia

Los valores por referencia son *instancias* de la misma variable. Digamos que es como la misma variable usada en muchos lugares, entonces al modificarlo en un subprograma lo harás para el resto del programa. Hasta ahora hemos asignado lo valores *ByVal*(Por valor), donde pones un numero, el número se copia y se usa en la subprograma, pero si modificas el número se modificara **solamente** en el subprograma, no en el resto.

A veces, cuando creamos un subprograma queremos que devuelva más de un valor. Sin embargo *return* no permite la devolución de dos variables distintas por lo que debemos recurrir a los parametros por *referencia*.

Pongamos en práctica esta situación creando un subprograma que calcule la solución de una ecuación de segundo grado. Como sabes, por lo general se obtienen dos resultados por lo que tenemos que pasarlos por referencia. **¿Cómo se pasan parámetros por referencia?** Agregando antes del nombre de la variable el símbolo *ampersand* **&**.

```

#include <iostream>
#include <cmath>
// Necesitamos la funcion sqrt, es decir raiz cuadrada.
void raiz(float a, float b, float c, float &sol1, float &sol2);

int main()
{
    float a = 10;
    float b = 3;
    float c = 7;
    float sol1, sol2;
    raiz(a,b,c,sol1,sol2);
    cout<< "Las dos soluciones de la ecuacion son: " << sol1
    << " y " << sol2;
    cin.get();
    return 0;
}

void raiz(float a, float b, float c, float &sol1, float &sol2)
{
    sol1 = (-b + sqrt((b*b)-(4*a*c)))/(2*a);
    sol2=  (-b - sqrt((b*b)-(4*a*c)))/(2*a)
}

```

3.0.3. Sobrecarga

Es posible crear subprogramas que se llamen igual *siempre* que tenga distinto número de argumentos, y ésto se le conoce como sobrecarga. Para que C++ distinga entre una u otra, lo hará según el número de argumentos ó *parámetros* que le pasemos.

```

#include <iostream>

int suma(int a);
int suma(int a, int b);

int main(){
    // Escogera la primera funcion, 1 solo argumento
    cout << suma(a) << endl;
    // Escogera la segunda funcion, 2 argumentos
    cout << suma(a,b) << endl;
    cin.get();
    return 0
}

int suma(int a)
{
    return a + 10;
}

```

```
}  
int suma(int a, int b)  
{  
    return a+b;  
}
```

De todas maneras, se **recomienda encarecidamente** poner nombres *claros* y *distintos* a los subprogramas que creamos.

Capítulo 4

Estructuras de control

Hasta ahora hemos podido crear pequeños subprogramas, realizar operaciones aritméticas y jugar con algunas funciones de las librerías de C++. Sin embargo, necesitamos ser capaces de poder controlar el flujo del programa, y para eso existen las diferentes estructuras de control. Existen diferentes tipos de estructuras:

- Estructuras *booleanas*.
- Estructuras iterativas.
- Estructuras de selección.

4.1. Expresiones *booleanas*

Operadores lógicos	
Operador	Significado
&	Y lógico
!	Negación lógica
	O lógico
Operaciones de comparación	
Expresion	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
!=	Distinto de

4.2. Estructuras iterativas

Las estructuras iterativas nos permiten repetir una serie de acciones de forma controlada hasta que se cumpla una condición. En el caso de los bucles *for*,

está predefinido cuando acabará, mientras que en otros como *while* o *do-while* lo hará cuando se alcance determinada condición.

4.2.1. Bucle *for*

Los bucles *for* realizarán una serie de acciones hasta que se alcance límite que establezcas.

```
for(<inicializacion>; <condicion>; <incremento/decremento>){
    . . . // Acciones a ejecutar
}
// Ejemplo:
for(int i = 0; i < 10; ++i)
{
    cout << i << endl;
}
```

Al abrir paréntesis, lo primero que ponemos es la variable sobre la que vamos a iterar y la iniciamos (Le damos un valor, 0 en este caso). Luego la condición, es en este caso que continúe mientras que *i* sea menor que 10. Finalmente, establecemos que la variable *i* tiene que incrementar *++*. En este ejemplo, se imprimirán en pantalla los 10 primeros números empezando desde 0. Podríamos haber inicializado la variable *i* en 1, 2, 3...

OJO: Si establecemos como condición que *i* sea menor que 10, y en vez de incrementar le decimos al bucle que *decremente* (*-i*), *nunca* se cumplirá la condición y por tanto, el programa *explotará*.

4.2.2. Bucle *while*

El bucle *while*, se define de la siguiente manera:

```
while(condicion)
{
    . . .
    acciones a realizar
    . . .
}
```

Mientras que la condición que hayamos establecido en el paréntesis sea falsa, se ejecutará todo el contenido de los corchetes. Pongamos un ejemplo de como utilizar esta estructura.

```
int x = 10;
while (x < 0)
{
    cout << x << endl;
    --x
}
```

```
}

```

Esta vez, empezaremos a contar 10 e imprimiremos en pantalla los valores de **x**, y cada iteración *decrementará* su valor en uno, hasta que se alcance la condición establecida.

4.2.3. *do-while*

El bucle *do-while* lo utilizaremos de la misma manera que *while*, pero cuando queramos asegurarnos de que se ejecute el código dentro de él al menos una vez. Pongamos un ejemplo de su estructura:

```
do {
    // Código a ejecutar aqui
    . . . . .
}
while(<condicion>);
```

Y ahora pongamos un ejemplo práctico:

```
int i = 0;
do {
    cout << "Esta es la interacion numero:" << i << endl;
    ++i;
}
while(i < 10);
```

Como ves, la variable **i** empezará en 0, y primero se ejecutara una vez imprimiendo el valor 0. Tras esto incrementa una vez su valor, y repetirá este conjunto de acciones hasta que se cumpla la condición **i < 10**.

4.3. Estructuras de control

A veces queremos tomar decisiones a lo largo del programa, o que el programa se comporte de una manera u otra dependiendo de una determinada condición. Es por ello que tenemos las estructuras de control.

4.3.1. *if-else*

La estructura *if-else* nos permitirá, según una condición dada ejecutar un *set(conjunto)* de instrucciones u otro. Veamos cómo se construye uno:

```
if(condicion)
{
    // Instrucciones a ejecutar si es cierto
}
else {
```

```
// Instrucciones a ejecutar si es falso
}
```

Y posteriormente, un ejemplo práctico.

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

int main()
{
    int n = 3;
    if(n % 2 == 0)
    {
        cout << "Es un número par" << endl;
    }
    else
    {
        cout << "Es un número impar" << endl;
    }
    cin.get();
    return 0;
}
```

En este ejemplo, tenemos **n**, que es un número impar. Lo que hacemos, es comprobar si el resto de dividir **n** entre 2 es *ceró*. En caso de serlo, sería un número par y aparecería en pantalla un mensaje avisando de que es un número par. Por el contrario, si el resto es distinto de 0, entonces no será un número par y por tanto imprimirá como en éste caso que **n** es un número *impar*.

4.3.2. Bloques *if* anidados

También es posible anidar los bloques *if*, por lo que podemos introducir otra condición una vez cumplida la anterior. Para verlo más fácilmente, vamos a poner un ejemplo:

```
int x = 7;
int y = 10;
if(x % 2 == 0)
{
    if(y % 2 == 0)
    {
        cout << "Ambos números no son primos" << endl;
    }
    else
    {
        cout << "Solo x es par" << endl;
    }
}
```

```
}
```

4.3.3. *switch*

La estructura *switch* sirve para sustituir muchos *if* diferentes que se hagan sobre la misma variable. Vamos a poner un ejemplo y su estructura:

```
switch(<variable>)  
{  
    case <valor>:  
        //codigo  
        break; // Para acabar el case, pondremos break;  
    case <valor2>:  
        //codigo  
        break;  
    // Default se refiere, a que en caso de que  
    // no sea ninguna de las opciones anteriores  
    // y debemos ponerlo siempre.  
    default:  
        //codigo default  
        break;
```

Ejemplo:

```
int a = 4;  
switch (a)  
{  
    case 1:  
        cout << "Es el numero uno" << endl;  
        break;  
    case 2:  
        cout << "Es el numero dos" << endl;  
        break;  
    case 3:  
        cout << "Es el numero tres" << endl;  
        break;  
    case 4:  
        cout << "Es el numero cuatro" << endl;  
        break;  
    default:  
        cout << "No es ningun numero del 1 al 4" << endl;  
        break;  
}
```

Como ves, dependiendo del número que sea *a* se imprimirá un mensaje diciendo su número. Si *a* no fuera un número del 1 al 4, entonces se utiliza *default* para

indicar que no es ninguno de los otros. No solamente estamos retringidos a números, como en el caso anterior, sino también a caracteres.

```
switch(c)
{
    case 's':
        sin(x);
        break;
    case 'c':
        cos(x);
        break;
    default:
        cout << "No se ha introducido ni la letra 's' ni la letra 'c'";
        break;
}
```

Capítulo 5

Arrays

5.1. Conceptos básicos

Los *arrays* o *matrices* sirven para almacenar grandes cantidades de información de forma estructurada. Digamos que básicamente es una forma de guardar muchos valores bajo el mismo nombre. Algo importante sobre los array es que no se empiezan a contar desde la posición 1, sino desde **0**. Puedes hacer arrays de todo tipo, desde enteros a *punteros* y *estructuras*.

Imagina que los *arrays* son como una cajonera. La variable sobre la que lo declaras es la cajonera, y cada cajón es un elemento del *array*, donde puedes almacenar diferentes objetos o datos.

Digamos que esto seria un *array* de 7 cajones, o *posiciones*:

□□□□□□□

Cuando metemos en el elemento 0, es decir en el primer cajón un número; la cajonera quedaria de la siguiente manera:

[10]□□□□□□□

5.1.1. Cómo declarar *arrays* en C++

Puedes hacerlo de la siguiente manera:

```
int numeros[10]; // Crea un array de enteros
// Con 10 cajones, o posiciones.
// [] [] [] [] [] [] [] [] [] []
```

Una vez hecho ya tenemos nuestros 10 cajoncitos para guardar los datos, pasamos entonces a almacenar datos en algunos cajones.

```
int numeros[10];
numeros[1] = 3;
numeros[5] = 1;
numeros[2] = 0;
```

¿Cómo quedarían los cajones ahora?

```
[[3][0]][1][1]]
```

5.1.2. Arrays multidimensionales

El aspecto más interesante de los *arrays* son los **multidimensionales**. Podemos crear *arrays* de diferentes dimensiones. `[[[]]`

De esta manera, tendríamos un array de 2x2. ¿Cómo lo declaramos en C++? Lo hacemos en un momentito:

```
int dimension[1][1];
// dimension[i][j] siendo i y j las posiciones.
dimension[0][0] = 5;
dimension[1][1] = 2;
```

De esta manera, nos queda:

```
[5][2]
```

Ahora te preguntarás, ¿y si quiero conocer todos los valores del array? Antes hemos visto la estructura *for* que nos permitirá recorrer todo el array.

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

int main()
{
    int matriz[5] = {100,26,277,65,288};
    for(int i = 0; i < 5; ++i)
    {
        cout << matriz[i] << endl;
    }
    cin.get();
    return 0;
}
```

Este programa, inicializará un array con los valores 100,26,277,65 y 288; y en el bucle *for* imprimirá en pantalla el contenido de cada posición del array.

5.1.3. Arrays en subprogramas

No se pueden pasar los arrays por copia (*ByVal*), por lo que tenemos que pasarlos como argumentos por referencia. Para poder hacer esto, tenemos que pasar al procedimiento el *array* de la siguiente manera.

```
#include <iostream>
using namespace std;
/* Tenemos que especificar que es un array
Poniendo en uno de los argumentos el tipo
Y seguidamente el nombre del array
```



```

seguido de corchetes cerrados.*/

void imprimirArray(int arr[], int n)
{
    for(int i = 0; i < n; i++)
    {
        cout << arr[i] << endl;
    }
}

int main()
{
    arr[5] = {10,15,20,36,10};
    imprimirArray(arr);
    cin.get();
    return 0;
}

```

¿Qué pasa si los *arrays* no se copian? Si no se copian, entonces no podemos devolverlos, por tanto utilizar *return* estará prohibido.

Pondremos otro ejemplo, en éste caso cómo calcular el producto escalar.

```

float productoScalar(float u[], float v[], int n)
{
    int res = 0;
    for(int i = 0; i < n; ++i)
    {
        res += u[i] * v[i];
    }
    return res;
}

```

En este caso **si** utilizamos *return*, pero es porque estamos devolviendo el valor que tiene almacenado la variable **res**, que es de tipo entero.

5.2. Cadenas de caracteres

Hasta ahora, solamente hemos podido almacenar números o un único carácter. Se pueden introducir cadenas de caracteres, una de las formas es crear un array de tipo *char*.

```

char c[100];
c[0] = 'a';
c[1] = 'e';
.
.
.

```

Pero como ves, esto es muy limitado. Lo primero, es que quedan espacios sin rellenar y rellenarlo todo de espacios en blanco *no lo permite el compilador*. ¿Cómo distinguimos cuando parar? Podemos poner el caracter `'\0'` que en la tabla *ASCII* representa el caracter nulo. De esta manera, podemos decirle al programa que pare justo cuando encuentre éste carácter.

```
int i = 0;
while(s[i] != '\0')
{
    cout << s[i];
    ++i;
}
```

Hagamos un ejemplo donde nuestro programa imprima nuestro nombre.

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

int main()
{
    char nombre[50];
    nombre[0] = 'F';
    nombre[1] = 'e';
    nombre[3] = 'e';
    nombre[4] = 'r';
    nombre[5] = 'n';
    nombre[6] = 'a';
    nombre[7] = 'n';
    nombre[8] = 'd';
    nombre[9] = 'o';
    nombre[10] = '\0';
    while(nombre[i] != '\0')
    {
        cout << s[i];
        ++i;
    }
    cin.get();
    return 0;
}
```

5.2.1. *Strings*

Como hemos visto, hacer un array de *char* no es una solución cómoda, ni si quiera fácil de manipular. Para ellos existen lo que denominamos **strings**, que es un **objeto** básico de C++.

Nota: A veces es necesario añadir en la cabecera del programa, además de `<iostream>` la cabecera `<string>`

Los strings, a diferencia de un array de *char* tienen longitud indefinida, y traen métodos que permiten manipular con *facilidad* cualquier string que hallamos creado. Vamos a ver a continuación como sería el ejemplo anteriormente expuesto, pero trabajando con **strings**.

```
#include <iostream>
#include <string> // Por si las moscas
using namespace std;

int main()
{
    string nombre;
    nombre = "Fernando";
    cout << nombre << endl;
    string nombreUsuario;
    cout << "Introduce tu nombre: ";
    cin >> nombreUsuario;
    cout << nombreUsuario;

    cin.get();
    return 0;
}
```

Como ves, es muchísimo más sencillo manipular cadenas de caracteres creando objetos **string**. Listamos a continuación una serie de métodos útiles de string:

- *length*: Devuelve la longitud de la cadena
- *size*: Devuelve el tamaño de la cadena.
- *getline*: Al pedir al usuario una cadena, capta todo hasta que hayas pulsado *enter*.
- *substr*: Extrae una subcadena.

Capítulo 6

Tratamientos secuenciales

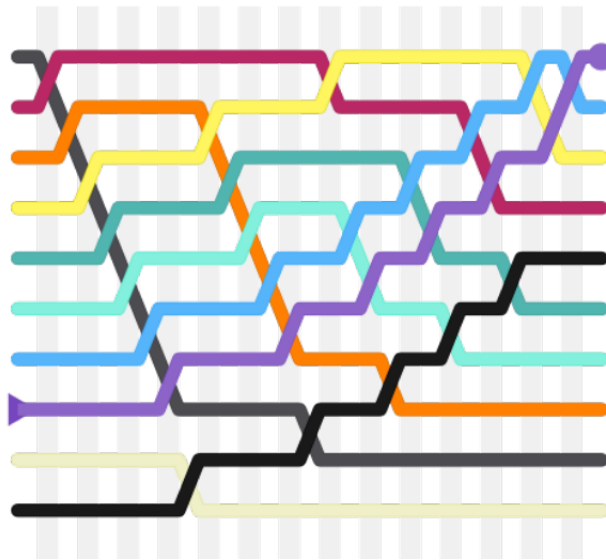
6.1. Algoritmos de ordenación

Podemos definir un *algoritmo de ordenación* como aquel que es capaz de poner elementos de una lista en un orden determinado. Es importante lograr un algoritmo de ordenación eficiente, para optimizar el uso de otros algoritmos que requieran de una lista ordenada; como pueden ser funciones que pretendan unir dos listas o buscar elementos dentro de ella. También se utiliza para facilitar la lectura de la salida de datos.

A continuación trataremos diferentes algoritmos de ordenación, unos más efectivos que otros. No quiere decir que éstos sean los únicos métodos, pero son los más usados.

6.1.1. *Bubble sort*

Se trata de un algoritmo simple, que funciona ”*saltando*” continuamente a través de la lista, comparando cada par de números e intercambiándolos si están en el orden equivocado. Este procedimiento se repite hasta que no quedan más intercambios que hacer, lo que indica que la lista está ordenada. A pesar de ser un algoritmo simple, existen métodos mucho más eficientes para listas grandes.



Bubble sort, expresado gráficamente.

A continuación, expondremos como aplicar dicho algoritmo a un programa real.

```
void bubbleSort(int a[], const int N)
{
    for(int tope = N-1; tope >= 1; tope--)
    {
        for(int i = 0; i < tope; ++i)
        {
            if(a[i] > a[i+1])
                intercambia(a[i], a[i+1]);
        }
    }
}
```

6.1.2. Select sort

Este algoritmo no es más que una combinación de búsqueda y ordenación. En cada paso, el elemento sin ordenar con el menor(o mayor) valor se mueve a la posición que debe en la lista. En este algoritmo, el *bucle interior* busca el menor de todos los números, y el *bucle exterior* introduce este valor en el lugar que debe.



Select sort, gráficamente.

```
void selectSort(int arr[])
{
    for(int i = 0; i < sizeof(arr)/sizeof(int); ++i)
    {
        int indice = i;
        for(int j = i + 1; j < sizeof(arr)/sizeof(int); ++j)
            if(arr[j] < arr[indice])
                indice = j;

        int numeroMasPequeno = arr[indice];
        arr[indice] = arr[i];
        arr[i] = numeroMasPequeno;
    }
}
```

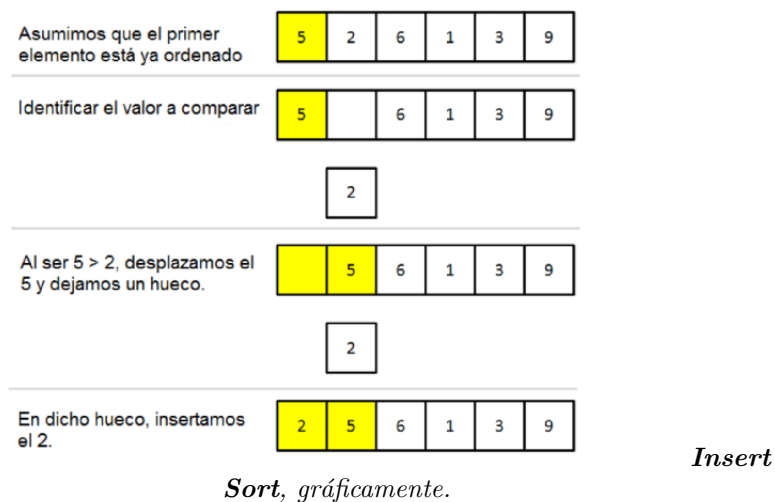
6.1.3. Insert sort

Se trata de otro de los algoritmos simples. Es **mucho** menos eficiente en listas grandes que otros algoritmos.

Ventajas de este algoritmo:

- Es muy simple
- Es muy eficiente para listas pequeñas
- Es estable
- Necesita una constante cantidad de memoria adicional.

Este algoritmo itera sobre la lista, saca un elemento de la lista, encuentra el lugar donde pertenece y lo inserta en él. Este proceso se repite hasta que no queden más elementos por ordenar.



A continuación, mostramos como aplicar este algoritmo.

```
void insertSort(int arr[])
{
    int t;
    for(int i = 1; i < sizeof(arr)/sizeof(int); ++i)
    {
        for(int j = i; j > 0; --j)
        {
            if(arr[j] < arr[j-1])
            {
                t = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = t;
            }
        }
    }
}
```



```

    }
  }
}

```

6.2. Búsqueda de caracteres

A veces, tendremos que encontrar en largas cadenas de caracteres determinados grupos. Por ejemplo, aplicado a biología podemos poder encontrar pares de bases como **AC**. En otros casos, podemos querer encontrar cual es el primer **7** en una cadena de caracteres, o incluso contar cuantos hay.

Para ello, podemos recurrir a una función muy simple que nos permitirá buscar el caracter o número que queramos:

```

int buscar(int a[], int n, int x)
{
    int i = 0;
    while ( i < n and a[i] != x ) // mientras no lo encuentre, seguira buscando
    {
        ++i;
    }
    if( i >= n ) // Retornar error en el caso de que sea superior o no se encuentra
    {
        return -1;
    }
    else // Devolver la posicion en la que se encuentra dicho numero.
    {
        return i;
    }
}

/* Version mas simplificada de dicho algoritmo.
*/
int buscar(TVector v, int x)
{
    for (int i = 0; i < N; ++i)
        if(a[i] == x) return i;
    return -1;
}

```

Podemos aplicar dicho algoritmo a un programita practico, que nos localice el numero 2 dentro de un *array*. Para ello, conociendo la función anteriormente expuesta, basta con aplicarla en nuestro programa.

```
/* En este programa, dado un array de numeros
buscaremos en el la posicion donde se encuentra.
Se podria aplicar de manera analoga,
modificando la funcion original para saber simplemente
si existe o no dicho caracter. */

int main() {
int arr[] = {4,3,3,0,5,6,3,7,2,9,9,55};
int x = 2;
// sizeof(a)/sizeof(int) devolvera el numero total de elementos de dicho

cout << "Se encuentra en:" << buscar(arr, sizeof(arr)/sizeof(int), x) <
return 0;
}
```

Capítulo 7

Estructuras

Cuando programamos, conviene tener un único nombre con el que referirnos a un grupo determinado de valores. A través de las estructuras, podemos almacenar diferentes valores en variables de diferentes tipos bajo el mismo nombre. Suelen ser muy útiles cuando es necesario agrupar mucha información junta. Podemos usarlo para crear una estructura, como puede ser una agenda telefónica que contenga datos como el teléfono, nombre, apellidos... Obviamente, son distintos tipos.

NOTA: En cuestiones de optimización de código, ayuda a tenerlo más organizado y a facilitar la reedición de código.

La forma de definir una estructura, es a través de la palabra clave **struct**:

```
struct Agenda {  
    int numero;  
    String nombre;  
    String apellidos  
};
```

7.1. Manipulando estructuras

Para acceder a una variable de la estructura, podemos hacerlo de la siguiente manera:

```
Agenda.numero;
```

Lo cual nos permitirá manipular las variables dentro de la estructura. Esto nos será útil a la hora de crear arrays de estructuras, donde poder almacenar una agenda telefónica completa por ejemplo, con cientos de contactos.

Pondremos un par de ejemplos, primero un ejemplo de manipulación y otro en un programa real.

```
struct Agenda {
```

```

        int telefono;
        String nombre;
        String apellidos;
};

struct Agenda mi_agenda;

mi_agenda.telefono = "6123458789";

```

De esta manera, podríamos haber devuelto en pantalla el valor de teléfono que previamente estaba vacío. Sin embargo, en esta ocasión hemos decidido poner un número aleatorio dentro de dicha variable. Ahora, pasemos a un ejemplo en un programita real:

```

struct Agenda {
    int telefono;
    String nombre;
    bool amigo; // esto nos indicara si es amigo nuestro o no
};

int main()
{
    struct Agenda mi_agenda;
    mi_agenda.telefono = "959795651";
    mi_agenda.nombre = "Francisco";
    mi_agenda.amigo = true;
    cout << "El número de teléfono es: " << mi_agenda.telefono;
    cout << ", pertenece a ";
    cout << mi_agenda.nombre << " y es amigo ";
    cout << mi_agenda.amigo << endl;
    return 0;
}

```

7.1.1. Array de estructuras

Es posible crear **arrays** de estructuras, y una vez creadas se pueden manipular de manera similar a los arrays convencionales, con la diferencia de tener que acceder a la variable de la manera expuesta anteriormente. *¿Para qué sirve esto?* Sencillo, nos permite en el caso de la Agenda, no tener que crear diferentes agendas para almacenar distintas personas. Gracias a esto, podemos crear por decirlo así, una agenda telefónica completa con el número de páginas que deseemos. Expongamos a continuación un ejemplo de esto:

```

// struct tipo variable[N];
struct Agenda mi_agenda[3];
// Nos aseguramos de que no haya nada al principio, para no tener

```

```
// posible basura.  
for(int i = 0; i < 3; ++i){  
    mi_agenda[i].telefono = 0;  
    mi_agenda[i].nombre = "";  
    mi_agenda[i].amigo = false;  
}  
  
// Si queremos asignar por ejemplo, a la primera pagina datos, basta con:  
    mi_agenda[0].telefono = "959795651";  
    mi_agenda[0].nombre = "Francisco";  
    mi_agenda[0].amigo = true;
```

Como ves, manipular estructuras no es tan complicado después de todo. Cuestión de practicar y pensar en formas de aplicar estructuras a nuestros programas, ya que cuando le coges el truco no dejarás de utilizarlas.

Capítulo 8

Ficheros

Conforme avancemos, queremos trabajar con información externa a nuestro programa en lugar de tener que proporcionársela nosotros a lo largo del código. Es por ello, que trataremos en este tema el tratamiento de ficheros, cómo trabajar con ellos y los distintos tipos que podemos encontrarnos. Además, esta información persistirá una vez apagado nuestro ordenador, a diferencia de la información a portada por *nosotros* a lo largo del programa. Cabe especificar, que con información aportada, se refiere a asignación de valores a variables y no de código en sí.

8.1. ¿Qué es un fichero?

Podemos definir un **fichero**, como una serie de datos almacenados de forma **persistente**. Con persistencia, se refiere a la constancia de dicha información aunque se desconecte de la red eléctrica el dispositivo que lo almacena.

Dicho *fichero*, se puede crear con cualquier programa, y se identifican por un **nombre** y una extensión precedida de un ".". Conocer dicha extensión, nos da información acerca de cómo abrir ese fichero y en la mayoría de los casos **qué** almacena.

Ejemplo de fichero:

notas.txt A través de dicho nombre, sabemos que *notas* es el nombre del archivo. Esto nos puede servir para tener una idea sobre qué almacena, en este caso supondremos que son notas de clase. **txt** nos dice que es un fichero de *texto plano*, y que probablemente haya sido guardado con un editor como **notepad**(*bloc de notas*) o **gedit**(**E**ditor de **linux**).

8.1.1. ¿Legible o no legible?

A pesar de lo que creamos, no podemos leer todos los tipos de ficheros. Con esto, nos referimos a si somos capaces de ver su contenido a simple vista, sin hacer uso de programas que interpreten su contenido. Por ejemplo, los ficheros

de texto podemos leerlos sin ningún problema. Pueden ser más o menos complejos, pero somos capaces de ver a simple vista su contenido. Sin embargo, contamos con los **binarios**, como son imágenes, las cuales no podemos entender sin un programa que interprete el significado de lo que se encuentra dentro de dicho archivo. Lo mismo ocurre con los ficheros **.mp3**, que necesitamos de un reproductor para que nos enseñe su contenido.

De la misma manera, un programa una vez **compilado**, podemos ejecutarlo pero **no visionar el código fuente**.

8.2. Tipos de ficheros

Se expondrán a continuación distintos tipos extensión de ficheros, para fa-

	Extensión	Tipo
	<code>.cpp</code>	Archivo fuente de C++
	<code>.h</code> , <code>.hpp</code>	Librerías de C, C++
	<code>.txt</code>	Documento de texto plano
miliarizarnos con ellos	<code>.html</code>	HyperText Markup Language (web)
	<code>.pdf</code>	Documento
	<code>.zip</code> , <code>.rar</code>	Comprimido de ficheros
	<code>.jpg</code> , <code>.png</code>	Imágenes
	<code>.rtf</code>	Texto enriquecido

8.3. Manipulación de ficheros

Los ficheros en C++ se tratan a través de flujos, o **streams** de entrada/salida. **cout** sería la salida de un fichero, mientras que **cin** el fichero de entrada. A la hora de tener un fichero, hemos de llevar a cabo *generalmente* las siguientes acciones:

- Abrir el fichero
- Manipular el fichero (Vease el siguiente apartado)
- Cerrarlo.

A continuación, mostraremos como abrir un archivo, y leer sobre él.

```
#include <iostream>
#include <fstream> // Necesario para la manipulacion de ficheros

int main() {
    /* Este programa, asignara a in, el flujo
    de los datos contenidos en numeros.dat
    y mientras exista contenido en el, lo
    vertira sobre n. En caso de no encontrarse
    el fichero, lanzara un error.
    */
```



```

int n;
ifstream in("numeros.dat");
if(in){
    in >> n;
} else {
    cout << "Error en la apertura del fichero" << endl;
}
in.close(); // Cerrara el flujo de entrada.
}

```

Podemos hacer uso de un string para leer el archivo, aunque necesitaremos hacer uso de `cstr` para poder utilizarla en el argumento.

```

...
string nomFich = "agenda.dat";
ifstream in(nomFich.c_str());
if(in)...
...

```

A continuación, mostraremos cómo leer el contenido del fichero:

```

...
ifstream fich("prueba.txt");
if(fich) {
    string s;
    while(getline(fich,s)) {
        cout << s << endl;
    }
}
...

```

De esta manera, mostraremos en pantalla el contenido de **cada línea** del contenido del fichero. Esto es debido a `getline`, que nos permite realizar dicha acción.

Una vez hecho esto, también queremos escribir dentro del fichero. Para ello, tendremos que cambiar el `ifstream` por `ofstream`. De esta manera, sería el flujo `ostream` el que nos permitiría trabajar de dicha forma.

```

...
ofstream fich("datos.dat");
if(fich) {
    fichsal << "Estoy escribiendo en el fichero..." << endl;
    fichsal << "y estoy acabando de escribir en el, ¡adios!" << endl;
}
...

```


Capítulo 9

Anexos

9.1. El compilador y el IDE. Fuente de muchos problemas

¡No te saltes este capítulo! Muchos tienen problemas con la instalación del compilador, y otros muchos no siguen el hilo de la programación porque simplemente éste no les funciona.

De ahora en adelante, tu editor de texto, *IDE*, junto a tu compilador serán tus armas. En ellos, diseñarás programas, los probarás, te equivocarás y aprenderás.

Antes, introduciremos los conceptos de compilador e *IDE*, de la forma más cerca al lector posible; sin entrar en detalles técnicos pues el objetivo de esta obra no es presentar los conceptos al mayor nivel técnico posible.

Para que nos entendamos, *IDE* (*Integrated development environment* - *Entorno de desarrollo integrado*), no es más que una aplicación con un editor de texto integrado, que posee herramientas automáticas de construcción (Compiladores), y un depurador.

Algunos diréis, ¿para qué narices querré un editor de texto si mi sistema operativo ya trae uno integrado? Muy sencillo, los editores como *emacs*, *Microsoft Visual Studio*, *netBeans*, *Eclipse*... Entre otros muchos, señalan errores en el programa, colorean las palabras clave e incluso cuidan de que cierres bien las llaves. Además, coloreará y señalará las variables, y funciones para que de esta manera sea sencillo y agradable leer el código.

Los compiladores para que nos entendamos, convertirán nuestro programa a un código que nuestro ordenador entienda, y de esta manera pueda *ejecutarlo*.

El depurador, es una herramienta muy poderosa. A menudo en proyectos grandes (Y no tan grandes), se cometen errores que no vemos a simple vista, o se obtienen resultados inesperados. Este nos permitirá localizar errores, dónde se producen y así poder arreglarlos. Durante este curso, no nos hará mucha falta el uso del depurador, pero al menos debes saber qué es.

9.1.1. Instalación del compilador

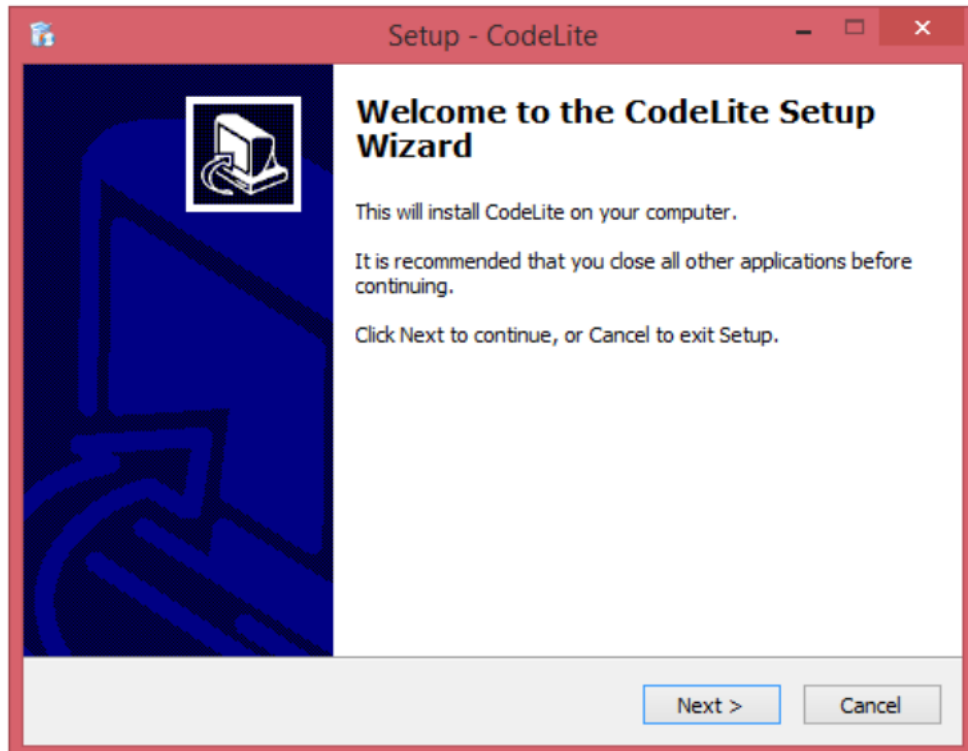
En este libro, se enseñará la instalación de un compilador para Windows y Linux. Supongo que la instalación en *MacOSx* será parecida a la de Linux, pero al no tener acceso a uno documentaré solamente aquellos a los que sí tengo acceso.

Primero procedemos a hacerlo en la plataforma *Windows*. Para ello, haremos uso del entorno *CodeLite*. ¿Por qué? Es sencillo e intuitivo, y además de ser *Open Source* es *multiplataforma*.

Para descargarlo, podéis acceder él vía <http://codelite.org>, más concretamente en <http://downloads.codelite.org>.

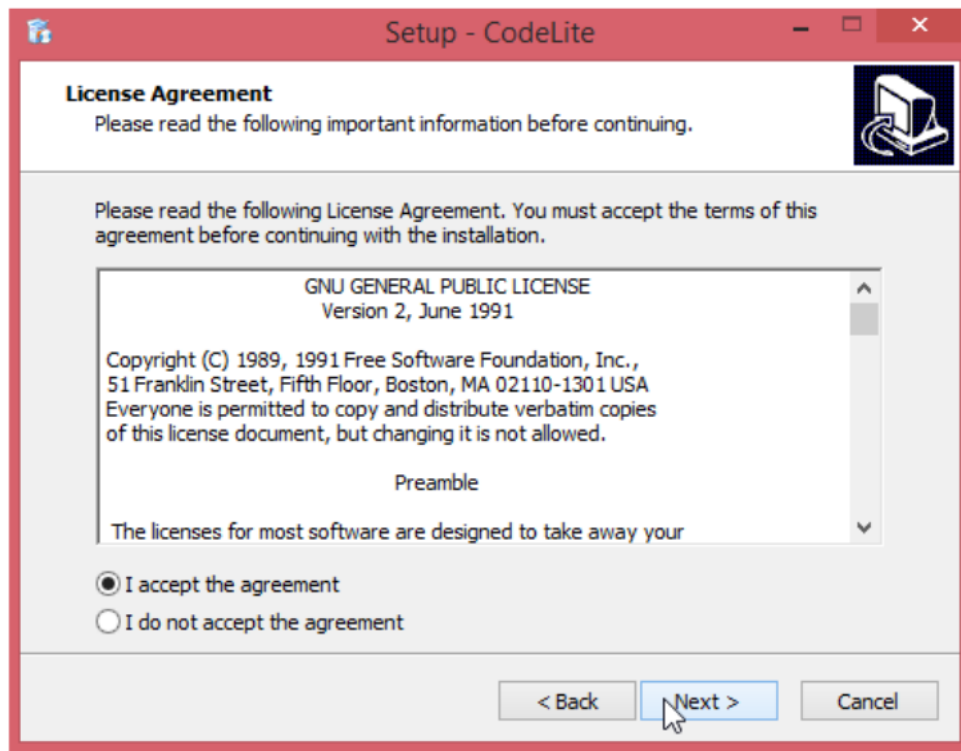
Al entrar en el segundo enlace, encontraremos diferentes versiones. Si utilizas un sistema operativo de *32-bits*, baja la versión '*CodeLite 7.0 for Windows 32 bit installer*'. Pulsa en *Direct Link* y se descargará tu archivo. De forma análoga, si tu versión de Windows es de *64 bits*, escoge la de 64.

Una vez descargado, ejecuta el archivo y tendrás una ventana como esta

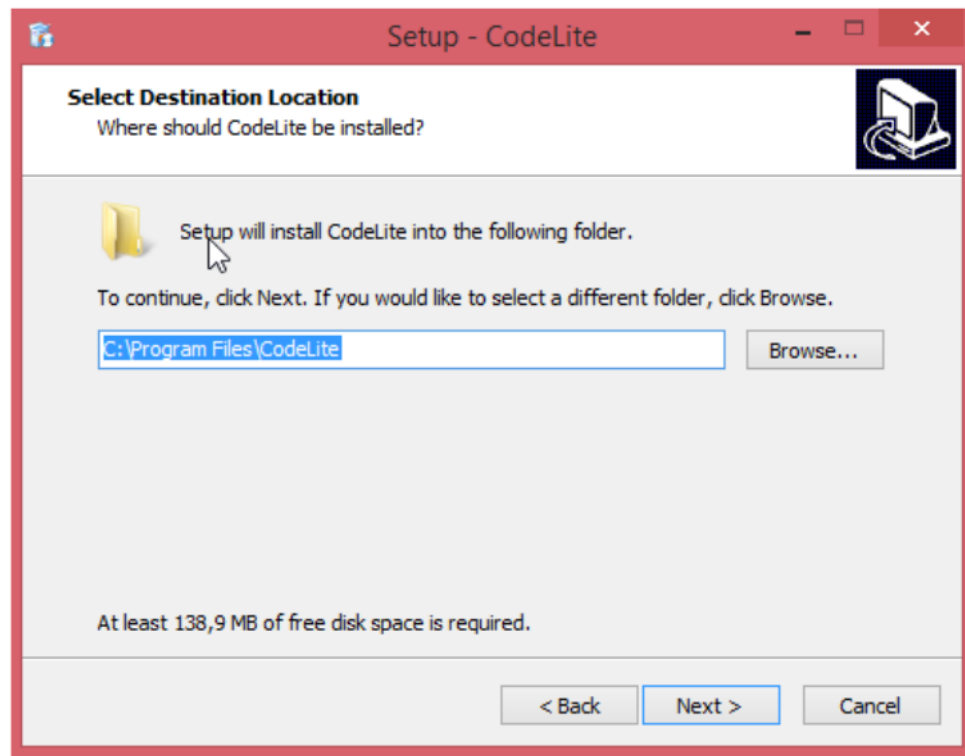


Pulsamos *next*.

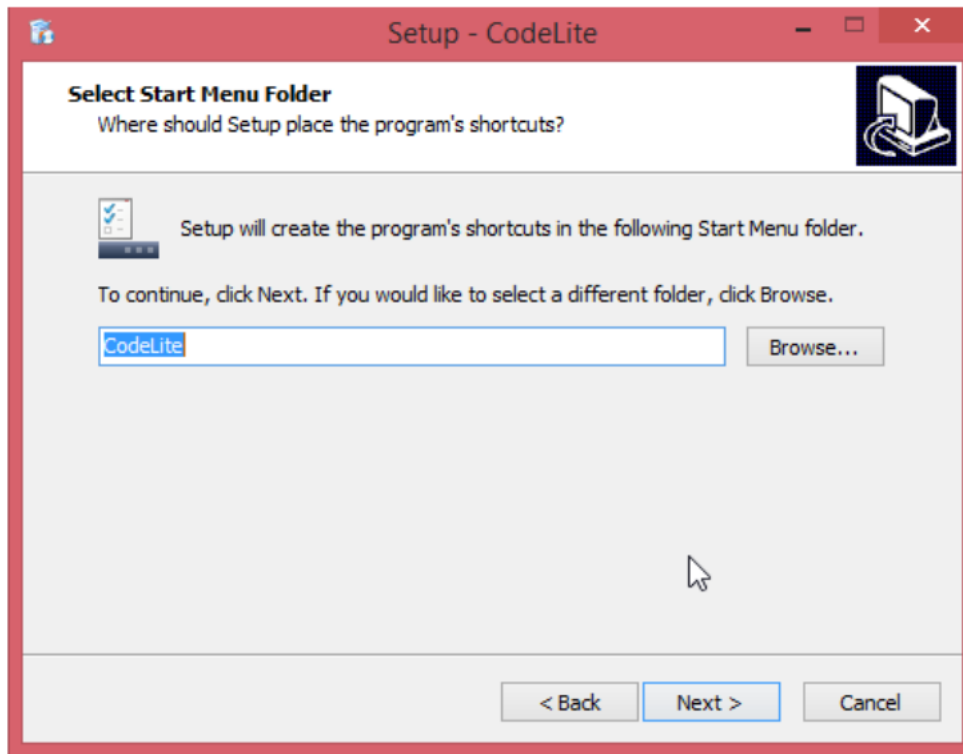
9.1. EL COMPILADOR Y EL IDE. FUENTE DE MUCHOS PROBLEMAS53



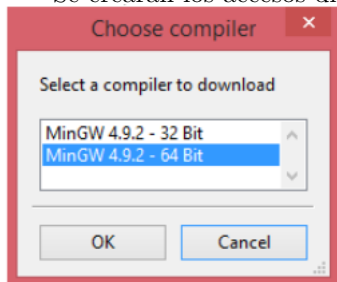
'Leemos' y aceptamos el acuerdo, luego pulsamos *next*



Elegimos la ruta donde queremos instalar la aplicación, en mi caso mantendré la ruta por defecto, aunque eres libre de cambiarla a tu elección.

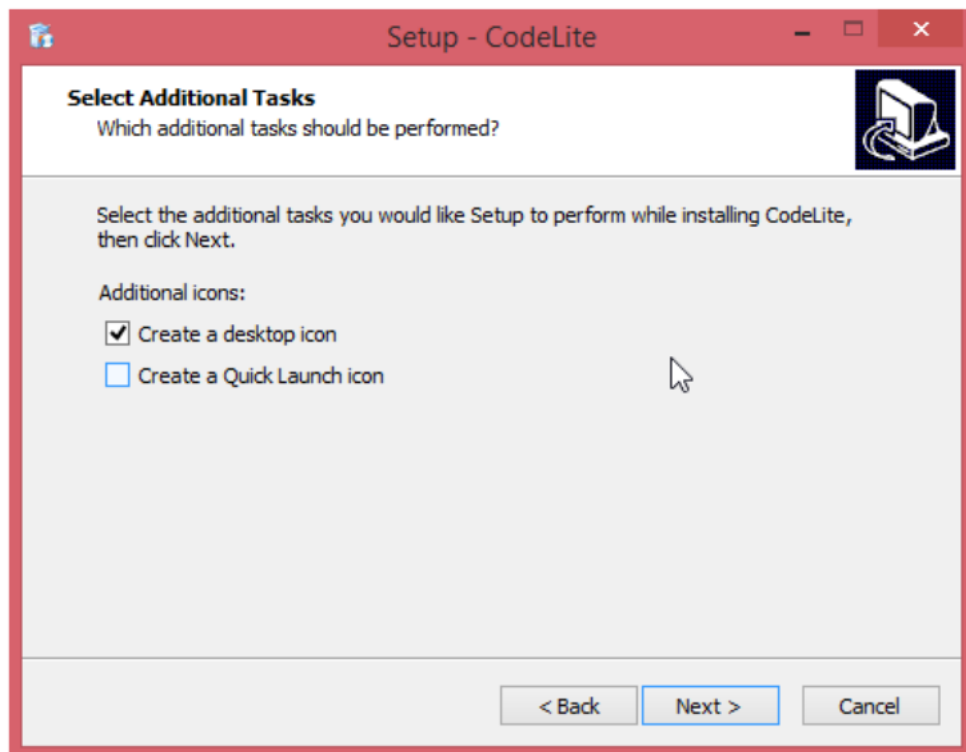


Se crearán los accesos directos en este directorio desde el menú inicio

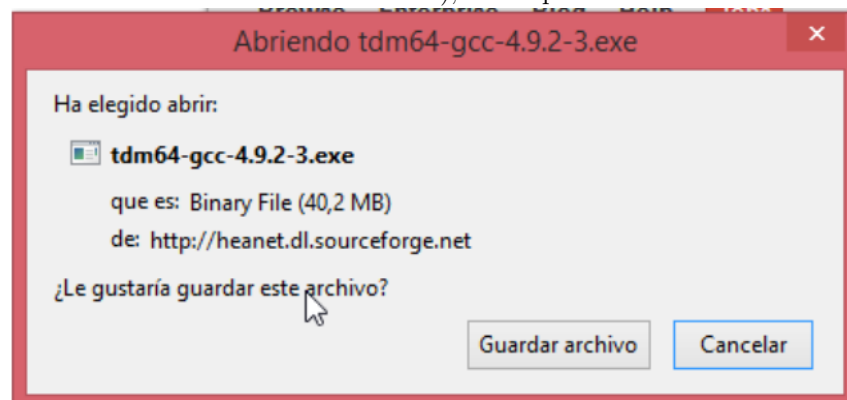


Seleccionamos *Create a desktop icon*, ya que nos permitirá acceder a la aplicación de forma fácil y rápida.

Una vez finalizada la instalación, vamos al escritorio y encontraremos 'CodeLite', por lo que procederemos a abrirla y lo primero que nos dirá es que no tenemos ningún compilador instalado. Te recomendará la instalación de *MinGW*, accedemos a instalarla como se muestra a continuación:

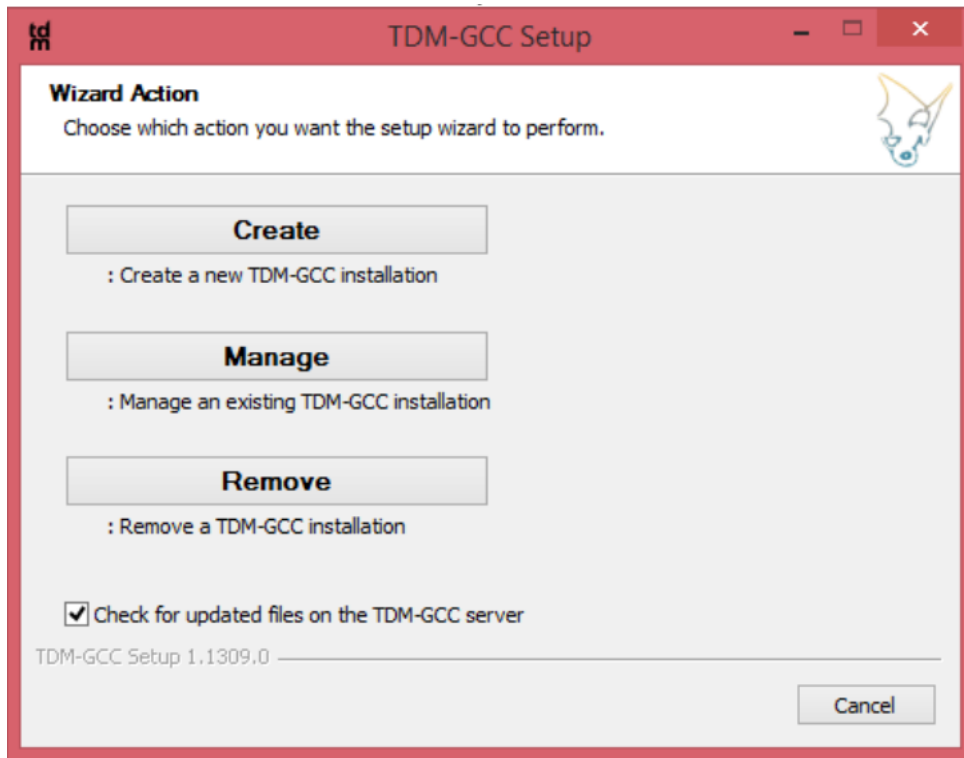


Seleccionamos *MinGW 64-bit* o 32 si nuestro sistema es de 32-bits. Una vez hecho esto, nos mandará a una página donde se descargará (Si nada falla, automáticamente), el compilador.

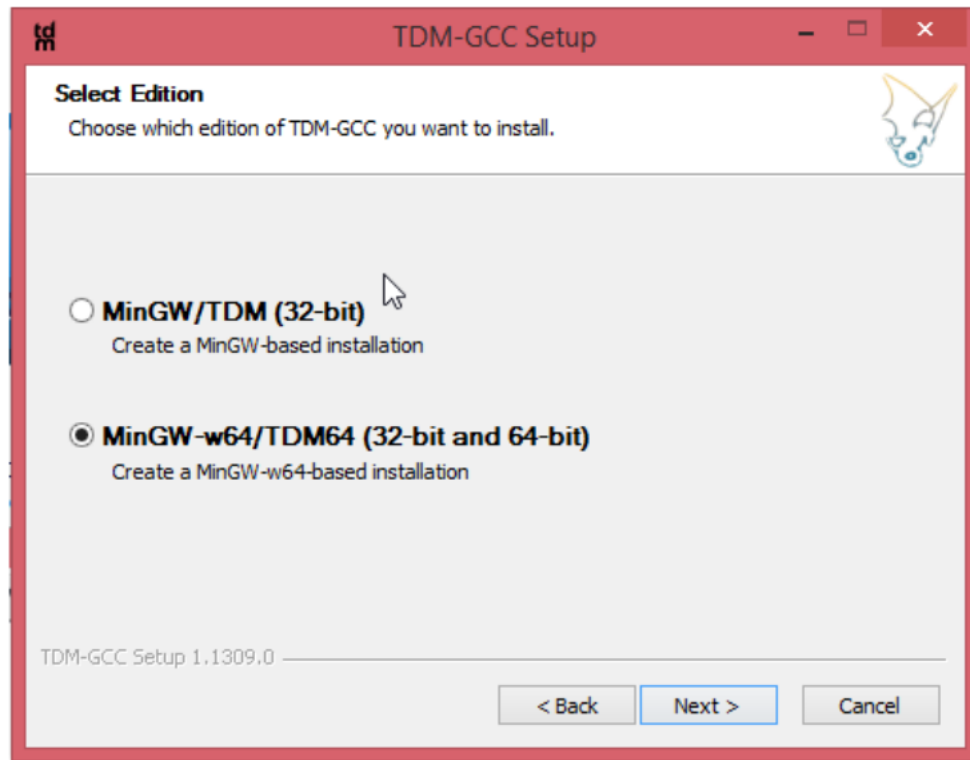


Guardamos el archivo y lo instalamos

9.1. EL COMPILADOR Y EL IDE. FUENTE DE MUCHOS PROBLEMAS57

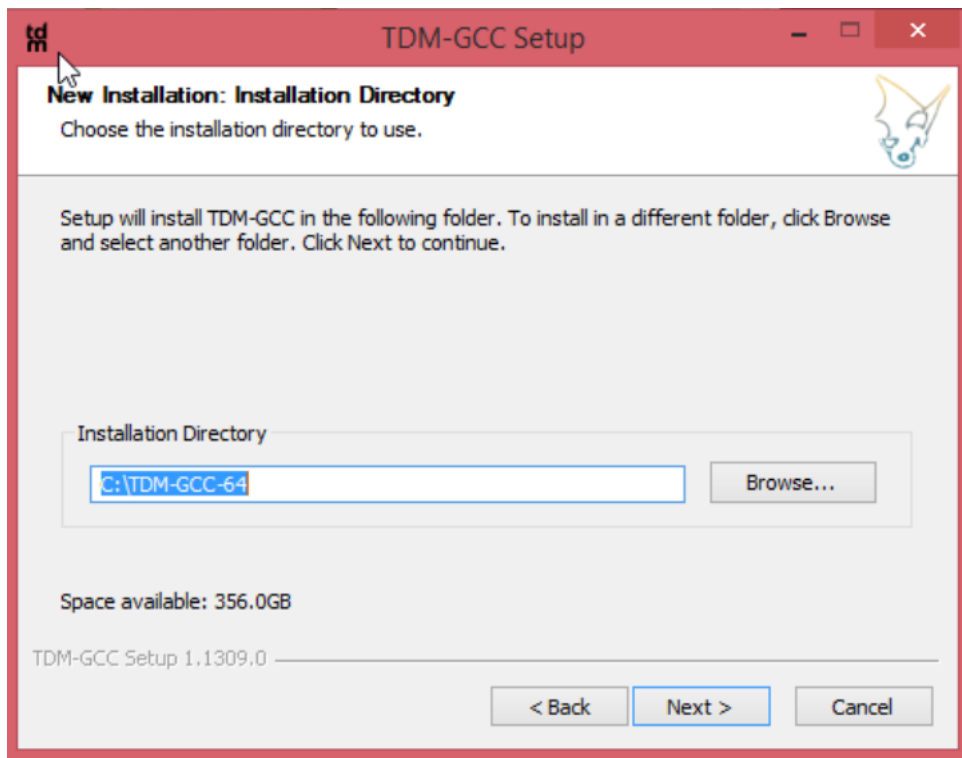


Una vez abierto, seleccionamos 'Create'



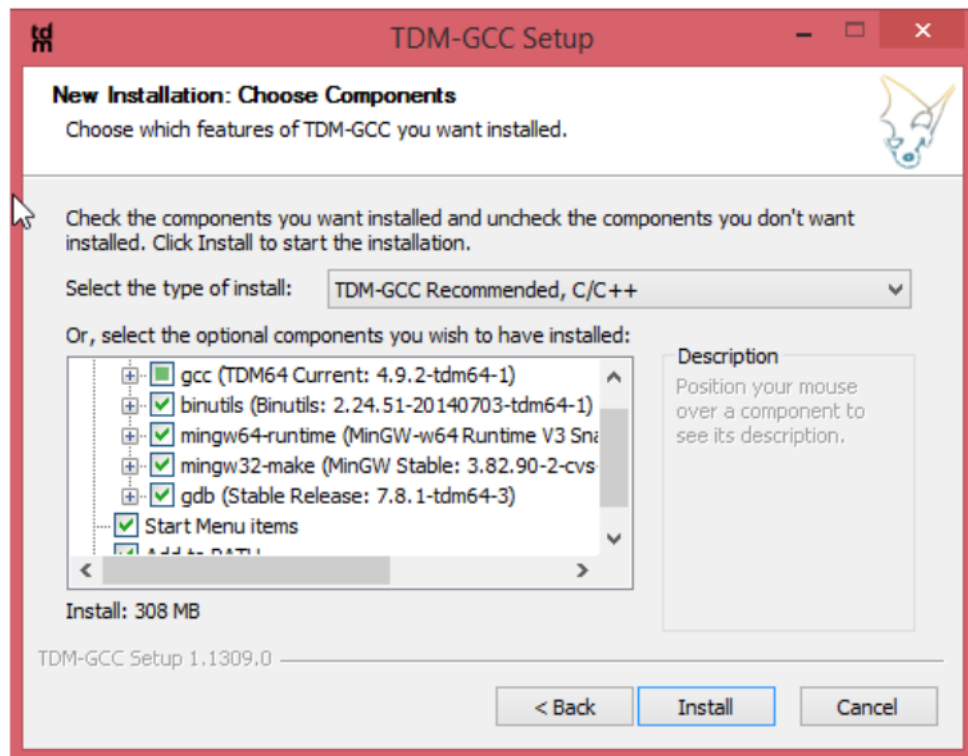
Seleccionamos *MinGW-w64/TDM64* y pulsamos siguiente.

9.1. EL COMPILADOR Y EL IDE. FUENTE DE MUCHOS PROBLEMAS59



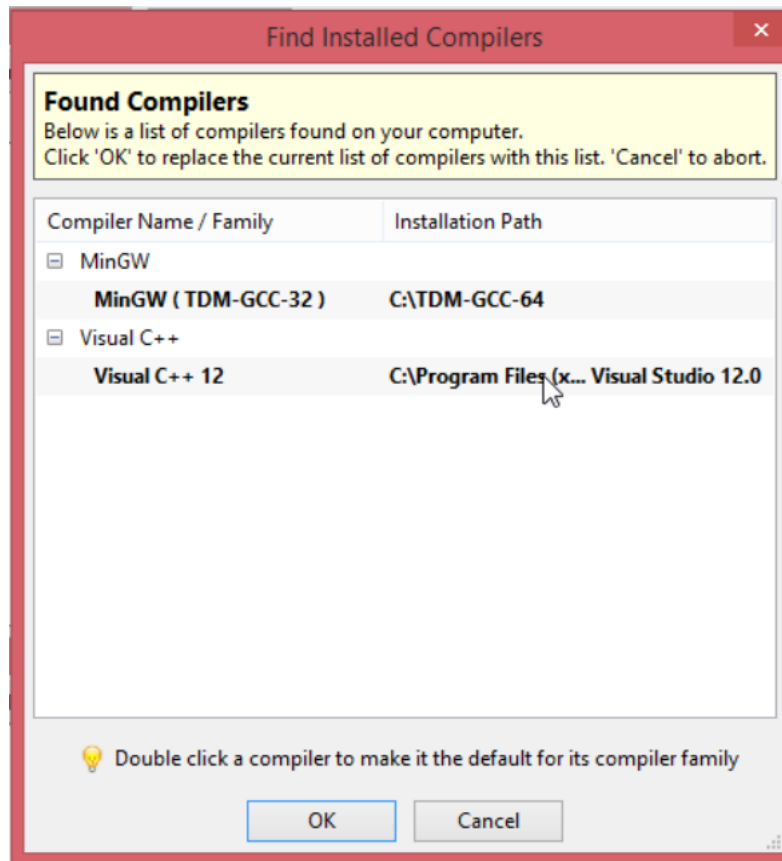
el directorio de instalación, lo dejare por defecto en mi caso.

Elegimos



En esta ventana, elegimos en el desplegable la opción TDM-GCC Recommended, C/C++. Nos aseguramos de que esten todas las casillas marcadas, sobretodo PATH, ya que ésta será la que más dolores de cabeza acabe dándote. Finalmente, pulsamos en Install y dejamos que acabe la instalación. Volvemos a abrir CodeLite y nos aparecera de nuevo la opción de elegir compilador

9.1. EL COMPILADOR Y EL IDE. FUENTE DE MUCHOS PROBLEMAS61



Por

supuesto, elegimos *MinGW* y pulsamos *OK*.

Una vez hecho esto ¡**FELICIDADES!** Acabas de conseguir instalar exitosamente tu entorno de desarrollo para el curso, ya te has quitado de encima la parte más difícil. A partir de ahora, todo irá viento en popa, ya verás.

9.1.2. Instalación del compilador en Linux.

Ahora, pasamos a la instalación en Linux que es más sencilla. Esta escrito pensando en que utilices una distribución basada en *Debian*, como puede ser *Ubuntu*, *Kubuntu*, *Xubuntu* o *Linux Mint*. A continuación se muestran enlaces a las webs de cada distribución, en este caso usaremos *Debian*

- Debian: <https://www.debian.org>
- Ubuntu: <http://www.ubuntu.com>
- Kubuntu: <http://www.kubuntu.org>
- Xubuntu: <http://xubuntu.org>

Afortunadamente, en distribuciones basadas en *Debian* tenemos por lo general incluido (Al menos, en las mencionadas anteriormente), el editor, por lo que solamente nos sería necesario instalar el compilador. De todas maneras, por si preferís utilizar también *CodeLite*, os dejo lo que tenéis que hacer, pero primero pasamos a lo sencillo.

Abrimos un terminal y en el escribimos:

```
$ sudo apt-get install clang
```

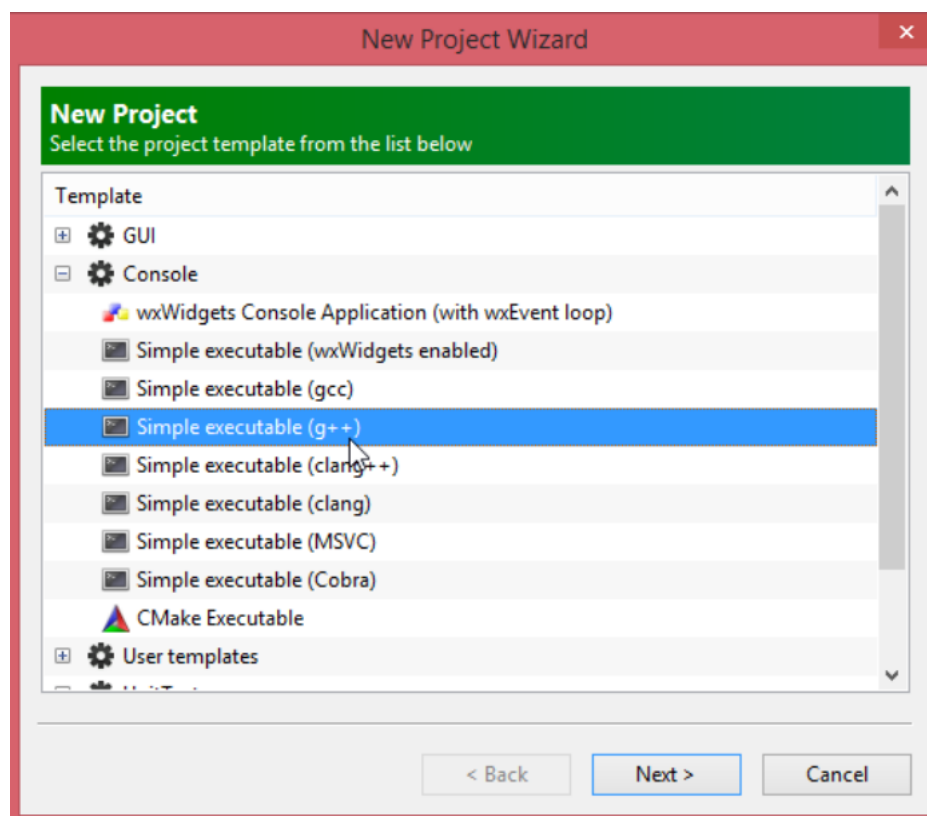
Dependerá de cómo hallas abierto el terminal, os pedirá una contraseña y confirmación para instalarse. Una vez instalado, escribís: `$ mkdir Curso && cd Curso $ gedit holamundo.cpp`

- `mkdir`: Crea un directorio cuyo nombre sea la palabra anterior.
- `cd`: Accede al directorio "Curso".
- `gedit nombredelarchivo.cpp`: Abrirá el editor *gedit*, y te permitirá editar el archivo *nombredelarchivo.cpp* escogido anteriormente.

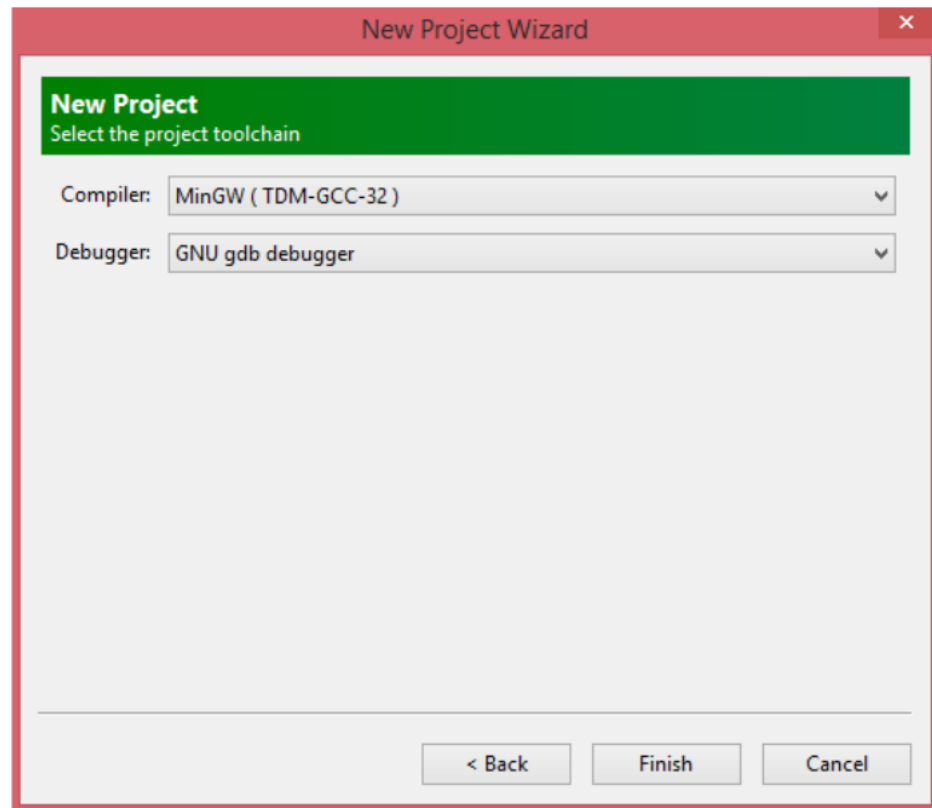
Una vez hecho esto, se nos abrirá *gedit* y escribiremos nuestro código. Al acabar, o al querer compilar el programa lo guardamos y escribimos en el terminal: `$ c++ holamundo.cpp $./a.out` A continuación, haremos nuestro primer programa.

9.2. Nuestro primer programa, ¡Hola mundo!

Como es costumbre, siempre que se empieza a programar o se aprende en un lenguaje nuevo lo primero que se enseña es el Hola Mundo, que no es más que hacer un programita básico que muestre en la *consola* un mensaje, puede ser el que queráis. Una vez que ya sabéis como crear y editar un programa en *Linux*, no os hará falta nada más para seguir este curso. Abrimos *CodeLite*, pulsamos: *File > New > New Workspace > C++ Workspace* En *Workspace Name* seleccionamos el directorio donde queremos trabajar. Una vez hecho esto, seguimos de nuevo: *File > New > New Project*

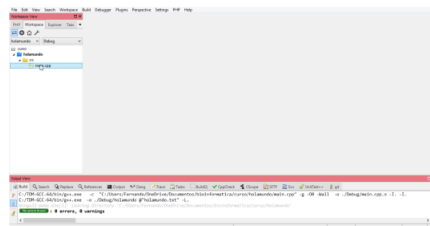


Seleccionamos *Simple Executable (g++)* y pulsamos *next*.



De nuevo pulsamos *next*.

Finalmente, tendremos una versión del entorno como la siguiente:



A continuación, borramos todo lo que aparece en el archivo *main.cpp* y escribimos:

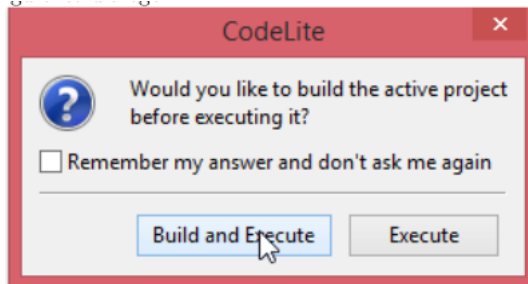

```

#include <iostream>

int main()
{
    std::cout << "Hola mundo" << std::endl;
    std::cin.get();
    return 0;
}

```

No te preocupes si todo esto te suena a chino, se explicara todo paso a paso. Para comprobar que el programa funciona, vamos a la pestaña *Run* y al pulsarla tendremos el siguiente dialogo:



Pulsamos la casilla y le damos a *Build and Execute*

Tras esto, se abrirá una consola y mostrare el mensaje *Hola mundo*, puedes cambiarlo por cualquier mensaje, prueba y juega todo lo que quieras con los ejemplos. Seguidamente, se explica el programa linea a linea:

- **#include** permite incluir una *libreria*. **Libreria:** Archivo que contiene un grupo de funciones predefinidas, que nos permitirá utilizar en nuestro programa sin saber si quiera cómo estan hechas. Existen diferentes librerias, utilizaremos las básicas como *iostream* (Entrada y salida), *fstream* (Control de archivos), *cmath* (Funciones matemáticas)... Hay miles de librerías para casi cualquier cosa, lo que facilitará mucho el trabajo en proyectos grandes.
- **cout >>:** Imprime un mensaje en pantalla, que esté entre comillas.
- **endl:** Introduce una nueva linea, es necesario usarlo con *cout* y previo a él debe llevar el separador <<.
- **cin.get():** Es un arreglo que hemos hecho para parar la ejecución del programa. Para usuarios de *Linux* esto no sera necesario. Básicamente sirve para que al ejecutarlo no se cierre la *consola* al acabar.
- **return 0:** Indica que todo ha acabado correctamente, esto es más a nivel interno.

Supongo que ahora os estaréis preguntando, ¿qué narices es eso de *std::*? Bien, como se explicó anteriormente, una libreria es un conjunto de muchas

funciones, entonces... ¿Qué pasaría si dos librerías diferentes tuvieran dos funciones que se llaman de la misma manera? ¿Cómo le decimos cual escoger? Para ello existe el *espacio de nombres*, pongamos un ejemplo práctico.

```
#include <archivos>
#include <libros>

int main()
{
    libros::leer();
    archivo::leer();
}
```

Como ves, tenemos dos funciones a leer, y de esta manera le indicamos al compilador cual coger. Esto ahora mismo no es importante, pero es conveniente conocerlo. A partir de ahora, nos ahorraremos escribir siempre *std::*: utilizando justo después de incluir las librerías, la coletilla de todos nuestros programas: *using namespace std*; Tras escribir esta línea, podremos omitir *siempre* escribir *std::*, aunque hemos de tener siempre en cuenta las consecuencias de esto. Por ello, una vez aplicado esto nuestro programa quedara de la siguiente manera:

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

int main() {
    cout << "Hola_mundo!";
    cin.get(); // cin.get() es opcional fuera de CodeLite
    return 0;
}
```

Compílalo y ejecútalo de nuevo, veras que funciona igual.

9.2.1. Comentarios

Vamos a ver de forma breve los comentarios. Mientras estamos haciendo nuestro programa, conforme se va haciendo más grande es más complicado luego entender qué se esta haciendo, es fácil perderse. Para ello, C++ permite el uso de comentarios. Siempre que se haga un comentario, esa parte de código será ignorada por el compilador, por lo que realmente *no* formara parte del programa. Los comentarios se pueden poner en una única línea, precedida por *//* o de diversas líneas */* */*. Pongamos un ejemplo:

```
// Esto es un comentario de una sola línea
/* Esto es un comentario
de varias líneas
y el compilador no lo vera nunca
*/
```

Capítulo 10

Cuestionarios

10.1. Lo basico

1) ¿Cual es el valor que devuelve el sistema cuando el programa se completa con éxito?

- a) -1
- b) 1
- c) 0
- d) Los programas no devuelven un valor de retorno

2) ¿Cual es la única función que todos los programas deben tener?

- a) start()
- b) system()
- c) main()
- d) program()

3) ¿Cual es la manera correcta de abrir y cerrar un bloque de código?

- a)
- b) <>
- c) BEGIN <codigo> END
- d)()

4) ¿Cual es el simbolo utilizado al final de la mayoría de lineas de C++?

- a) .

- b) ;

- c) :

- d) '

5) ¿Cual de los siguientes es un comentario correcto?

- a) */Comentario */

- b) ** Comentario **

- c) /* Comentario */

- d) {Comentario}

6) ¿Cual de los siguientes no es un tipo correcto de variable?

- a) float

- b) real

- c) int

- d) double

7) ¿Cual de los siguientes es el operador correcto para comparar dos variables?

- a) :=

- b) =

- c) equal

- d) ==

Respuestas:

1C - 2C - 3A - 4B - 5C - 6B - 7D

10.2. Estructuras de control

1) ¿Cual de los siguientes es cierto?

- a) 1

- b) 66

- c) -1

- d) Todos los anteriores

2) ¿Cual de los siguientes es el operador **booleano** para el **and** lógico?

- a) &
- b) &&
- c) —
- d) &—

3) Evalua `!(1 && !(0 — 1))` ó NOT (1 AND NOT(0 OR 1))

- a) Verdadero
- b) Falso
- c) No se puede evaluar

4) ¿Cual de las siguientes expresiones muestra la sintaxis correcta para un bloque **if**?

- a) if expresión
- b) if expresión
- c) if (expresión)
- d) expresión if

Respuestas:

1D - 2B - 3A - 4C

10.3. Estructuras iterativas

1) ¿Cual es el valor final de x cuando ejecutamos el código `for(int x = 0; x < 10; x++)`?

- a) 10
- b) 9
- c) 0
- d) 1

2) ¿Cuándo se ejecuta `while(x < 100)`?

- a) Cuando x es menor que 100
- b) Cuando x es mayor que 100
- c) Cuando x es igual a 100
- d) Aleatoriamente

3) ¿Cual de las siguientes no es una estructura iterativa?

- a) for
- b) do-while
- c) while
- d) repeat until

4) ¿Cuántas veces se garantiza que se cumpla un bucle *do-while*?

- a) 0
- b) Infinitamente
- c) 1
- d) Variable

Respuestas:

1A - 2A - 3D - 4C

10.4. Funciones

1) ¿Cual de los siguientes no es un prototipo correcto?

- a) `int funct(char x, char y);`
- b) `double funct(char x)`
- c) `void funct();`
- d) `char x();`

2) ¿Cual es el tipo de retorno de la función: *int func(char x, float v, double t)?*

- a) char
- b) int
- c) float
- d) double

3) ¿Cual de los siguientes es una llamada válida a una función?

- a) `func;`
- b) `func x,y;`
- c) `func();`

- d) int func()
- 4) Cual de los siguientes es una función completa?
- a) int func();
 - b) int func(int x) return x=x+1;
 - c) void func(int) cout << Hola;
 - d) void func(x) cout << Hola;

Respuestas:

1B - 2B - 3C - 4B

10.5. Switch-case

- 1) ¿Cual de los siguientes sigue a la sentencia **case**?
- a) :
 - b) ;
 - c) -
 - d) Nueva linea
- 2) ¿Que se necesita para evitar ejecutar el código del siguiente case?
- a) end;
 - b) break;
 - c) Stop;
 - d) ,
- 3) ¿Qué palabra clave cubre las opciones no manejadas?
- a) all
 - b) unhandled
 - c) default
 - d) other
- 4) ¿Cual es el resultado del siguiente codigo?

```
int x=0;

switch(x)
{
    case 1: cout<<"Uno";

    case 0: cout<<"Cero";

    case 2: cout<<"Hola_Mundo";

}
```

- a) Uno
- b) Cero
- c) Hola Mundo
- d) CeroHolaMundo

Respuestas:

1A - 2B - 3C - 4D

10.6. Arrays

1) ¿Cual de los siguientes declara correctamente un *array*?

- a) int miArray[10];
- b) int myArray;
- c) myarray(10);
- d) array miArray[10]

2) ¿Cual es el indice del último elemento de un *array* de 29 elementos?

- a) 29
- b) 28
- c) 0
- d) El programador lo elige

3) ¿Cual de los siguientes es un array **bidimensional**?

- a) array miArray[10][10];
- b) int myArray[10][10];
- c) int miArray[20,20];
- d) char array[20];

4) ¿Cual de los siguientes accede corectamente al séptimo elemento de un *array* de 100 posiciones almacenado en la variable *var*?

- a) var[6];
- b) var[7];
- c) var(6);
- d) var;

5) ¿Cual de los siguientes devuelve la dirección de memoria del primer elemento del array *var*, que posee 100 elementos?

- a) var[0];
- b) var;
- c) &var
- d) var[1]

Respuestas

A1 - 2B - 3B - 4A - 5B

10.7. Estructuras

1) ¿Cual de las siguientes accede a una variable *var* en la estructura *b*?

- a) b->var;
- b) b.var;
- c) b-var;
- d) b>var;

2) ¿Cual de los siguientes accede a la variable en un puntero a una estructura, **b*?

- a) b->var;
- b) b.var;

- c) b-var;
 - d) b>var:
- 3) ¿Cual de los siguientes es una estructura correctamente definida?
- a) struct {int a;}
 - b) struct estructura int a;
 - c) struct estructura int a;
 - d) struct estructura int a;
- 4) ¿Cual de los siguientes declara una variable de tipo estructura de *var* ?
- a) struct var
 - b) var foo;
 - c) var
 - d) int var

Respuestas:

1B - 2A - 3D - 4B

10.8. Punteros

- 1) ¿Cual de los siguientes es una declaración correcta de un puntero?
- a) int x;
 - b) int &x
 - c) ptr x;
 - d) int *x;
- 2) ¿Cual de los siguientes da la dirección de memoria de un entero de variable **a**?
- a) *a;
 - b) a;
 - c) &a;
 - d) direccion(a);
- 3) ¿Cual de los siguientes da la dirección de memoria de un puntero a?

- a) a;
- b) *a;
- c) &a
- d) direccion(a)

4) ¿Cual de los siguientes da el valor almacenado en la dirección apuntada por el puntero **a**?

- a) a;
- b) val(a);
- c) *a;
- d) &a;

5) ¿Cual de los siguientes es la palabra clave correcta para asignar memoria?

- a) new
- b) malloc
- c) create
- d) value

6) ¿Cual de las siguientes es la palabra clave correcta para desasignar memoria?

- a) free
- b) delete
- c) clear
- d) remove

Respuestas

1D - 2C - 3A - 4C - 5A - 6B