Informe tutorial de C++

Martínez Walteros Andrés Camilo

Cristian Stiven Sánchez Rubio

Johan Andrés Pardo Polanco

Jhonatan Stiven Herrera

María Liliana Leiva

Servicio Nacional De Aprendizaje SENA

Centron De Diseño Y Metrologa

Analisis Y Desarrollo De Sistemas De Informacion

Bogotá

2015

Informe tutorial de C++

Martínez Walteros Andrés Camilo

Cristian Stiven Sánchez Rubio

Johan Andrés Pardo Polanco

Jhonatan Stiven Herrera

María Liliana Leiva

Jimmy Daniel Gamba Casallas

Servicio Nacional De Aprendizaje SENA

Centron De Diseño Y Metrologa

Analisis Y Desarrollo De Sistemas De Informacion

Bogotá

2015

# CONTENIDO

1. RESUMEN 5
2. INTRODUCCION 6
3. CAPITULO 16 CLASES Y SUS AMIGAS 7
4. CAPITULO 17 ENTRADA Y SALIDA 11
5. CAPITULO 18 PREPROCESADOR 14
   1. Directiva #define 14
   2. Directiva #undef 15
   3. Directivas #if, #elif, #else y #endif 16
   4. Directivas #ifdef e #ifndef 18
   5. Directiva ”#error” 19
   6. Directiva ”#include” 20
   7. Directiva ”#line” 21
   8. Directiva ”#pragma” 22
   9. Directiva “#pragma pack()” 23
   10. Directiva #warning 23
6. CONCLUCIONES 24
7. BIBLIOGRAFIA 25

**Resumen**

En el presente trabajo se analizan los capítulos 16,17 y 18 del libro tutoriales de C++.

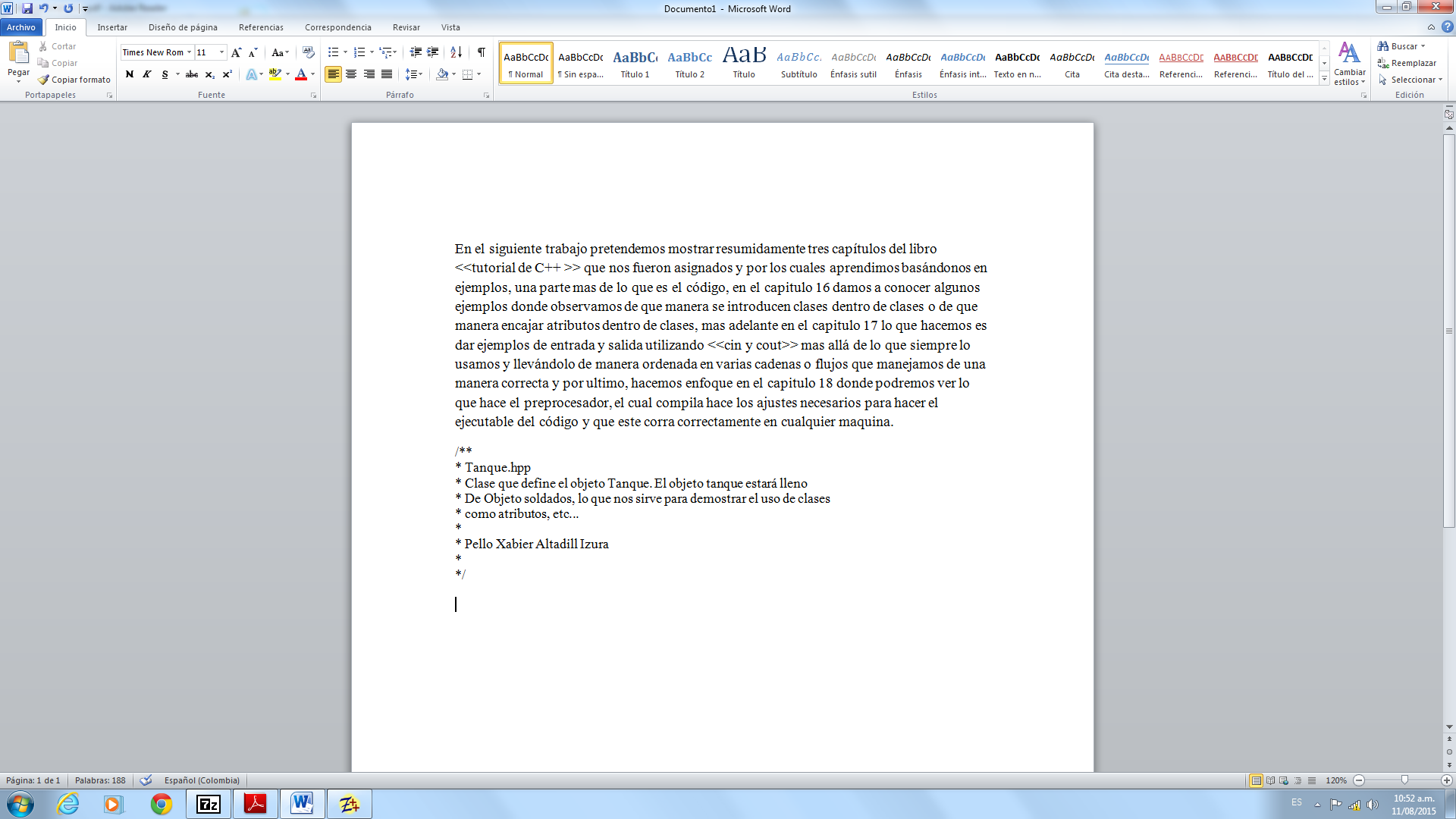
En el primer capítulo podemos observar la programación orientada a objetos por medio de las clases y como estas clases se unen entre ellas llamándose clases amigas unas de otras y se comparten información de la una a la otra.

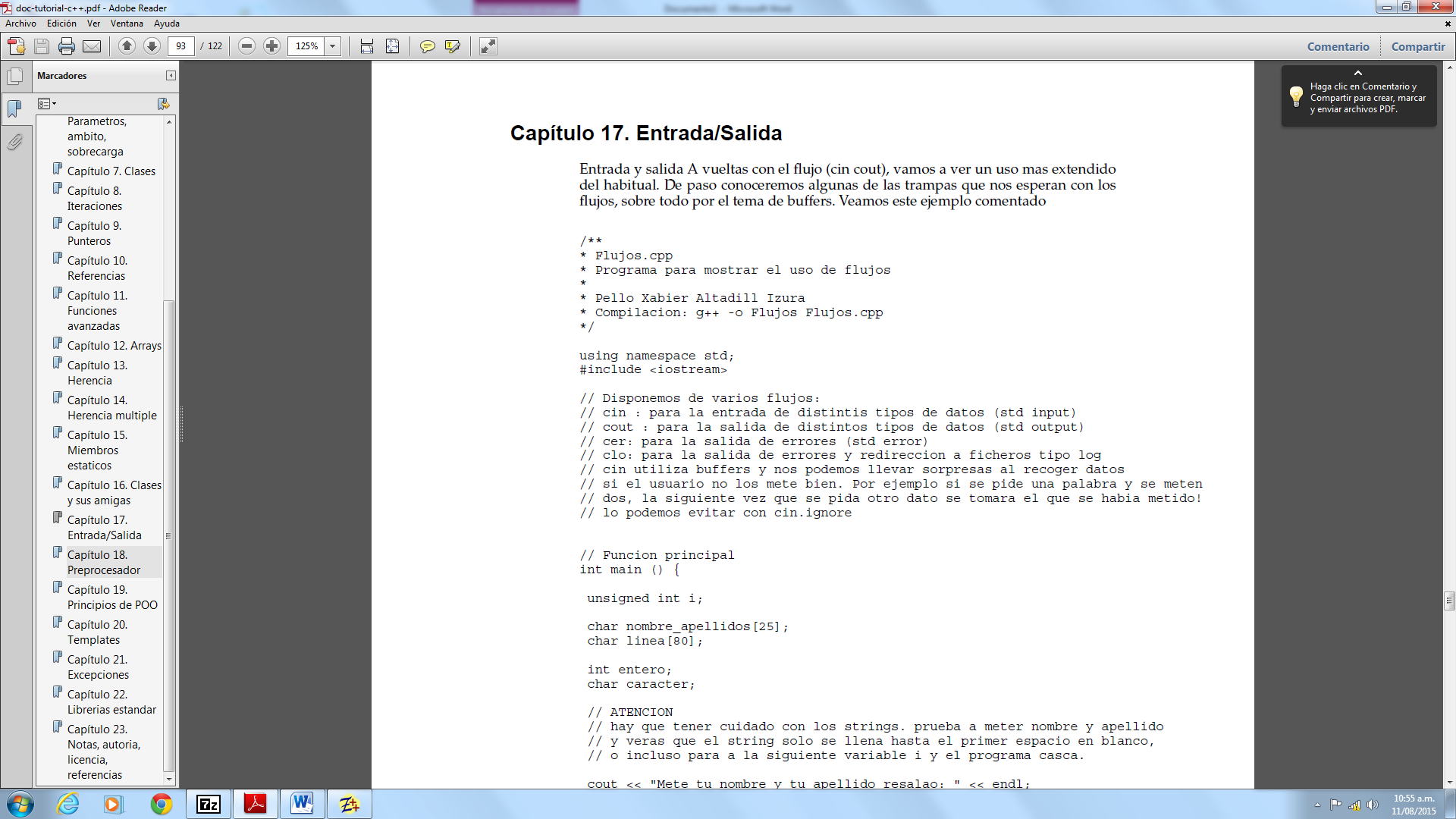
En el segundo capítulo podemos observar la importancia de las entradas y las salidas en C++ esto se maneja por medio de (cin y cout) de esta manera el programa puede recibir información del usuario que hace usos del aplicativo y también mostrar información al usuario. En el último capítulo se relata la importancia del preprocesador en donde podemos observar que el preprocesador es el que ejecuta las directivas del programa antes de iniciar a ejecutar el programa.

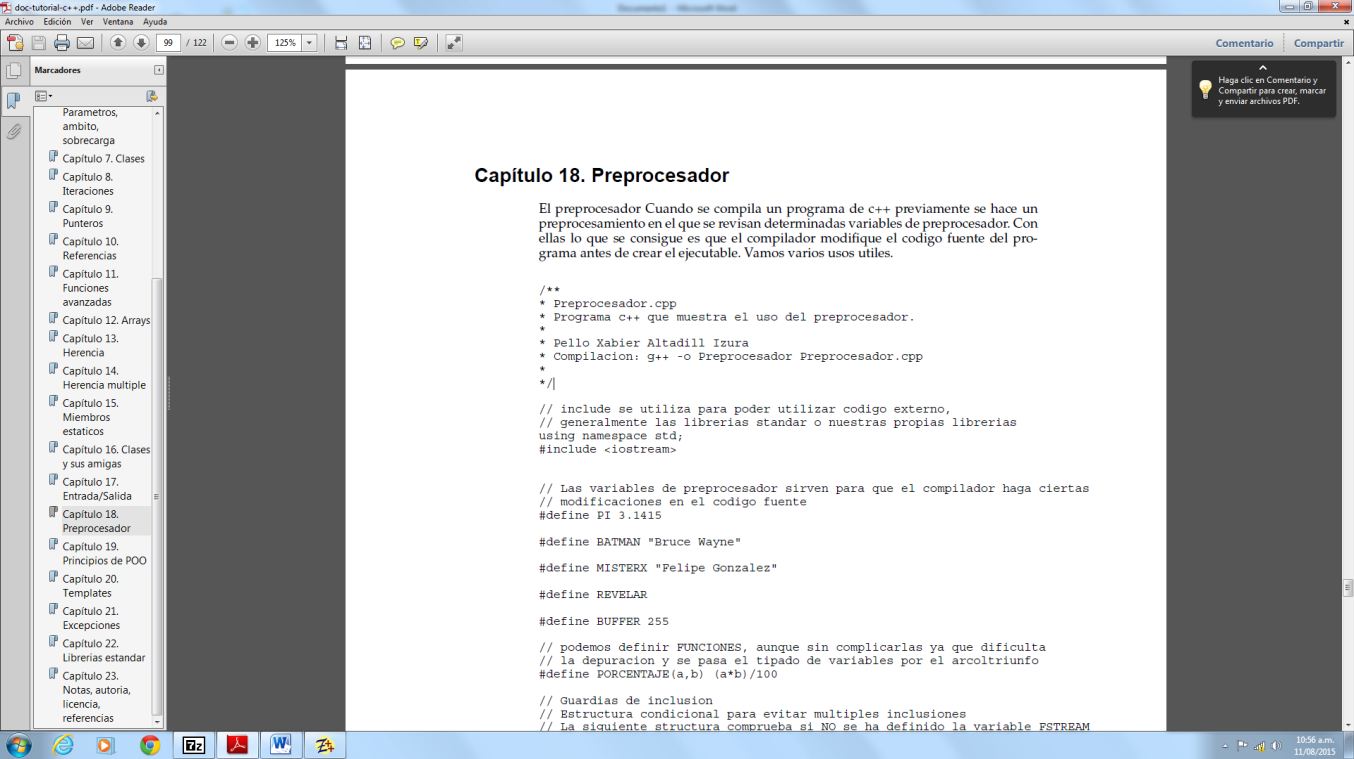
En el presente trabajo podemos observar bastantes ejemplos de cómo se realizan las uniones entre clases y como una clase amiga de otra puede compartir información, estos ejemplos ayudan y facilitan el entendimiento de este tema que tal vez para muchos es un poco complejo, además en el capítulo 18 se hace un relato de cada una de las directivas que se utilizan en la programación C++ y como los lee el programa para convertir estas indicaciones a un lenguaje que entiende la máquina y ejecutar las instrucciones del programador.

**Introducción**

En el siguiente trabajo pretendemos mostrar resumidamente tres capítulos del libro <<tutorial de C++ >> que nos fueron asignados y por los cuales aprendimos basándonos en ejemplos, una parte más de lo que es el código, en el capítulo 16 damos a conocer algunos ejemplos donde observamos de qué manera se introducen clases dentro de clases o de qué manera encajar atributos dentro de clases, más adelante en el capítulo 17 lo que hacemos es dar ejemplos de entrada y salida utilizando <<cin y cout>> más allá de lo que siempre lo usamos y llevándolo de manera ordenada en varias cadenas o flujos que manejamos de una manera correcta y por último, hacemos enfoque en el capítulo 18 donde podremos ver lo que hace el preprocesador, el cual compila hace los ajustes necesarios para hacer el ejecutable del código y que este corra correctamente en cualquier máquina.







**Capítulo 16 Clases Y Sus Amigas**

Uno de los aspectos fundamentales de este capítulo son las clases por ello es importante definir que son las clases en programación.  Una clase es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. Cada clase es un modelo que define un conjunto de variables y métodos apropiados. Para operar con dichos datos en otras palabras el comportamiento. Cada objeto creado a partir de la clase se denomina instancia de la clase. Las clases están constituidas de varios objetos que tienen aspectos en común.

En C++ la clase es la fundación para el soporte de la programación orientada objetos. Dentro de las clases podemos definir funciones (o clases) como amigas de la clase. Esto significa que le estamos dando acceso a una función (o clase) a que modifique los datos miembros de la clase. En otras palabras podríamos decir que las clases se unen y comparten información entre sí. Esto es especialmente útil para la sobrecarga de operadores como veremos más adelante.

A estas clases que se unen entre si se les llama clases amigas. Una función (o clase) amiga tiene acceso a los datos miembros de la clase que la declara como amiga.

Para ilustrar esto veremos el siguiente ejemplo. Tenemos una clase llamada Número, que consta de un dato miembro llamado x. Tiene una función pública mostrar () que saca por pantalla el valor de x. Además declara una función amiga llamada establecer (), que recibe el objeto a modificar y un número de punto flotante doublé para establecer el valor del objeto.

Como vemos estamos utilizando la función no a través del objeto sino pasándole el objeto pasado como parámetro.

#include <iostream>

using std::cout;

using std::endl;

class Numero

{

friend void establecer(double ,Numero &);//establecer es una funcion amiga.

public:

Numero(double);//constructor.

void mostrar();

private:

double x;//dato miembro.

};

Numero::Numero(double b)

{

x = b;

}

void Numero::mostrar()

{

cout << "El numero es: " << x << endl;

}

void establecer(double a, Numero &objeto)

{

objeto.x = a; //puede modificar el dato miembro porque es amiga.

}

int main()

{

cout << "Creamos el objeto Pi" << endl;

Numero Pi(7.8);

Pi.mostrar();

cout << "Modificamos con la funcion amiga" << endl;

establecer(3.14,Pi);//utilizamos la funcion amiga.

Pi.mostrar();

return 0;

}

Características de las relaciones de amistad Declaro relaciones de amistad entre clases o funciones.

● La amistad no puede transferirse: Si A es amigo de B, y B es amigo de C -> A no tiene que ser amigo de C (y viceversa) “Los amigos de mis amigos son mis amigos” NO APLICA.

● La amistad no puede heredarse: Si A es amigo de B, y C deriva de B -> A no tiene que ser amigo de C (y viceversa) “Los hijos de mis amigos son mis amigos” NO APLICA. La amistad no es simétrica: Si A es amigo de B -> B no tiene por qué se amigó de A “....”

**Introducción:** funciones amigas

Hemos visto que el acceso entre clases es imposible cuando definimos los miembros como private.

class Punto

{

private :

float x;

float y;

public :

Punto ( );

void visualizar ( );

} ;

* Desde una clase no se puede acceder a los métodos o atributos privados de otra clase.
* Desde una función normal tampoco se puede acceder a miembros privados de una clase

**Capítulo 17 Entrada/Salida**

Entrada y salida A vueltas con el flujo (cin cout), vamos a ver un uso más extendido

Del habitual.

Cin: para la entrada de distintos tipos de datos

Cout: para la salida de distintos tipos de datos

Cer: para la salida de errores

Clo: para la salida de errores y redirección a ficheros tipo log

Cin utiliza buffers y nos podemos llevar sorpresas al recoger datos

* dos, la siguiente vez que se pida otro dato se tomara el que se había metido! lo podemos evitar con cin.ignore

Hay que tener cuidado con los strings. Prueba a meter nombre y apellido y veras que el string solo se llena hasta el primer espacio en blanco, o incluso para a la siguiente variable i y el programa casca.

Función principal

int main

Cuando nos referimos a entrada/salida estándar (E/S estándar) queremos decir que los [datos](http://www.monografias.com/trabajos11/basda/basda.shtml) o bien se están leyendo del [teclado](http://www.monografias.com/trabajos5/sisope/sisope2.shtml#tecla), ó bien se están escribiendo en el [monitor](http://www.monografias.com/trabajos5/losperif/losperif2.shtml#moni) de [video](http://www.monografias.com/trabajos10/vire/vire.shtml). Como se utilizan muy frecuentemente se consideran como los dispositivos de E/S por default y no necesitan ser nombrados en las instrucciones de E/S.

Las [operaciones](http://www.monografias.com/trabajos6/diop/diop.shtml) de entrada y salida no forman parte del conjunto de sentencias de C++, sino que pertenecen al conjunto de [funciones](http://www.monografias.com/trabajos7/mafu/mafu.shtml) y clases de la [biblioteca](http://www.monografias.com/trabajos10/ponency/ponency.shtml) estándar de C++. Ellas se incluyen en los [archivos](http://www.monografias.com/trabajos7/arch/arch.shtml) de cabecera **iostream.h** por lo que siempre que queramos utilizarlas deberemos introducir la línea de [código](http://www.monografias.com/trabajos12/eticaplic/eticaplic.shtml) **#include <iostream.h>**

 Esta biblioteca es una implementación orientada a objetos y está basada en el [concepto](http://www.monografias.com/trabajos10/teca/teca.shtml) de flujos. A nivel abstracto un flujo es un medio de describir la secuencia de datos de una fuente a un destino o sumidero. Así, por ejemplo, cuando se introducen caracteres desde el teclado, se puede pensar en caracteres que fluyen o se trasladan desde el teclado a las [estructuras](http://www.monografias.com/trabajos15/todorov/todorov.shtml#INTRO) de datos del programa.

    Los objetos de flujo que vienen predefinidos serán:

1. **cin,** que toma caracteres de la entrada estándar (teclado);
2. **cout**, pone caracteres en la salida estándar (pantalla);
3. **cerr y clog** ponen mensajes de error en la salida estándar.

**Operaciones de entrada**

Algo muy usual en un programa es esperar que el usuario introduzca datos por el teclado. Para ello contamos con varias posibilidades: Usar las funciones de la biblioteca estándar, crear nuestras propias interrupciones de teclado ([MS-Dos](http://www.monografias.com/trabajos/manualdos/manualdos.shtml)) o usar funciones de alguna biblioteca diferente (como por ejemplo Allegro).

En este [trabajo](http://www.monografias.com/trabajos34/el-trabajo/el-trabajo.shtml) topare solo la primera opción, usando las funciones de la biblioteca estándar. Pero veamos por encima las otras posibilidades.

Las funciones estándar están bien para un programa sencillito. Pero cuando queremos hacer [juegos](http://www.monografias.com/trabajos15/metodos-creativos/metodos-creativos.shtml) por ejemplo, no suelen ser suficiente. Demasiado lentas o no nos dan todas las posibilidades que buscamos, como comprobar si hay varias teclas pulsadas. Para solucionar esto tenemos dos posibilidades:

La más complicada es crear nuestras propias interrupciones de teclado. ¿Qué es una interrupción de teclado? Es un pequeño programa en [memoria](http://www.monografias.com/trabajos13/memor/memor.shtml) que se ejecuta continuamente y comprueba [el estado](http://www.monografias.com/trabajos12/elorigest/elorigest.shtml) del teclado. Podemos crear uno nuestro y hacer que el ordenador use el que hemos creado en vez del suyo.

**Capitulo 18 Preprocesador**

La parte fundamental de este capítulo es la función que tiene el procesador a la hora de realizar un programa. El preprocesador como su nombre lo indica pre por que realiza una función antes de ejecutar el código es el que analiza el fichero fuente antes de la fase de compilación real, y realiza las sustituciones de macros y procesa las directivas del preprocesador. El preprocesador también elimina los comentarios. El preprocesador se encarga de convertir todo el código que nosotros vemos a 0 y 1 ya que este es el lenguaje que entiende la maquina.

El preprocesador es el encargado de la inclusión de ficheros, compilación condicional y definición y expansión de macros. Para indicarle al preprocesador la realización de estas tareas este posee un lenguaje propio. Las directivas del preprocesador comienzan siempre con el símbolo # y emplean toda la línea en la que se utilizan.

Una directiva de preprocesador es una línea cuyo primer carácter es un #.

A continuación se describen las directivas del preprocesador.

* **Directiva #define**

La directiva #define, sirve para definir macros. Las macros suministran un sistema para la sustitución de palabras, con y sin parámetros.

**Sintaxis:** #define identificador\_de\_macro <secuencia>

El preprocesador sustituirá cada ocurrencia del identificador\_de\_macro en el fichero fuente, por la secuencia. Aunque como veremos, hay algunas excepciones. Cada sustitución se conoce como una expansión de la macro. La secuencia es llamada a menudo cuerpo de la macro.

Si no se especifica una secuencia, el identificador\_de\_macro sencillamente, será eliminado cada vez que aparezca en el fichero fuente.

Después de cada expansión individual, se vuelve a examinar el texto expandido a la búsqueda de nuevas macros, que serán expandidas a su vez. Esto permite la posibilidad de hacer macros anidadas. Si la nueva expansión tiene la forma de una directiva de preprocesador, no será reconocida como tal.

Existen otras restricciones a la expansión de macros:

Las ocurrencias de macros dentro de literales, cadenas, constantes alfanuméricas o comentarios no serán expandidas.

Una macro no será expandida durante su propia expansión, así #define A A, no será expandida indefinidamente.

No es necesario añadir un punto y coma para terminar una directiva de preprocesador. Cualquier carácter que se encuentre en una secuencia de macro, incluido el punto y coma, aparecerá en la expansión de la macro. La secuencia termina en el primer retorno de línea encontrado. Las secuencias de espacios o comentarios en la secuencia, se expandirán como un único espacio.

* **Directiva #undef**

Sirve para eliminar definiciones de macros previamente definidas. La definición de la macro se olvida y el identificador queda indefinido.

**Sintaxis:** #undef identificador\_de\_macro

La definición es una propiedad importante de un identificador. Las directivas condicionales #ifdef e#ifndef se basan precisamente en esta propiedad de los identificadores. Esto ofrece un mecanismo muy potente para controlar muchos aspectos de la compilación.

Después de que una macro quede indefinida puede ser definida de nuevo con #define, usando la misma u otra definición.

Si se intenta definir un identificador de macro que ya esté definido, se producirá un aviso, un warning, si la definición no es exactamente la misma. Es preferible usar un mecanismo como este para detectar macros existentes:

#ifndef [NULL](http://c.conclase.net/librerias/?ansimac=NULL)

#define [NULL](http://c.conclase.net/librerias/?ansimac=NULL) 0L

#endif

De este modo, la línea del #define se ignorará si el símbolo NULL ya está definido.

* **Directivas #if, #elif, #else y #endif**

Permiten hacer una compilación condicional de un conjunto de líneas de código.

**Sintaxis:**

#if expresión-constante-1 <sección-1>

#elif <expresión-constante-2> <sección-2>

#elif <expresión-constante-n> <sección-n>

<#Else> <sección-final>

#endif

Todas las directivas condicionales deben completarse dentro del mismo fichero. Sólo se compilarán las líneas que estén dentro de las secciones que cumplan la condición de la expresión constante correspondiente.

Estas directivas funcionan de modo similar a los operadores condicionales C++. Si el resultado de evaluar la expresión-constante-1, que puede ser una macro, es distinto de cero (true), las líneas representadas por sección-1, ya sean líneas de comandos, macros o incluso nada, serán compiladas. En caso contrario, si el resultado de la evaluación de la expresión-constante-1, es cero (false), la sección-1 será ignorada, no se expandirán macros ni se compilará.

En el caso de ser distinto de cero, después de que la sección-1 sea preprocesada, el control pasa al #endifcorrespondiente, con lo que termina la secuencia condicional. En el caso de ser cero, el control pasa al siguiente línea #elif, si existe, donde se evaluará la expresión-constante-2. Si el resultado es distinto de cero, se procesará la sección-2, y después el control pasa al correspondiente #endif. Por el contrario, si el resultado de la expresión-constante-2 es cero, el control pasa al siguiente #elif, y así sucesivamente, hasta que se encuentre un #else o un #endif. El #else, que es opcional, se usa como una condición alternativa para el caso en que todas la condiciones anteriores resulten falsas. El #endif termina la secuencia condicional.

Cada sección procesada puede contener a su vez directivas condicionales, anidadas hasta cualquier nivel, cada #if debe corresponderse con el #endif más cercano.

El objetivo de una red de este tipo es que sólo una sección, aunque se trate de una sección vacía, sea compilada. Las secciones ignoradas sólo son relevantes para evaluar las condiciones anidadas, es decir asociar cada #if con su #endif.

Las expresiones constantes deben poder ser evaluadas como valores enteros.

* **Directivas #ifdef e #ifndef**

Estas directivas permiten comprobar si un identificador está o no actualmente definido, es decir, si un#define ha sido previamente procesado para el identificador y si sigue definido.

**Sintaxis:**

#ifdef <identificador>

#ifndef <identificador>

La línea:

#ifdef identificador

tiene exactamente el mismo efecto que

#if 1

si el identificador está actualmente definido, y el mismo efecto que

#if 0

si el identificador no está definido.

#ifndef comprueba la no definición de un identificador, así la línea:

#ifndef identificador

tiene el mismo efecto que

#if 0

si el identificador está definido, y el mismo efecto que

#if 1

si el identificador no está definido.

Por lo demás, la sintaxis es la misma que para #if, #elif, #else, y #endif.

Un identificador definido como nulo, se considera definido.

* **Directiva #error**

Esta directiva se suele incluir en sentencias condicionales de preprocesador para detectar condiciones no deseadas durante la compilación. En un funcionamiento normal estas condiciones serán falsas, pero cuando la condición es verdadera, es preferible que el compilador muestre un mensaje de error y detenga la fase de compilación. Para hacer esto se debe introducir esta directiva en una sentencia condicional que detecte el caso no deseado.

**Sintaxis:**

#error mensaje\_de\_error

Esta directiva genera el mensaje:

Error: nombre\_de\_fichero nº\_línea : Error directive: mensaje\_de\_error

Este ejemplo está extraído de uno de los ficheros de cabecera del compilador GCC:

#ifndef BFD\_HOST\_64\_BIT

#error No 64 bit integer type available

#endif /\* ! defined (BFD\_HOST\_64\_BIT) \*/

* **Directiva #include**

La directiva #include, como ya hemos visto, sirve para insertar ficheros externos dentro de nuestro fichero de código fuente. Estos ficheros son conocidos como ficheros incluidos, ficheros de cabecera o "headers".

**Sintaxis:**

#include <nombre de fichero cabecera>

#include "nombre de fichero de cabecera"

#include identificador\_de\_macro

El preprocesador elimina la línea #include y, conceptualmente, la sustituye por el fichero especificado. El tercer caso haya el nombre del fichero como resultado de aplicar la macro.

El código fuente en si no cambia, pero el compilador "ve" el fichero incluido. El emplazamiento del#include puede influir sobre el ámbito y la duración de cualquiera de los identificadores en el interior del fichero incluido.

La diferencia entre escribir el nombre del fichero entre "<>" o """", está en el algoritmo usado para encontrar los ficheros a incluir. En el primer caso el preprocesador buscará en los directorios "include" definidos en el compilador. En el segundo, se buscará primero en el directorio actual, es decir, en el que se encuentre el fichero fuente, si no existe en ese directorio, se trabajará como el primer caso.

Si se proporciona el camino como parte del nombre de fichero, sólo se buscará es el directorio especificado.

* **Directiva #line**

No se usa, se trata de una característica heredada de los primitivos compiladores C.

**Sintaxis:**

#line constante\_entera <"nombre\_de\_fichero">

Esta directiva se usa para sustituir los números de línea en los programas de referencias cruzadas y en mensajes de error. Si el programa consiste en secciones de otros ficheros fuente unidas en un sólo fichero, se usa para sustituir las referencias a esas secciones con los números de línea del fichero original, como si no se hubiera integrado en un único fichero.

La directiva #line indica que la siguiente línea de código proviene de la línea "constante\_entera" del fichero "nombre\_de\_fichero". Una vez que el nombre de fichero ha sido registrado, sucesivas apariciones de la directiva #line relativas al mismo fichero pueden omitir el argumento del nombre.

Las macros serán expandidas en los argumentos de #line del mismo modo que en la directiva #include.

La directiva #line se usó originalmente para utilidades que producían como salida código C, y no para código escrito por personas.

* **Directiva #pragma**

**Sintaxis:**

#pragma nombre-de-directiva

Con esta directiva, cada compilador puede definir sus propias directivas, que no interferirán con las de otros compiladores. Si el compilador no reconoce el nombre-de-directiva, ignorará la línea completa sin producir ningún tipo de error o warning.

Teniendo lo anterior en cuenta, veamos una de las directivas pragma más extendidas en compiladores, pero teniendo en cuenta que no tienen por qué estar disponibles en el compilador que uses.

Tampoco es bueno usar estas directivas, ya que suelen hacer que el código no sea portable, es mejor buscar alternativas.

* **Directiva #pragma pack()**

Esta directiva se usa para cambiar la alienación de bytes cuando se declaran objetos o estructuras.

Recordemos lo que nos pasaba al aplicar el operador sizeof a estructuras con el mismo número y tipo de campos, aunque en distinto orden.

Algunas veces puede ser conveniente alterar el comportamiento predefinido del compilador en lo que respecta al modo de empaquetar los datos en memoria. Por ejemplo, si tenemos que leer un objeto de una estructura y con un alineamiento determinados, deberemos asegurarnos de que nuestro programa almacena esos objetos con la misma estructura y alineamiento.

La directiva #pragma pack() nos permite alterar ese alineamiento a voluntad, indicando como parámetro el valor deseado.

* **Directiva #warning**

**Sintaxis:**

#warning mensaje\_de\_aviso

Sirve para enviar mensajes de aviso cuando se compile un fichero fuente C o C++. Esto nos permite detectar situaciones potencialmente peligrosas y avisar al programador de posibles errores.

**Conclusiones**

* Un lenguaje orientado a objetos como C++ precisa de cierta explicación previa antes de meterse en desarrollos serios, y para aprenderlo que mejor que programar ejemplos.
* El modificador friend puede aplicarse a clases o funciones para inhibir el sistema de protección.
* Las relaciones de "amistad" entre clases son parecidas a las amistades entre personas.
* El preprocesador es una parte fundamental en el proceso de crear programas o ejecutables
* El preprocesador se encarga de procesar las directivas para indicar las instrucciones que debe realizar el programa
* El preprocesador analiza el fichero de fuente antes de la fase de compilación
* C++ surge de fusionar dos ideas: la eficiencia del lenguaje C para poder acceder al hardware al ejecutar tareas que realmente demandaban recursos de memoria; y las ideas de abstracción que representan las el nuevo conceptos de clases y objetos.
* El lenguaje C++ presenta grandes herramientas de desarrollo para los programadores como las funciones, bibliotecas, clases y los objetos. De manera que el programador se ocupa de utilizar dichas herramientas para resolver un problema específico.
* El lenguaje C++ posee una serie de características que lo hacen distinto del lenguaje C. Aunque es posible verlo como una simple extensión del lenguaje C, en realidad implica un cambio en la forma de pensar por parte del programador.

**Bibliografía**

* <http://www.curn.edu.co/documentos/prod_escritural/ORGANIZACION_DEL_TRABAJO_ESCRITO_N_APA.pdf>
* <http://www.colconectada.com/normas-apa/>
* http://es.slideshare.net/SantiMe1/normas-apa-clase
* <http://www.zator.com/Cpp/E4_9_10b.htm>
* Libro Tutorial De C++: o el diario de Peter Class.Por Peter Class y Pello Xabier Altadill Izura