UNIVERSIDAD DEL VALLE, FACULTAD DE CIENCIAS, DEPARTAMENTO DE FISICA

CAPÍTULO I

Introducción y conceptos fundamentales.

Objetivo

Estudiar los pasos necesarios para generar un programa en lenguaje C/C++.

1. Introducción

Los ejemplos que ilustran cada taller se puedan compilar con cualquier versión de compilador, sin embargo, por ejemplo puede usar el compilador MinGW, (Minimalist GNU for Windows), que es una versión para Windows del compilador GCC para Unix y Linux, y que está adaptado para crear programas en Windows. Es decir, los programas que se ajusten al estándar de C++ deberían funcionar con este compilador tanto en Windows como en Linux.

GCC es un compilador integrado del proyecto GNU para C, C++, Objective C y Fortran; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de correr. La sigla GCC significa "GNU Compiler Collection". Originalmente significaba "GNU C Compiler"; todavía se usa GCC para designar una compilación en C. G++ refiere a una compilación en C++.

Puede usar algún IDE (Entorno de Desarrollo Integrado), como Dev-C++ de Bloodshed o Code::Blocks para crear programas en modo consola.

Los programas de Windows tienen dos modos de cara al usuario:

- El modo consola simula el funcionamiento de una ventana MS-DOS, trabaja en modo de texto, es decir, la ventana es una especie de tabla en la que cada casilla sólo puede contener un carácter. El modo consola de Windows no permite usar gráficos de alta resolución. Pero esto no es una gran pérdida, se verán otras maneras de hacer gráficos más adelante, e incluso existen librerías externas no estándar para tal fin.
- El otro modo es el GUI, o Interfaz Gráfico de Usuario. Es el modo tradicional de los programas de Windows, con ventanas, menús, iconos, etc., éste lo tiene en el entorno Dev-C++.

Para otros entornos como Linux, Unix o Mac, no sirve el entorno Dev-C++, ya que está diseñado especialmente para Windows. Pero esto no es un problema serio, todos los sistemas operativos disponen de compiladores de C++ que soportan la norma ANSI, sólo se mencionan Dev-C++ y Windows porque es el entorno más común.
En el presente curso no se saldrá del ANSI, es decir del C++ estándar, así que no es probable que surjan problemas con los compiladores.

1.2 Vocabulario Técnico

1.2.1. Archivo fuente y programa o código fuente

Los programas C/C++ se escriben con la ayuda de un editor de textos del mismo modo que cualquier texto corriente. Los archivos que contiene programas en C/C++ en forma de texto se conocen como archivos fuente, y el texto del programa que contiene se conoce como programa o código fuente. Nosotros **siempre** escribiremos programas fuente y los guardaremos en archivos fuente.

1.2.2. Interpretes y compiladores

Tanto C como C++ son lenguajes compilados, y no interpretados. Esta diferencia es muy importante, ya que afecta mucho a muchos aspectos relacionados con la ejecución del programa.

En un **lenguaje interpretado**, el programa está escrito en forma de texto, es el propio programa fuente. Este programa fuente es procesado por un programa externo, el intérprete, que traduce el programa, instrucción a instrucción, al tiempo que lo ejecuta. En los lenguajes interpretados no existen programas ejecutables directamente por el computador. El intérprete traduce, en tiempo real, cada línea del programa fuente, cada vez que se quiere ejecutar el programa.

El los **lenguajes compilados** el proceso de traducción sólo se hace una vez. El programa compilador toma como entrada el código fuente del programa, y da como salida un archivo que puede ser ejecutado por el computador directamente. Una vez compilado, el programa ejecutable es autónomo, y ya no es necesario disponer del programa original ni del compilador para ejecutarlo.

Cada opción tiene sus ventajas e inconvenientes, y algunas características que son consideradas una ventaja, pueden ser un inconveniente en ciertas circunstancias, y viceversa.

- Los lenguajes interpretados son fácilmente modificables, ya que necesitamos tener el el código fuente disponible en el computador. En los compilados, estos archivos no son necesarios, una vez compilados.
- Los lenguajes interpretados necesitan un programa externo, llamado intérprete o a veces máquina virtual, o framework. Este programa actúa como intermediario entre el fuente y el sistema operativo. En los compilados ese papel lo desempeña el compilador, pero al contrario que con el intérprete, una vez ha hecho su trabajo, no es necesario que esté presente para ejecutar el programa.
- Estas dos características, lógicamente, hacen que los programas compilados requieran menos espacio de memoria que los interpretados (si contamos el espacio usado por el intérprete), y en general, los compilados son más rápidos, ya que sólo se compilan una vez, y el tiempo dedicado a esa tarea no se suma al de ejecución.

Entre los lenguajes interpretados están: BASIC, MatLab, JavaScript, PHP, Perl, Phyton, Tcl. Muchos lenguajes de script, etc.

Entre los lenguajes compilados están: C, C++, Fortran, Delphi, Pascal, Visual Basic.

1.2.3. Archivos objeto, código objeto y compiladores

Como hemos dicho antes, en los lenguajes compilados, los programas fuente no pueden ejecutarse. Son archivos de texto, pensados para que los comprendan los seres humanos, pero incomprensibles para los computadores.

Para conseguir un programa ejecutable hay que seguir algunos pasos. El primero es compilar o traducir el programa fuente a su código objeto equivalente. Este es el trabajo que hacen los compiladores de C y C++. Consiste en obtener un archivo equivalente a nuestro programa fuente comprensible para el computador, este archivo se conoce como archivo objeto, y su contenido como código objeto.

Los compiladores son programas traductores, que leen un archivo de texto que contiene el programa fuente y generan un archivo que contiene el código objeto.

El código objeto no suele tener ningún significado para los seres humanos. Además es diferente para cada computador y para cada sistema operativo. Por lo tanto existen diferentes compiladores para diferentes sistemas operativos y para cada tipo de computador. Sin embargo en general, si el programa fuente es escrito usando un lenguaje estandar y compilado usando un compilador estandar hoy en día el archivo objeto resultante se puede ejecutar en todo computador que tenga C/C++ estandar.

1.2.4. Librerías o librerías

Junto con los compiladores de C y C++, se incluyen ciertos archivos llamados librerías. Éstas contienen el código objeto de muchos programas que permiten hacer cosas comunes, como leer el teclado, escribir en la pantalla, manejar números, realizar funciones matemáticas, etc. Las librerías están clasificadas por el tipo de trabajos que hacen, las hay de entrada y salida, matemáticas, de manejo de memoria, de manejo de textos, etc.

Hay un conjunto de librerías muy especiales, que se incluyen con todos los compiladores de C/C++. Son las librerías ANSI **o estándar**. Pero también las hay no estándar, y dentro de estas las hay públicas y comerciales. ***En este curso sólo usaremos librerías ANSI o estándar***.

1.2.**5. Archivo ejecutables y enlazadores**

Cuando obtenemos el archivo objeto, aún no hemos terminado el proceso. El archivo objeto, a pesar de ser comprensible para el computador, no puede ser ejecutado. Hay varias razones para eso:

1.2.5.1 Nuestros programas usaran, en general, funciones que estarán incluidas en librerías externas, ya sean ANSI o no. Es necesario combinar nuestro archivo objeto con esas librerías para obtener un ejecutable.
1.2.5.2 Muy a menudo, nuestros programas estarán compuestos por varios archivos fuente, y de cada uno de ellos se obtendrá un archivo objeto. Es necesario unir todos los archivos objeto, más las librerías en un único archivo ejecutable.
1.2.5.3 Hay que dar ciertas instrucciones al computador para que cargue en memoria el programa y los datos, y para que organice la memoria de modo que se disponga de una pila de tamaño adecuado, etc. La pila es una zona de memoria que se usa para que el programa intercambie datos con otros

programas o con otras partes del propio programa. Veremos esto con más detalle durante el curso.
1.2.5.4 No siempre obtendremos un archivo ejecutable para el código que escribimos, a veces querremos crear archivos de librería, y en ese caso el proceso será diferente.
Existe un programa que hace todas estas cosas, se trata del "linker", o enlazador. El enlazador toma todos los archivos objeto que componen nuestro programa, los combina con los archivos de librería que sean necesarios y crea un archivo ejecutable.

Una vez terminada la fase de enlazado, ya podremos ejecutar nuestro programa.

1.2.6. Errores

Por supuesto, somos humanos, y por lo tanto nos equivocamos. Los errores de programación pueden clasificarse en varios tipos, dependiendo de la fase en que se presenten.

1.2.6.1 Errores de sintaxis: son errores en el programa o código fuente. Pueden deberse a palabras reservadas mal escritas, expresiones erróneas o incompletas, variables que no existen, etc. Los errores de sintaxis se detectan en la fase de compilación. El compilador, además de generar el código objeto, nos dará una lista de errores de sintaxis. De hecho nos dará sólo una cosa o la otra, ya que si hay errores no es posible generar un código objeto.
1.2.6.2 Avisos (Warnings): además de errores, el compilador puede dar también avisos. Los avisos son errores, pero no lo suficientemente graves como para impedir la generación del código objeto. No obstante, es importante corregir estos errores, ya que ante un aviso el compilador tiene que tomar decisiones, y estas no tienen por qué coincidir con lo que nosotros pretendemos hacer, ya que se basan en las directivas que los creadores del compilador decidieron durante la creación del compilador.
1.2.6.3 Errores de enlazado: el programa enlazador también puede encontrar errores. Normalmente se refieren a funciones que no están definidas en ninguno de los archivos objetos ni en las librerías. Puede que hayamos olvidado incluir alguna librería, o algún archivo objeto, o puede que hayamos olvidado definir alguna función o variable, o lo hayamos hecho mal.
1.2.6.4 Errores de ejecución: incluso después de obtener un archivo ejecutable, es posible que se produzcan errores. En el caso de los errores de ejecución normalmente no obtendremos mensajes de error, sino que simplemente el programa terminará bruscamente. Estos errores son más difíciles de detectar y corregir. Existen programas auxiliares para buscar estos errores, son los llamados depuradores (*debuggers*). Estos programas permiten detener la ejecución de nuestros programas, inspeccionar variables y ejecutar nuestro programa paso a paso (instrucción a instrucción). Esto resulta útil para detectar excepciones, errores sutiles, y fallos que se presentan dependiendo de circunstancias distintas.
1.2.6.5 Errores de diseño: finalmente los errores más difíciles de corregir y prevenir. Si nos hemos equivocado al diseñar nuestro algoritmo, no habrá ningún programa que nos pueda ayudar a corregir los nuestros. ***Contra estos errores sólo cabe practicar y pensar***.

1.3. Propósito de C y C++

¿Qué clase de programas y aplicaciones se pueden crear usando C y C++?

La respuesta es muy sencilla: ***TODOS***.

C/C++ son lenguajes de programación de propósito general. Todo puede programarse con ellos, desde sistemas operativos y compiladores hasta aplicaciones de bases de datos y procesadores de texto, pasando por juegos, aplicaciones a medida, etc.

Oirás y leerás mucho sobre este tema. Sobre todo diciendo que estos lenguajes son complicados y que requieren páginas y páginas de código para hacer cosas que con otros lenguajes se hacen con pocas líneas. Esto es una verdad a medias. Es cierto que un listado completo de un programa en C o C++ para gestión de bases de datos (por poner un ejemplo) puede requerir varios miles de líneas de código, y que su equivalente en Visual Basic sólo requiere unos pocos cientos. Pero detrás de cada línea de estos compiladores de alto nivel hay cientos de líneas de código en C/C++, la mayor parte de estos compiladores están respaldados por enormes librerías escritas en C/C++. Nada te impide a ti, como programador, usar librerías, e incluso crear las tuyas propias.

Una de las propiedades de C y C++ es la reutilización del código en forma de librerías de usuario. Después de un tiempo trabajando, todos los programadores desarrollan sus propias librerías para aquellas cosas que hacen frecuentemente. Y además, raramente piensan en ello, se limitan a usarlas.

Además, los programas escritos en C o C++ tienen otras ventajas sobre el resto. Con la excepción del ensamblador, generan los programas más compactos y rápidos. El código es transportable, es decir, un programa ANSI en C o C++ podrá ejecutarse en cualquier máquina y bajo cualquier sistema operativo. Y si es necesario, proporcionan un acceso a bajo nivel de hardware sólo igualado por el ensamblador.
Otra ventaja importante es que C tiene más de 30 años de vida, y C++ casi 20 y no parece que su uso se debilite demasiado. No se trata de un lenguaje de moda, y probablemente a ambos les quede aún mucha vida por delante. Sólo hay que pensar que sistemas operativos como Linux, Unix o incluso Windows se escriben casi por completo en C/C++.

Por último, existen varios compiladores de C y C++ gratuitos, o bajo la norma GNU, así como cientos de librerías de todo propósito y miles de programadores en todo el mundo, muchos de ellos dispuestos a compartir su experiencia y conocimientos.

1.4. Proceso de creación de un programa en C/C++.

1.4.1. Edición

El proceso de desarrollo empieza con la escritura o edición del código fuente en un archivo fuente. La edición se hace mediante un sencillo editor de texto.
Son habituales las siguientes extensiones o sufijos de los nombres de archivo:

.c	fuente en C
.C .cc .cpp .c++ .cp .cxx	fuente en C++; se recomienda .cpp
.m	fuente en Objective-C
.i	C preprocesado
.ii	C++ preprocesado
.s	fuente en lenguaje ensamblador
.o	código objeto
.h	archivo para preprocesador (encabezados), no suele figurar en la linea de comando de gcc

1.4.2. Compilación

La sintaxis es:

```
gcc [ opción | archivo ] ...
g++ [ opción | archivo ] ...
```

Las opciones van precedidas de un guión, como es habitual en UNIX, pero las opciones en sí pueden tener varias letras; no pueden agruparse varias opciones tras un mismo guión. Algunas opciones requieren después un nombre de archivo o directorio, otras no. Finalmente, pueden darse varios nombres de archivo a incluir en el proceso de compilación.

1.4.3. Ejecución

Corregido el programa y verificada su correcta operación, se carga el fichero con el ejecutable, producto del proceso de compilación en memoria y se ejecuta el programa.

1.5. Entornos de desarrollo en Windows y en Linux.

El estudiante tiene a su disposición dos opciones para el desarrollo de programas en C, bajo Windows y bajo Linux, ambas conformes con la política de Software Libre de la Universidad del Valle.

1.5.1 Windows

Bajo este sistema operativo son muy populares los "ambientes de desarrollo", IDE (Integrated Development Environment). El IDE es un programa que reúne el editor, el compilador y herramientas de depuración. Se recomienda el DEV–C++ (http://www.bloodshed.net), un IDE gratuito y fácil de utilizar. Quien decida utilizar un IDE como entorno de desarrollo, se encontrará con la facilidad de la integración de las herramientas necesarias para la producción de un programa, de modo tal que con la presión de una tecla se llevarán a cabo todos los pasos de compilación y ejecución del programa desarrollado. Debe visitar la página mencionada, bajar e instalar el compilador; si tiene algún problema con la consecución del software, o con su instalación, consulte con el docente o el monitor de la sala de computo.

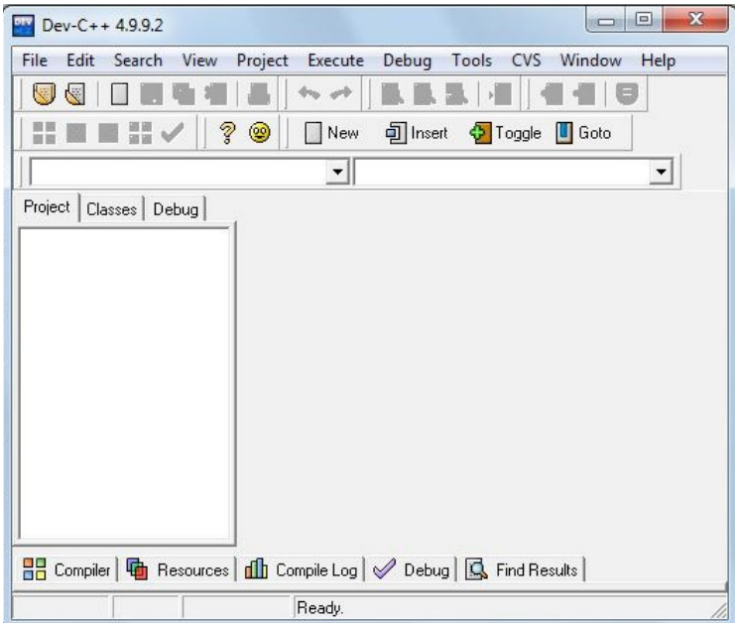

Fig. 1: Devian C++: Ventana del IDE.

Para nuestros propósitos la configuración en defecto del compilador es suficiente, para usarlo corra el ejecutable "devcpp.exe" que abrirá la ventana del IDE mostrada en la Fig. 1.

Para abrir el editor y editar un archivo nuevo escoja la opción "File>New>Source File" ; para cargar un archivo previamente editado, como es el caso nuestro, escoja la opción "File>Open Project or File" y navegue en el árbol de directorios que se le abre hasta la carpeta donde haya puesto el archivo que quiera editar (por ejemplo

hola.cpp) y ábralo; ahora tiene el archivo fuente listo para edición.

Compilación

Una vez editado el programa, compílelo mediante la opción "Execute>Compile". Observe que el proceso de compilación produce el archivo ejecutable "hola.exe".

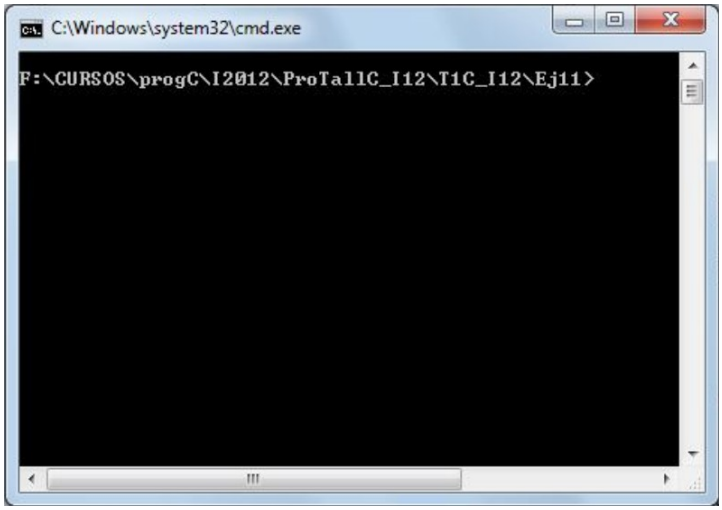

Fig. 2: Ventana de comando

Ejecución del programa.

El programa creado es lo que se conoce como "aplicación de consola", es un comando que se ejecuta desde una ventana de comando como la de la Fig. 2; abra una invocando el comando "cmd" desde el menú de inicio de WINDOWS, cambie de directorio hasta la carpeta de trabajo y ejecute el programa.

1.5.2 Linux

Este sistema operativo es el usado en la sala de cómputo de la Facultad. Usted deberá tener algunas nociones fundamentales del sistema operativo, específicamente debe saber como:

- Se crea una carpeta de trabajo (con `mkdir`). Usted debe crear una y nombrarla con su apellido. Evite los nombres largos, mezclar mayúsculas y minúsculas (produce confusiones) y usar espacios o caracteres distintos a los del alfabeto. En esta carpeta guardará su trabajo.
- Empaquetar y comprimir una carpeta, utilice por ejemplo
  `tar -vcf nombre_archivo.tar nombre_carpeta_a_empaquetar`
- Desempaquetar y descomprimir una carpeta, utilice por ejemplo
  `tar -vxf mi_archivo.tar`
- Ubicar su carpeta en el disco duro de la máquina; debe saber cambiar de carpeta (cambiar de directorio, comando `cd`).
- Copiar de una lado a otro con `cp` si es un archivo o con `cp -r` si es una carpeta.
- Listar el contenido de una carpeta (list, comando `ls`).
- Usar un editor de texto y la forma de invocarlo (si usa el editor gedit utilice `gedit mi_archivo.cpp&` o si usa el editor emacs utilice `emacs mi_archivo&`).
- Ubicar el camino de direcciones de un cierto archivo con `pwd`.

- Abrir una consola o terminal (konsole o terminal) y aprender a invocar el compilador de C/C++, como se explicará a continuación.

Bajo este sistema operativo existen también dos maneras para el usuario.

1.5.2.1 Los programas requeridos, el editor de texto y el compilador, son independientes.

El programador debe utilizar un editor para levantar el texto del código fuente y después invocar el compilador de C/C++. Para producir un ejecutable con fuente de un solo archivo:

1.5.2.1.1 Ejemplos para compilar

1. `$ gcc hola.c`

compila el programa en C hola.c, genera un archivo ejecutable a.out.

2. `$ gcc hola.cpp`

compila el programa en C++ hola.cpp, genera un archivo ejecutable a.out.

3. `$ gcc -o hola hola.c`

compila el programa en C hola.c, genera un archivo ejecutable hola.

4. `$ g++ -o hola hola.cpp`

compila el programa en C++ hola.cpp, genera un archivo ejecutable hola.

5. `$ g++ -o ~/bin/hola hola.cpp`

genera el ejecutable hola en el subdirectorio bin del directorio propio del usuario.

6. `$ g++ -L/lib -L/usr/lib hola.cpp`

indica dos directorios donde han de buscarse librerías. La opción -L debe repetirse para cada directorio de búsqueda de librerías.

7. `$ g++ -I/usr/include hola.cpp`

indica un directorio para buscar archivos de encabezado (de extensión .h).

1.5.2.1.2 Ejemplo para Ejecutar

Los archivos objeto o ejecutables de los ejemplos para compilar:

1. y 2. se ejecutan con: `./a.out`

3. y 4. se ejecutan con: `./hola`

5. se ejecuta con: `./~/bin/hola`

6. y 7. se ejecutan con: `./a.out`

1.5.2.2 Usando un IDE (Entorno de Desarrollo Integrado)

a. Puede aprender a utilizar NetBeans o Eclipse, nosotros veremos lo básico de ellos.

b. El gedit en la imagen tiene una pequeña consola abajo, gedit como otros editores en linux, permiten el uso de complementos así que para activar ese complemento lo instalamos primero su -c 'yum install gedit-plugins'

Una vez hecho esto, abrimos gedit vamos a Editar/Preferencias y ahí la pestaña Complementos, ahi buscamos y activamos Terminal Empotrado. Después en Ver/Barra inferior y de esta manera no necesita tener una terminal abierta, basta con el gedit abierto.

1.6. Estructura del programa en C/C++

Código fuente 1:

```
1 // my first program in C++
2 #include <iostream>
3
4 int main()
5 {
6   std::cout << "Hello World!" << std::endl ;
7 }
```

Salida:

```
Hello World!
```

El pánel de la izquierda muestra el código en C++. El de la derecha muestra el resultado cuando el programa es ejecutado. Los números grises a la izquierda de los páneles son introducidos para realizar la discusión del programa, ellos no son parte del programa. Examinaremos el programa línea a línea.

Línea 1: `// my first program in C++`
El sígno de doble slash indica que el resto de la línea es un comentario escrito por el programador pero no afecta al programa. Los programadores deben usar explicaciones cortas u observaciones concernientes al código para hacerlo entendible para ellos y para otra persona que lea el programa.

Línea 2: `#include <iostream>`
Líneas que inician con un signo de hash (#) son directrices leídas e interpretadas por lo que se conoce como el preprocesador. Son líneas especiales interpretadas antes de la compilación del programa. En éste caso en particular, la directriz #include <iostream> indica al preprocesador incluir una de las librerías estandar de C++ conocida como el encabezado *iostream*, que nos permite realizar operaciones de entrada y salida, tal como escribir en la pantalla Hello World.

Línea 3: Una línea en blanco.
Líneas en blanco no afectan el progrma, son usadas para hacer el código más fácil de leer.
Línea 4: int main ()
Esta línea inicia la declaración de una función. Una función es un grupo de estamentos de código que tienen un nombre, en éste caso se les llama "`main`" que significa principal. La función será discutida en detalle en el capítulo de funciones. Pero en escencia, la definición de ésta función es introducida con la secuencia:
El tipo de variable (`int`), el nombre (`main`) y un par de paréntesis (()), opcionalmente dentro de dichos paréntisis se pueden escribir parámetros.

La función llamada main es una función especial en todos los progrmas de C++. Esta función es llamada cuando se corre el programa. La ejecución de todos los programas de C++ comienzan con la función main sin importar dónde este ubicada dentro del código, puede estar al inicio, en la mitad o al final.

Líneas 5 and 7: { y }
El corchete abierto ({) en la línea 5 indica el inicio de la definición de la función main y el corchete

cerrado (}) en la línea 7, indica su final. Todo lo que se encuentre entre los corchetes es el cuerpo de la función que define lo que sucede cuando main es llamada. Todas las funciones usan corchetes para indicar el inicio y el final.

Línea 6: `std::cout << "Hello World!" << std::endl;`
Esta línea es una declaración de C++. Una declaración es una expresión que puede producir un efecto. Las declaraciones son ejecutadas en el mismo orden en que aparecen dentro del cuerpo de la función. Ésta declaración tiene tres partes:
1) `std::cout`, define el dispositivo de salida de la librería estandar (**st**andar**d** **c**haracter **out**put), si no se ha especificado anteriormente un archivo de salida entonces la salida está dada en pantalla.
2) El operador de inserción (<<) indica que lo que sigue es insertado en `std::cout`.
3) La sentencia entre comillas (`"Hello world!"`) es el contenido que se inserta en la salida estandar (standard output).
4) El operador de inserción (<<) indica que lo que sigue es insertado después de la frase entre comillas.
5) `std::endl`, define el dispositivo de terminación y corte de línea de la librería estandar.
6) La declaración termina con un punto y coma (;) que marca el final de la declaración de la misma manera que un punto y aparte termina una frase en español. Todas las declaraciones en C++ deben terminar en punto y coma, éste es uno de los errores más comunes en C++.

Código fuente 2:

```
// my first program in C++
#include <iostream>
#include <string>

int main()
{
  std::cout << "Escribe tu nombre aqui";
  std::string nombre ;
  std::cin >> nombre ;
  std::cout << "Hola, " << nombre << ". En que año naciste?" << std::endl ;
  std::int nac ;
  std::cin >> nac ;
  std::cout << " Estamos en el 2014 , tu edad es " << 2014 - nac -1
<< " o " << 2014 - nac << "." << std::endl ;
}
```

Salida:

```
Escribe tu nombre aqui
Pepito
Hola, Pepito. En que año naciste?
1997
Estamos en el 2014 , tu edad es 16 o 17.
```

1.7 **Usando el namespace std**

Es mucho más cómodo poder usar sólo el cout en lugar de std::cout. Ambos nombran el mismo objeto: el primero usa el nombre sin clasificar (cout), mietras que el segundo lo clasifica directamente dentro del namespace std (de manera std::cout)

cout es parte de la librería estandar y todos los elementos de la librería estandar de C++ son declarados dentro de lo que es llamado un namespace: el namespace std.

Para referir los elementos en el namespace std, un programa debe clasificar cada elemento que usa de la librería (como lo hicimos colocando el prefijo std:: a todas las declaraciones), o introduciendo visibilidad a todos los componentes. La manera más típica de introducir dicha visibilidad es usando la declaración:

```
using namespace std;
```

La declaración anterior permite tener acceso a todos los elementos del namespace std de manera declasificada, es decir sin el prefijo std::
Escribamos nuestro último ejemplo haciendo uso desclasificado de los elementos de std:
Código fuente 3:

```
// my first program in C++
#include <iostream>
#include <string>
using namespace std;

int main()
{
  cout << "Escribe tu nombre aqui";
  string nombre ;
  cin >> nombre ;
  cout << "Hola, " << nombre << ". En que año naciste?" << endl ;
  int nac ;
  cin >> nac ;
  cout << " Estamos en el 2014 , tu edad es " << 2014 - nac -1
<< " o " << 2014 - nac << "." << endl ;
}
```

Salida:

```
Escribe tu nombre aqui
Pepito
Hola, Pepito. En que año naciste?
1997
Estamos en el 2014 , tu edad es 16 o 17.
```

Ambas maneras de acceder a los elementos del namespace std son válidas y producen el mismo resultado. El uso de namespaces mejoran la lectura de los códigos. En los sucesivo usaremos este namespace sin embargo note que el uso explicito de la clasificación de las librerías es la manera que disminuye errores y evita el choque entre nombres de declaraciones.
Los namespaces serán explicados en detalle más adelante en otro capítulo.

Hemos introducido en el encabezado #include <string>. Siempre que utilicemos variables de tipo `string` es conveniente añadir esta línea. Las variables de tipo `string` guardan cadenas de texto.
En el ejemplo se definen las variables sin inicializarlas a ningún valor, porque se leerán a continuación con `cin`. El comando `cin` funciona de modo parecido a `cout`, excepto que utilizamos >> en vez de << ya que `cin` pone datos en una variable, en vez de sacarlos.

Por lo tanto, `cout <<` espera recibir *valores* para mostrarlos (por ejemplo, 2014-nac-1, mientras que `cin >>` espera recibir *variables* donde guardar los datos leídos.

En este programa, `nac` es una variable. Una variable es un trozo de memoria donde el computador almacena datos. La línea (`int nac;`) pide que se reserve un trozo de memoria, llamado a partir de ahora `nac`, con suficiente capacidad para almacenar números enteros (`int`, del ingles integer), y que no está inicializada. De este punto en adelante, siempre que en el programa aparezca `nac`, se consulta el trozo de memoria correspondiente para saber qué entero contiene `nac`.

Código fuente 4:

```
# include < iostream >
using namespace std ;
int main () {
cout << " Elige un numero 'n '. " << endl ;
int n =74;
cout << " [ elijo el " << n << " ]" << endl ;
cout << " Doblalo ." << endl ;
n =2* n; //se puede escribir también n*=2;
cout << " [ me da " << n << " ]" << endl ;
cout << " Sumale 6. " << endl ;
n=n +6;  //se puede escribir también n+=6;
cout << " [ obtengo " << n << "] " << endl ;
cout << " Dividelo entre 2 y restale 3. " << endl ;
n=n /2 -3;
cout << " [ sorpresa ! obtengo el numero inicial , " << n << " ]. " << endl ;
}
```

Salida:

```
Elige un numero 'n '.
[ elijo el 74]
Doblalo .
[ me da 148]
Sumale 6.
[ obtengo 154]
Dividelo entre 2 y restale 3.
[ sorpresa ! obtengo el numero inicial , 74].
```

En éste ejemplo la línea `int n =74;` inicializa la variable entera `n` con el valor de 74. A lo largo del programa la variable n cambia de valor a medida que se le van haciendo asignaciones.
La línea `n =2* n; //se puede escribir también n*=2;` tiene dos partes. La primera parte es la que se ejecuta en el programa `n =2* n;` cambia el valor de la variable n, asignandole el valor de 2*n. La segunda parte de la línea `//se puede escribir también n*=2;` no se ejecuta ya que es

un comentario (precedido por `//`). Más adelante veremos las diferentes maneras que se pueden escribir las operaciones en C++.

1.8. Tipos de Variables.

Para poder escribir programas útiles, es necesario introducir el concepto de *variable*.

Ejercicio Mental: Imagine que le solicito recordar el número 5 y luego le pido memorizar el número 2 al mismo tiempo. Usted tiene almacenados dos valores diferentes en su memoria (5 y 2). Ahora, le solicito sumar 1 al primer número que le dije, usted debe retener en su memoria los números 6 (esto es 5+1) y 2. Por último le pido que reste los dos valores y obtenga 4 como resultado.

El ejercicio mental, es un ejemplo sencillo de lo que un computador puede hacer con dos variables. El mismo proceso puede ser expresado en C++ con las siguientes declaraciones:

```
1 a = 5;
2 b = 2;
3 a = a + 1;
4 result = a - b;
```

Definimos *variable* como una porción de memoria destinada para almacenar un valor.

Cada variable necesita un nombre que la identifica y que la distingue de otras. Por ejemplo, los nombres de variables `a`, `b`, and `result`. Puede ser usado cualquier nombre siempre y cuando son identificadores válidos en C++.

1.8.1. Identificadores

Un identificador **válido** es una secuencia de una o más letras, dígitos o guión bajo (_).
Además, los identificadores deben comenzar con una letra (o con guión bajo o con dos guiones bajos, pero éstos son reservado para palabras claves específicas del compilador o identificadores externos).

Identificadores **NO válidos:** espacios, signos de puntuación (. , ; :) y símbolos (! # $ % & / ?)
No pueden en ningún caso comenzar con un dígito.

C++ utiliza un número de palabras clave para identificar operaciones y descripciones de datos, así que los identificadores creados por el programador no pueden ser iguales a estas palabras claves. Las palabras claves estándar reservadas que el programador no puede ser como identificadores son:

```
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr,
```

`operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq`

Importante!!:
C++ es un lenguaje sensitivo. Lo que significa que un identificador escrito en mayúsculas no es equivalente a otro con el mismo nombre pero en minúsculas. En el ejemplo anterior, la variable `RESULT` no es la misma que la variable `result`, o que la variable `Result`. Estos son tres identificadores diferentes que identifican tres variables diferentes.

1.8.2. Tipos fundamentales de datos

En C sólo existen cinco tipos fundamentales que son: **void**, **char**, **int**, **float** y **double** y los tipos enumerados, **enum**. C++ añade un séptimo tipo, el **bool**, y el resto de los tipos son derivados de ellos. También existen ciertos modificadores, que permiten ajustar ligeramente ciertas propiedades de cada tipo; los modificadores pueden ser: **short**, **long**, **signed** y **unsigned**, y pueden combinarse algunos de ellos.

Los valores de las variables son almacenados en algún lugar no específicado en la memoria del computador en forma de ceros y unos. El programa no sabe la locación de la variable, éste sólo se refiere a ella por su nombre. Lo que el programa necesita es saber el tipo de dato que es almacenado en la variable. No es lo mismo almacenar un simple número entero que una letra o que número largo de punto flotante o real, así todos estén representados por unos y ceros, no son interpretados de la misma manera y sobre todo no ocupan la misma cantidad de memoria.

Los tipos fundamentales de datos son las unidades de almacenamiento básicas (que se encuentran en la mayoría de los sistemas). Pueden ser clasificados de la siguiente manera:

- **Tipo Caracteres** : Pueden representar un caracter simple como `'A'` o `'$'`. El tipo más básico es `char`, el cual es un caracter de 1 bit.
- **Tipo Número Entero:** Pueden almacenar el valor de un número completo, como `7` o `1024`. Existen en una variedad de tamaños y pueden ser *signed* o *unsigned*, depediendo de si pueden almacenar valores negativos o no.
- **Tipo Punto Flotante (o Real):** Pueden almacenar valores reales, como `3.14` o `0.01`, con diferentes niveles de precisión, dependiendo de cuál de los tres tipos de punto flotante se usa.
- **Tipo Booleano:** Es conocido en C++ como `bool`, y puede sólo representar dos estados `true` o `false`.

Aquí encontrarán una lista completa de los tipos fundamentales en C++:

Grupo	Nombre del Tipo de dato*	Notes sobre tamaño / precisión
Tipo caracter	**`char`**	Tiene exactamente 1 bit de tamaño/Por lo menos 8 bits.
	`char16_t`	No más pequeño que `char`/Por lo menos 16 bits.
	`char32_t`	No más pequeño que `char16_t`./Por lo menos 32 bits.
	`wchar_t`	Puede representar el conjunto de caracteres más largo que soporta C++
Tipo Entero (signed)	**`signed char`**	Mismo tamaño que `char`./ Por lo menos 8 bits.
	`signed` **`short`** *`int`*	No más pequeño que `char`./Por lo menos 16 bits.
	`signed` **`int`**	No más pequeño que `short`. /Por lo menos bits.
	`signed` **`long`** *`int`*	No más pequeño que `int`. /Por lo menos 32 bits.
	`signed` **`long long`** *`int`*	No más pequeño que `long`. /Por lo menos 64 bits.
Tipo Entero (unsigned)	**`unsigned char`**	(tiene los mismo tamaños que su homólogo signed)
	`unsigned short` *`int`*	
	`unsigned` *`int`*	
	`unsigned long` *`int`*	
	`unsigned long long` *`int`*	
Tipo Punto Flotante	**`float`**	Por lo menos 32 bits.
	`double`	Precisión no menor que `float`
	`long double`	Precisión no menor que `double`
Boolean type	**`bool`**	
Void type	**`void`**	No almacena
Null pointer	**`decltype(nullptr)`**	

*Los nombres de ciertos tipos de enteros puden ser abreviados sin sus componentes `signed` e `int`, sólo la parte que NO está en itálica se requiere para identificar el tipo de dato, la parte en itálica es opcional. Por ejemplo: *`signed`* `short` *`int`* puede ser abreviado como `signed short`, `short int`, o simplemente `short`; todos ellos identifican el mismo tipo de dato.

Dentro de cada uno de los grupos antes descritos, la diferencia entre los tipos dedatos es solamente su tamaño (cuánto ocupan ellos en memoria): El primer tipo de cada grupo es el más pequeño (ocupa menos espacio en memoria) y el último el más largo (ocupa más espacio en memoria), con cada tipo siendo como mínimo tan largo como el inmediantamente anterior. Los tipos dentro de cada grupo tienen las mismas propiedades.

Note que en la lista anterior únicamente `char` tiene un tamaño especifico, el cual es exactamente de un byte u 8 bits, de resto ningún tipo fundamental de dato tiene un tamaño específico estándard, sólo un mínimo tamaño a lo sumo.
Los tipos están expresandos en bits; mientras más bits un tipo tiene, mayor es la cantidad de valores que puede representar, pero al mismo tiempo, consume también más espacio en memoria:

Tamaño	Valores represntables	
8-bit	256	$= 2^8$
16-bit	65 536	$= 2^{16}$
32-bit	4 294 967 296	$= 2^{32}$
64-bit	18 446 744 073 709 551 616	$= 2^{64}$

Para ***datos tipo enteros***, tener más posibilidades para representarlos, significa que los rangos de los valores que representan son más grandes. Por ejemplo, un unsigned integer de 16-bits puede ser representar 65536 de valores diferentes en un rango de 0 a 65535, mientras que su homólogo signed será capaz de representar en el mejor de los casos valores entre -32768 y 32767. Note que el rango de los valores positivos es aproximadamente la mitad comparando éstos dos tipos de datos, esto es debido al hecho que uno de los 16 bits es usado por el signo. Esta es una diferencia modesta en el rango y es una razón que justifica que uso indiscrimidado de los tipos unsigned pasaos puramente en el rango de valores positivos que pueden representar.

Para ***datos tipo puntos flotantes***, el tamaño afecta la precisión.

Si el tamaño o la precisión del tipo de dato no es su principal problema, entonces `char`, `int`, y `double` son los que los programadores usualmente usan para representar caracteres, enteros y puntos flotantes (reales) respectivamente. Los otros tipos en los respectivos grupos son utilizados sólo en casos muy particulares.

Los tipos de datos descritos anteriormente (caracteres, enteros, puntos flotantes y booleanos) son colectivamente conocidos como tipos aritméticos. Pero existen adicionalmente dos tipos fundamentales: `void`, que identifica la falta de tipo de dato; y el tipo `nullptr`, que es un tipo especial de puntero. Ambos serán discutidos en capítulos más adelante.

C++ soporta una amplia variedad de tipos basados en los tipos fundamentales discutidos aquí, estos otros tipos son conocidos como "tipos de datos compuestos" y constituyen una de las fortalezas del lenguaje C++, es posible que las veamos en futuros capítulos.

1.9. Declaración de variables

C++ es un lenguaje de programación con una fuerte dependencia a los tipos de datos, cada variable requiere ser declarada con su tipo de dato antes de ser usada por primera vez. Ésto informa al compilador, el tamaño en memoria que debe estar disponible y cómo debe interpretar dicho valor. La

sintaxis para declarar una nueva variable en C++ es una sola: debe escribir el tipo de dato seguido del nombre de la variable (o identificador). Por ejemplo:

```
int a;
float mynumber;
```

Las anteriores son dos declaraciones válidas de variables:

- La primera declara una variable tipo entera `int` con un identificador (o nombre) `a`.
- La segunda declara una variable tipo real o punto flotante `float` con el identificador `mynumber`.

Una vez declaradas, las variables `a` y `mynumber` pueden ser usadas dentro del resto del programa a su alcance. Si declara más de una variable del mismo tipo, pueden ser declaradas en un sólo renglón seprando sus identificadores con comas y terminando el renglón con un punto y coma. Por ejemplo:

```
int a, b, c;
```

Ésto declara tres variables (`a`, `b` y `c`), todas del mismo tipo `int`, y tiene exactamente el mismo significado de :

```
int a;
int b;
int c;
```

Para ver como lucen estas declaraciones en un programa en un ejemplo sobre el ejercicio mental propuesto al principio del numeral 8 de éste capítulo:

Código fuente 5:

```
// operando con variables

#include <iostream>
using namespace std;

int main ()
{
  // declarando las variables:
  int a, b;
  int result;

  // proceso:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // escribir out el resultado:
  cout << result;

```

Salida:

```
4
```

```
21   // terminar el programa:
22   return 0;
23 }
```

No se preocupe si algo aparte de las declaraciones de las variables luce un poco extraño en el programa. Todo será explicado en detalle bien sea en éste o en otro capítulo.

1.9.1 Initialization of variables

Cuando las variables en el ejemplo anterior son declaradas, ellas pueden tener un valor indeterminado hasta que son asignadas a un valor por primera vez. Pero también es posible para una variable tener un valor específico desde el mismo momento de ser declarada. A esto es lo que se llama *inicialización* de la variable: cuando se le dá un valor por primera vez.
En C++ hay tres maneras de inicializar variables. Todas son equivalentes y son producto de la historia evolutiva del lenguaje en años:

La primera es conocida como *inicialización tipo C* (la herencia viene del lenguaje C), consiste en colocar un signo igual (=) entre el tipo de dato y el valor al cual es inicializada:
`type identifier = initial_value;`

Por ejemplo, para declarar una variable tipo `int` llamada `x` e inicializda al valor cero en el momento de ser declarada, escribimos:

```
int x = 0;
```

El segundo método es conocido como la *inicialización tipo constructor* (introducida por el lenguaje C++) donde se encierra el valor entre paréntesis. (()).
`type identifier (initial_value);`
Por ejemplo:

```
int x (0);
```

Finalmente, el tercer método es conocido como inicialización uniforme, es similar a la anterior pero se usan corchetes ({}) en lugar de paréntesis redondos (esta versión fue introducida en la versión revisada de C++ en el 2011).
`type identifier {initial_value};`

Por ejemplo:

```
int x {0};
```

Las tres maneras de inicialar variables son válidas y equivalentes en C++.

Código fuente 6: Salida:

```
// initialización de variables
#include <iostream>
using namespace std;

int main ()
{
  int a=5;               // valor inicial: 5
  int b(3);              // valor inicial: 3
  int c{2};              // valor inicial: 2
  int result;            // valor inicial indeterminado

  a = a + b;
  result = a - c;
  cout << result;

  return 0;
}
```

```
6
```

1.9.2 Introdución a las cadenas de datos

Los tipos de datos fundamentales representan los tipos más basicos que las maquinas puede manipular. Sin embargo, como digimos anteriormente, una de las fortalezas de C++ es la riqueza en tipos compuestos para los cuales los tipos fundamentales son la base que los forma.

Un ejemplo de tipo compuesto es la clase `string`. Variables de este tipo están disponibles para almacenar secuencias de caracteres, tales como palabras o frases. Son una herramienta muy útil!

La primera diferencia con los tipos de datos fundamentales es que para poder declarar y usar objetos (variables) de este tipo de datos, los programas necesitan incluir un encabezado donde el tipo está definido dentro de una librería (en nuestro caso estamos trabajando con la librería estandar), el header es `<string>`:

Código fuente 7: Salida:

```
//Mi segundo string. El primero esta en
//el código fuente 2.
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "Este es un string";
  cout << mystring;
  return 0;
}
```

```
Este es un string
```

Como se ve en el código anterior, las cadenas o strings puden ser inicializados con una una cadena de de letras o caracteres válida literalmente hablando, así como pensaríamos que un tipo de variable numérica sería inicializada con números. De igualmanera que los tipos fundamentales, todos los formatos de inicialización son válidos con las cadenas o strings:

```
1 string mystring = "This is a string";
2 string mystring ("This is a string");
3 string mystring {"This is a string"};
```

Se pueden realizar con los strings todas las operaciones básicas de igual m anera que para los tipos de datos fundamentales, como por ejemplo ser declararas con o sin valores iniciales y cambiar su valor durante el programa:

Código fuente 8:

```
1  //Mi segundo string. El primero esta en //el
   código fuente 2.
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main ()
7  {
8    string mystring;
9    mystring = "Este es el contenido inicial
   del string";
10   cout << mystring << endl;
11   mystring = "Este es el contenido inicial
   del string";
12   cout << mystring << endl;
13   return 0;
14 }
```

Salida:

```
Este es el contenido inicial
del string
Este es el contenido inicial
del string
```

Nota: recuerde que el manipulador `endl` termina la linea (ends the line). Tenga en cuenta que adicionalmente `endl` descarga en el perisférico, en nuestro caso el perisférico es la pantalla, esto lo veremos en más detalles.

1.9. Constantes.

Son expresiones con un valor fijo.

1.9.1 Literales

Son el tipo de constantes más obvias. Son usadas para expresar valores particules dentro de un código fuente de un programa. Ya hemos usado algunas en las secciones anteriores para dar valores específicos a las variables o para expresar mensajes que queremos que se impriman en un perisférico

como la pantalla. Por ejemplo, se puede escribir:

```
a = 5;
```

El 5 de éste trozo de código es llamado una *constante literal*.
Las constantes literales pueden ser clasificadas en: enteras, puntos flotantes, caracteres, strings, booleanas, punteros y literales definidos por el usuario.

1.9.1.1 Literales Enteros.

```
1776
707
-273
```

Son constantes numéricas que identifican valores enteros. En su defecto son del tipo `int`. Sin embargo ciertos sufijos pueden ser colocados a un literal entero para especificar un tipo de entero diferente:

Suffix	Type modifier
`u` *or* `U`	`unsigned`
`l` *or* `L`	`long`
`ll` *or* `LL`	`long long`

Unsigned puede ser combinado con cualquiera de los otros dos para formar `unsigned long` o `unsigned long long`.

For example:

```
75          // int
75u         // unsigned int
75l         // long
75ul        // unsigned long
75lu        // unsigned long
```

En los casos anteriores, los sufijos pueden ser especificados en mayúsculas o minúsculas

1.9.1.2 Numerales de punto flotante.

Expresan valores reales con decimales o exponentes, potencias con base real y con el caracter e y su exponente; y potencias con base entera, el caracter e y exponente. Por ejemplo:

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

El primer número es PI, el segundo el número de Avogadro, el tercero es la carga eléctrica del electrón, todos aproximados. Y el último es el número tres expresado como un literal numérico de punto flotante. En su defecto los literales de punto flotante son `double`. Literales tipo `float` o `long double` pueden ser especificados adicionando los siguientes sufijos:

Suffix	Type
`f` *or* `F`	`float`
`l` *or* `L`	`long double`

Por ejemplo:

```
3.14159L   // long double
6.02e23f   // float
```

Todas las letras que pueden ser parte de una constante numérica de punto flotante (`e`, `f`, `l`) pueden ser escritas usando mayúsculas o minúsculas sin distinción.

1.9.1.3 Literales tipo caracteres y strings:

Estan siempre entre comillas:

```
'z'
'p'
"Hello world"
"How do you do?"
```

Las primeras dos expresiones representan un literal tipo caracter simple y los dos siguientes con literales string compuestos por varios caracteres. Note que para representar un caracter simple se colocan entre comillas simples (') mientras que los strings se colocan entre comillas dobles (").

Varios literales de cadena o strings pueden ser concatenados para formar uno sólo simplemente separandolos por uno o más espacios en blanco, incluyendo tabs o cualquier caracter válido de espacio. Por ejemplo:

```
"this forms" "a single"     " string "
"of characters"
```

La cadena anterior es equivalente a:

```
"this formsa single string of characters"
```

Note como los espacios dentro de las comillas son parte del literal mientras los que estan fuera no lo son.

Algunos programadores incluyen largos literales de cadena en multiples líneas, en C++, un backslash (\) al final de la linea es considerado como un caracter de continuación de línea que une las líneas en una sola. Por ejemplo:

```
x = "string expressed in \
two lines"
```

Es equivalente a

```
x = "string expressed in two lines"
```

1.9.1.4 Otros literales

Existen tres palabras claves en los literales en C++: `true`, `false` y `nullptr`:

- `true` y `false` son los dos valores posibles para el tipo `bool`.

- `nullptr` es el valor de *pointer nulo*.

```
bool foo = true;
bool bar = false;
int* p = nullptr;
```

1.9.2. Expresiones tipo constantes:

Algunas veces es conveniente dar un nombre a un valor constante:

```
const double pi = 3.1415926;
const char tab = '\t';
```

Se pueden usar estos nombres en lugar de los literales a los que éstos se refieren:

Código fuente 9:

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

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
  double r=5.0;                 // radius
  double circle;

  circle = 2 * pi * r;
  cout << circle;
  cout << newline;
}
```

Salida:

```
31.4159
```

1.9.3. Definición en el preprocesador (#define)

Otro mecanismo para dar nombres a valores constantes es usando definiciones en el preprocesador, que tienen la forma:
`#define identificador remplazo`

Después de esta línea de comando, cualquier aparición de `identificador` en el código es interpretada como `remplazo`, donde el remplazo es cualquier secuencia de caracteres (hasta el final de la línea). El remplazo es realizado por el preprocesador y ocurre antes que el programa sea compilado, lo que lo hace una especie de remplazo a ciegas, la validez del tipo, o la sintaxis involucrada no es revisada en ningun momento.
Por ejemplo:

Código fuente 10

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

#define PI 3.14159
#define NEWLINE '\n'

int main ()
```

Salida

```
31.4159
```

```
{
  double r=5.0;                 // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

}
```

Note que las líneas `#define` son directivas del preprocesador y como tal son líneas de intrucciones o comandos simples que, a menos que sea un requerimiento particuar de C++, no requieren punto y coma (;) al final y la directiva se extiende hasta el final de la linea.

1.10. Operadores

Luego de introducir las variables y constantes podemos operarlas usando los *operadores*. En lo seguido se hará una lista de los operadores, no es necesario conocerlos y entenderlos todos, pero es una lista que es importante tenerla siempre a mano pues sirve de referencia para cuando se necesite.

1.10.1 Operador de asignación (=)

Asigna un valor a una variable, de derecha a izquierda y NUNCA en la otra dirección.

```
x = 5;
```

En el siguiente ejemplo

```
x = y;
```

El valor anterior de x se pierde y x toma el nuevo valor dado por y.

Tomemos el ejemplo del siguiente código, el contenido de las variables está expresado en los comentarios de la línea correspondiente.

Código fuente 11

```
// assignment operator
#include <iostream>
using namespace std;

int main ()
{
  int a, b;         // a:?,  b:?
  a = 10;           // a:10, b:?
  b = 4;            // a:10, b:4
  a = b;            // a:4,  b:4
  b = 7;            // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;
}
```

Salida

```
a:4 b:7
```

La siguiente expresión también es válida en C++

```
x = y = z = 5;
```

Donde se está asignando el valor de 5 a todas las tres variables x, y, z, siempre de derecha a izquierda.

1.10.2 Operadores Aritméticos

Los 5 operadores aritméticos de C++ son:

operador	descripción
+	suma
-	resta
*	multiplicación
/	división
%	módulo

El operador módulo da el residuo de una división entre dos valores. Por ejemplo:

```
x = 11 % 3;
```

El resultado en la variable x contiene el valor 2, pues al dividir 11 en 3 el resultado es 3 y el residuo es 2.

1.10.3 Asignaciones compuestas (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Los operadores de asignaciones compuestas modifican el valor actual de una variable realizando unan operación sobre ésta. Eso es equivalente a asignar el resultado de una operación a la que fue primero operada:

expresión	equivalente a...
`y += x;`	`y = y + x;`
`x -= 5;`	`x = x - 5;`
`x /= y;`	`x = x / y;`
`x *= unidad + 1;`	`x = x * (unidad+1);`

Y lo mismo para los otros operadores compuestos asignados, por ejemplo:

Código fuente 12

```
1  // compound assignment operators
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int a, b=3;
8    a = b;
9    a+=2;              // equivalent to a=a+2
10   cout << a;
11 }
```

Salida

```
5
```

1.10.4 Incrementos y decrementos (++,---)

Algunas expresiones pueden ser acortadas aún más: el operador de incremento (++) y el de decremento(--), aumenta o reduce por uno el valor almacenado en la variable. Ellos son el equivalente a +=1 y a -=1 respectivamente:

```
1 ++x;
2 x+=1;
3 x=x+1;
```

Todos son equivalentes en su funcionalidad; los tres incrementan por uno el valor de la variable x.

Dicho operador se puede usar como prefijo (++x) y como sufijo (x++). Estas expresiones pueden cambiar la operación a evaluar dependiendo de su utilización. En el caso de usarlo como prefijo del valor (++x), la expresión evalúa al valor final de x una vez ya está incrementado. Por otro lado, en el caso de sufijo (x++), el valor es incrementado pero la expresión evalúa al valor que x tiene antes de ser incrementada. Note la diferencia en el siguiente código:

Ejemplo 1	Ejemplo 2

`x = 3;`
`y = ++x;`
`// x contiene 4, y contiene 4`
	`x = 3;`
	`y = x++;`
	`// x contiene 4, y contiene 3`

En el *Ejemplo 1*, el valor asignado a y es el valor de x despues de ser incrementado. Mientras que en *Ejemplo 2*, el valor de y es el valor que x tiene antes de ser incrementado. En ambos casos el valor de x es incrementado, asignación a y es la que cambia de caso a caso.

1.10.5 Operadores de Relación y comparación (==, !=, >, <, >=, <=)

Dos expresiones pueden ser comparadas usando los operadores de relación e igualdad. Por ejemplo, para saber si dos valores son iguales o si uno es mayor que otro.

El resultado de dichas operaciones es bien sea verdadero o falso, esto es un valor booleano. Los operadores de relación en C++ son:

operador	descripción
==	Igual a
!=	No igual a
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

Algunos ejemplos.

```
1 (7 == 5)      // evalúa falso
2 (5 > 4)       // evalúa verdadero
```

```
(3 != 2)      // evalúa verdadero
(6 >= 6)      // evalúa verdadero
(5 < 5)       // evalúa falso
```

No es sólo para valores numéricos, cualqueir valor puede ser comparado, incluyendo variables. Suponga que `a=2`, `b=3` y `c=6`, entonces:

```
(a == 5)      // evalúa falso, ya que a no es igual 5
(a*b >= c)    // evalúa verdadero, ya que (2*3 >= 6) es verdadaero
(b+4 > a*c)   // evalúa falso, ya que (3+4 > 2*6) es falso
((b=2) == a) // evalúa verdadero
```

Tenga cuidado! El operador de asignación (=, con un sólo signo de igualdad) NO es el mismo al operador de comparación de iguadad (==, con dos signos de igualdad); El primero (=) asigna el valor de la derecha a la variable de la izquierda. El segundo (==) compara si los valores a ambos lados son iguales. Por tanto en la expresión (`(b=2) == a`), primero se asigna el valor de 2 a b y luego se compara con a, la cual también tiene el valor de 2 y por lo tanto es `true (verdadero)`.

1.10. 6. Operadores Lógicos (!, &&, ||)

El operador (!) en C++ es destinado para la operación booleana NOT. Este sólo opera hacía su derecha invirtiendo el resultado, es decir produce falso si el operando era verdadero o verdadero si el operando era falso. Básicamente devuelve el opuesto del valor booleano de la operación realizada. Por ejemplo:

```
!(5 == 5)    // evalúa falso porque la expresión a la derecha (5==5) es verdadero
!(6 <= 4)    // evalúa verdadero porque (6 <= 4) sería falso
!true        //  evalúa falso
!false       // evalúa verdadero
```

Los operadores lógicos && y || son usados cuando se evalúan dos expresiones para obtener un único resultado que las relaciones. El operador && corresponde a la operación lógica booleana AND, la cual devuelve verdadero si ambas expresiones son verdaderas y falso de otro modo. En el siguiente cuadro se relaciona el resultado del operador && evaluando la expresión `a&&b`:

&& OPERADOR (AND)		
a	**b**	**a && b**
true	true	true
true	false	false
false	true	false
false	false	false

El operador || corresponde a la operación lógica booleana OR, la cual arroja verdadero si una de las expresiones es verdadera, y arroja falso sólo cuando ambas expresiones son falsas. En la tabla estan los posibles resultados de `a||b`:

|| OPERADOR (OR)		
a	**b**	**a || b**
true	true	true
true	false	true

`false`	`true`	`true`
`false`	`false`	`false`

Por ejemplo:

```
( (5 == 5) && (3 > 6) )  // evalúa falso ( true && false )
( (5 == 5) || (3 > 6) )  // evalúa verdadero ( true || false )
```

Cuando se usan los operadores lógicos, C++ sólo evalúa lo que es necesario de izquierda a derecha para devolver un resultado que relaciona las dos expresiones ignorando el resto. En el ejemplo ((5==5)||(3>6)), C++ evalúa primero si 5==5 es verdadero, si esto es así, nunca evalúa si 3 > 6 es verdadero o falso. Esto se conoce como una evaluación de circuito corto. Y funciona de la siguiente manera:

operador	Circuito-corto
`&&`	Si la expresión en el lado izquierdo es `false`, el resultado combinado es `false` (la expresión de la derecha nunca es evaluada)
`||`	Si la expresión de la izquierda es `true`, el resultado combinado es `true` (el lado derecho nunca es evaluado).

Esto es importante tenerlo en cuenta cuando la expresión de la derecha altera el valor de la variable:

```
if ( (i<10) && (++i<n) ) { /*...*/ }   // note that the condition increments i
```

Aquí la expresión condicional combinada se incrementa en uno SOLO SI la condición en la izquierda de && es `true`, porque de otro modo la condición de la derecha (`++i<n`) nunca es evaluada.

1.10.7. Operadores Condicionales de Ternas (?)

Este operador condicional evalúa la expresión `condition`, devolviendo el primer valor `resultado1` si la expresión evaluada es `true`, y el segundo valor `resultado2` si la expresión evaluada es `false`. La sintaxis es:

```
condicion ? resultado1 : resultado2
```

Por ejemplo:

```
7==5 ? 4 : 3      // evalúa a 3, ya que 7 no es igual a 5.
7==5+2 ? 4 : 3    // evalúa a 4, ya que 7 es igual a 5+2.
5>3 ? a : b       // evalúa al valor de a, ya que 5 es mayor que 3.
a>b ? a : b       // evalúa al que sea mayor, a o b.
```

Código fuente 13: Salida:

```
// conditional operator
#include <iostream>
using namespace std;

int main ()
{
  int a,b,c;
```

```
7
```

```
 8
 9   a=2;
10   b=7;
11   c = (a>b) ? a : b;
12
13   cout << c << '\n';
14 }
```

En este ejemplo, a era 2 y b era 7, de tal manera que las expresiones evaluadas (a>b) no es verdadera, así que el primer valor especificado despues del signo de interrogación fue descartado y fue favorecido el segundo valor, el que se encuentra después de los dos puntos, el cual fue b con el valor de 7.

1.10.8. Operador Coma (,)

Es usado para separar dos o más expresiones que eson incluidas donde se espera una sola expresión. Cuando el conjunto de expresiones es evaluada sólo la expresión que se encuentre a la extrema derecha es considerada.

Por ejemplo:

```
a = (b=3, b+2);
```

en este caso primero se asigna el valor de 3 a b y luego se asigna el valor de b+2 a la variable a. Por tanto la variabla a contiene el valor de 5 y b contiene 3.

1.10.9. Operadores Binarios (&, |, ^, ~, <<, >>)

Los operadores binarios modifican variables considerando los patrones binarios que representan los valores que ellos almacenan.

operador	asm equivalente	descripción
&	AND	AND binario
|	OR	OR inclusivo binario
^	XOR	OR exclusivo binario
~	NOT	Unario complemento (inversión de bit)
<<	SHL	Corrimiento de bits a la izquierda
>>	SHR	Corrimiento de bits a la derecha

1.10.10 Operador de Cambio de tipo dato

Permiten convertir un valor de un cierto tipo de dato a otro tipo de dato. Hay varias maneras de hacerlo en C++. La más simple, heredada del lenguaje C, es preceder la expresión a ser convertida con el nuevo tipo de dato encerrado entre paréntesis (()):

```
1 int i;
2 float f = 3.14;
3 i = (int) f;
```

En este caso, se convierte el número de punto flotante 3.14 a un valor entero 3, el resto del número se

pierde. El operador que cambia el tipo de dato es ((int)).

Otra manera de hacerlo es usar una notación de funciones precediendo a la expresión a ser convertida por el nuevo tipo de dato y encerrar la expresión entre paréntesis.

```
i = int (f);
```

Ambas maneras de cambiar el tipo de dato son válidas.

1.10.11. sizeof

Este operador acepta sólo un parámetro, el cual puede ser o bien sea un tipo de dato o una variable y retorna el tamaño en bytes del parámetro y lo asigna a una variable.

```
x = sizeof (char);
```

Aquí, a x se le asigna el valor de 1, porque char es un tipo de dato con el tamaño de 1 byte.

El valor retornado por sizeof se llama constante de tiempo de compilación debido a que se determina antes durante el proceso de compilación antes que sea ejecutado.

Otros operadores.

El lenguaje C++ soporta algunos operadores adicionales, como aquellos que se refieren a los punteros o aquellos que se especifican directamente en la programación orientada a objetos.

1.10.12 Prelación de los operadores

Una expresión puede tener multiples operadores. Por ejemplo:

```
x = 5 + 7 % 2;
```

Note la diferencia entre las siguientes expresiones, el hecho de usar paréntesis incluye la prelación en el orden:

```
1 x = 5 + (7 % 2);     // x = 6 (lo mismo que sin paréntesis)
2 x = (5 + 7) % 2;     // x = 0
```

De mayor a menor prioridad, los operadores de C++ son evaluados en el siguiete orden:

Nivel	Grupo de prelación	Operador	Descripción	Orden
1	Acción	::	Acción de calificar	Izq-a-der
2	Postfix (unary) o sufijo	++ --	sufijo aumentar /disminuir	Izq-a-der
		()	Forma funcional	
		[]	subindice	
		. ->	Miembro de acceso	
3	Prefix (unary)	++ --	prefijo aumentar /disminuir	Der-a-Izq

		`~ !`	NOT binario/ NOT lógico	
		`+ -`	Prefijo unario	
		`& *`	referencia / dereferencia	
		`new delete`	alocación / delocación	
		`sizeof`	Parámetro de paquete	
		`(`*`type`*`)`	estilo-C type-casting	
4	Puntero-a-miembro	`.* ->*`	Acceseo a pointer	Izq-a-der
5	Aritmetica: escalamiento	`* / %`	multiplicar, dividir, módulo	Izq-a-der
6	Aritmetica: adición	`+ -`	suma, resta	Izq-a-der
7	Desplazamiento binario	`<< >>`	Corrimiento a izq, corrimiento a right	Izq-a-der
8	Relación	`< > <= >=`	Operadores de comparación	Izq-a-der
9	Igualdad	`== !=`	Igualdad / desigualdad	Izq-a-der
10	And	`&`	AND binario	Izq-a-der
11	Exclusivo or	`^`	XOR binario	Izq-a-der
12	Inclusivo or	`|`	OR binario	Izq-a-der
13	Conjunción	`&&`	AND lógico	Izq-a-der
14	Disyunción	`||`	OR lógico	Izq-a-der
15	Asignación-expresions de nivel	`= *= /= %= += -= >>= <<= &= ^= |=`	asignación / asignación compuesta	Der-a-Izq
		`?:`	Operador condicional	
16	Secuencia	`,`	Separador coma	Izq-a-der

Cuando una expresión tiene odos operadores con el mismo nivel de prelación, el Orden o agrupamiento determina cual es evaluado primero, bien sea de Izq-a-der (izquierda a derecha) o de Der-a-Izq (derecha a izquierda).

1.11. Lectura/Escritura (Input/Output) usando archivos

C++ provee las siguientes clases para realizar la escritura y la lectura de caracteres a/desde archivos:

- **ofstream:** Stream clase para escribir en archivos
- **ifstream:** Stream class para leer desde archivos
- **fstream:** Stream class to both leer y escribir desde/a archivos.

Estas clases provienen directa o indirectamente de las clases istream y ostream que serán entendidas más adelante en el curso. Ya se han usando objetos de estas clases: cin es un objeto de la clase istream y cout es un objeto de la clase ostream. Así que ya se han usado estas clases que estamos relacionando con el flujo de archivos. Y de hecho los archivos se usarán de la misma manera como se han usado tanto cin como cout, con sólo la diferencia que debemos asociar es particularmente a un archivo físico.

Veamos por ejemplo:

Código fuente 14:

```
1  // operaciones básicas con archivos
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    ofstream file;
8    file.open ("ejemplo.txt");
9    file << "Escribiendo en file.\n";
10   file.close();
11   return 0;
12 }
```

```
[file ejemplo.txt]
Escribiendo en file.
```

Este código crea un archivo llamado ejemplo.txt e incerta una frase en éste de la misma manera que se ha usado con cout pero esta vez usando el apuntador del archivo llamado file.

Para leer desde un archivo se realiza de la misma manera que se hizo con cin:

Código fuente 15:

```
1  // lectura de un archivo
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6
7  int main () {
8    string line;
9    ifstream file ("ejemplo.txt");
10   if (file.is_open())
11   {
12     while ( getline (file,line) )
13     {
14       cout << line << '\n';
15     }
16     file.close();
17   }
18
19   else cout << "No se pudo abrir file";
20
21   return 0;
22 }
```

```
This is a line.
Esta es la linea que
se extrae del
archivo.
```

En el ejemplo se lee un archivo de texto y se imprime su contenido en la pantalla. Un loop-while lee el archivo línea por línea usando `getline`. El valor retornado por `getline` es una referencia por si mismo y es usado como condición booleana a evaluar en el loop: si es verdadera es porque la función puede seguir siendo operada, y si es falsa es porque o bien el archivo ha llegado al final o algún error ha ocurrido.