Extreme Guide:

Usando C++ en Taller de Programación Orientada a Objetos

Be free… I'm free, I use Linux… Close your Windows, open your mind!!!

Extreme Guide… no more else!!!

Editorial Mi-K-sita
1ª Edición (en español)
Ejemplares impresos: 1

La presente guía, **Extreme Guide:** *Usando C++ en Taller de Programación Orientada a Objetos*, no es más que eso: una guía práctica para el seguimiento de la materia *Taller de Programación Orientada a Objetos*, dictada en las aulas de la Extensión Región Centro Sur – Anaco de la Universidad de Oriente.

Se sobreentenderá que ya usted cuenta con un lenguaje de programación denominado **C++Builder** (de la Borland), instalado en el computador en el cual realizará las prácticas recomendadas en esta guía. En caso de no contar con esto se le recomienda su adquisición e instalación a la brevedad posible, ya que sin él no podrá seguir con la utilización de la Extreme Guide.

Acá se pretende, en un principio, llevar de la mano al estudiante en las primeras prácticas que se deberán llevar a cabo en el Laboratorio de Computadores (o en la comodidad de su hogar también, por que no).

En el primer capítulo se habla de cómo iniciar la aplicación C++Builder, crear un proyecto nuevo, escribir nuestro primer programa, compilarlo, ejecutarlo, guardarlo, cerrarlo y abrir un proyecto ya existente.

El capítulo siguiente, el dos, nos llevará a cómo hacer un programa que en realidad haga algo por nosotros que, en este caso, será un programa que nos resuelva una ecuación de segundo grado. Estaremos aplicando el uso de instrucciones de bifurcación, asignación de valores a variables, cálculos mediante operadores y funciones matemáticas (incluyendo las respectivas librerías), lectura de valores desde el teclado y la escritura de valores al monitor del computador.

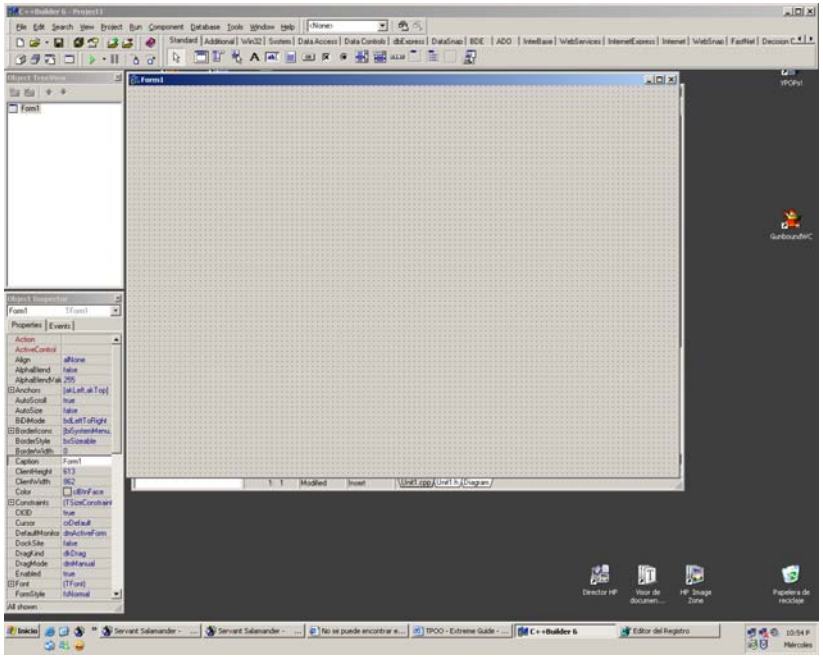
Los subsiguientes capítulos están enfocados a las actividades que deberá hacer usted (en este caso, las asignaciones semanales), para entregárselas al profesor de su materia, para la posterior evaluación. Deberá utilizar, para la resolución de dichas actividades, sus conocimientos adquiridos en la materia teórica, denominada **Programación Orientada a Objetos**.

Bueno, no queda más que encender la computadora y empezar a trabajar, esperando que la presente les sirva de ayuda.

1. Iniciando C++Builder

Para trabajar, o programar, en el entorno denominado C++Builder debemos empezar por iniciar dicha aplicación. Para esto sigamos los siguientes pasos:

- Hacemos clic en el botón de *Inicio*, en Windows.
- Ubicamos, dentro del menú *Programas*, el ítem denominado *Borland C++Builder 6* (en este caso el 6 indica la versión de la aplicación).
- Al abrir dicho menú se abrirá otro submenú, en el cual debemos escoger la opción *C++Builder 6*. Al momento de escoger dicha opción aparecerá en pantalla un splash de la aplicación. Paciencia, deberemos esperar unos momentos hasta que se cargue por completo. Esto lo sabremos cuando en la pantalla aparece algo como:

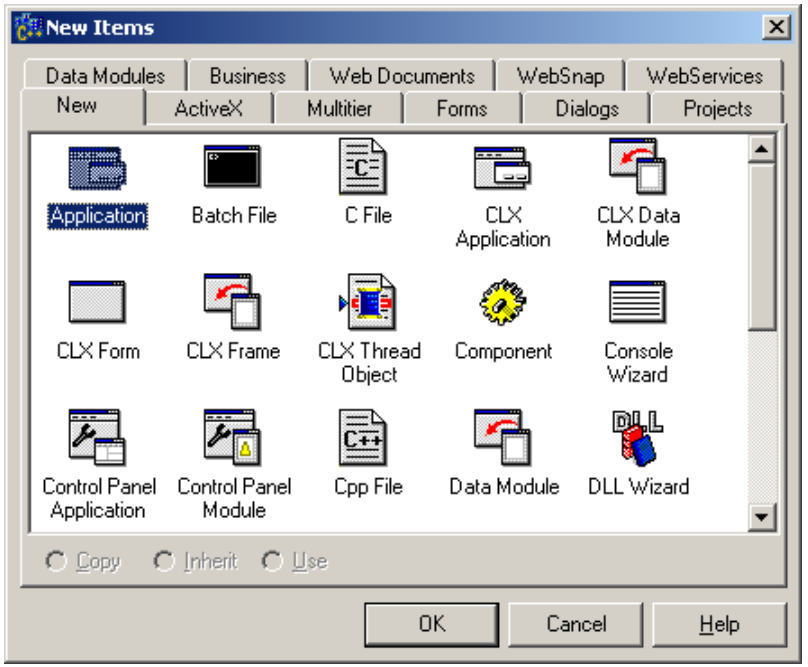


- En este momento, C++Builder (desde ahora simplemente lo llamaremos **CPP** por comodidad y por la flojera de escribir que tengo en estos momentos… discúlpenme… gracias) tiene un proyecto listo para empezar a utilizar.

1.1. Modo Consola

Lo malo de los pasos anteriores (y eso es algo que no se le puede cambiar a esta versión del CPP) es que siempre, cuando lo iniciamos, despliega un proyecto nuevo pero para un programa en Modo Windows, y nosotros lo que queremos es un proyecto en Modo Consola. Para conseguir esto hagamos lo siguientes:

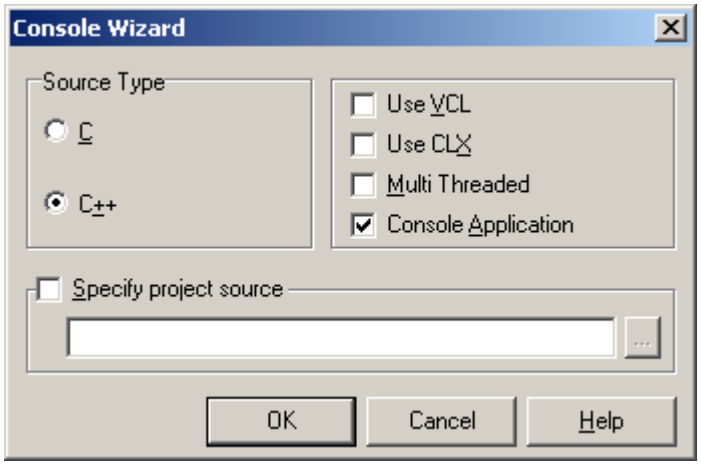
- Con el CPP abierto (ver figura anterior), activamos el menú *File*.
- Una vez escogida dicha opción, se desplegará un submenú, del cual escogeremos la opción que dice *Close All*. Llegado a este punto, lo que CPP hará será cerrar todas las ventanas que tiene abierta del lado derecho, y dejará las del lado izquierdo inactivas.
- A continuación, volvemos a activar el menú *File*, y de él escogeremos ahora la opción *New*. En este punto le estaremos diciendo a CPP que queremos crear algo nuevo, pero aún no le hemos dicho qué.
- Se abrirá un nuevo submenú, del cual escogeremos la opción *Other*, ya que de las opciones mostradas en el submenú por defecto no nos interesa ninguna. Acá se desplegará una ventana de diálogo parecida a la de la siguiente figura:



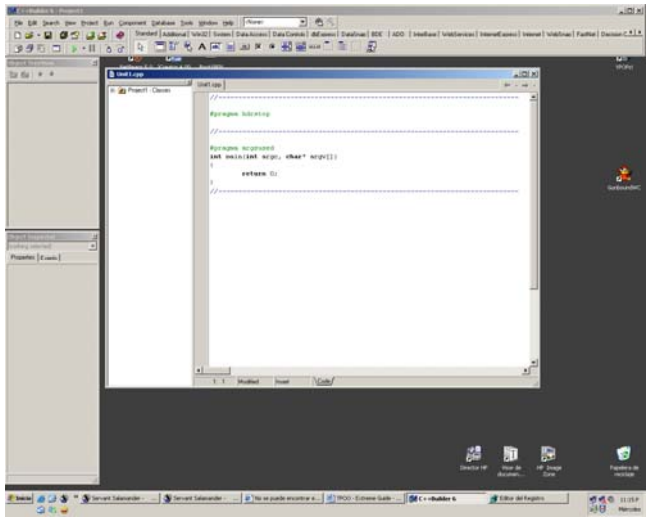
- De todos los "dibujitos" (que se denominan iconos), escogeremos el que debajo de él indica *Console Wizard* (que en la figura anterior se muestra al final de la segunda fila de arriba hacia abajo, o el segundo de la quinta columna de izquierda a derecha). En este punto ya CPP conoce que lo que queremos hacer es crear un nuevo proyecto en Modo Consola.

> **NOTA:** La diferencia de una aplicación en Modo Consola con una estándar, denominada Modo Windows, radica en que la primera será una aplicación o programa en el cual solo tendremos una interfase con el usuario de puro texto, y el modo Windows son aquellos programas en los cuales, como el mismo CPP, podremos hacer uso de dibujos, gráficos, ventanas a colores, manejo del ratón, botones, cajas de texto, etc.

- Haga doble clic sobre el icono que se menciona en el apartado anterior. Al hacerlo, CPP desplegará otra ventana de diálogo (ver siguiente figura), en la cual deberemos estar seguro de que en el grupo de la izquierda (Source Type) esté marcada la opción *C++*, y en el grupo de la derecha sólo deberá estar marcada la opción *Console Application*. En el grupo inferior (*Specify Project source*) lo podremos dejar tal cual está (en blanco y sin activar).

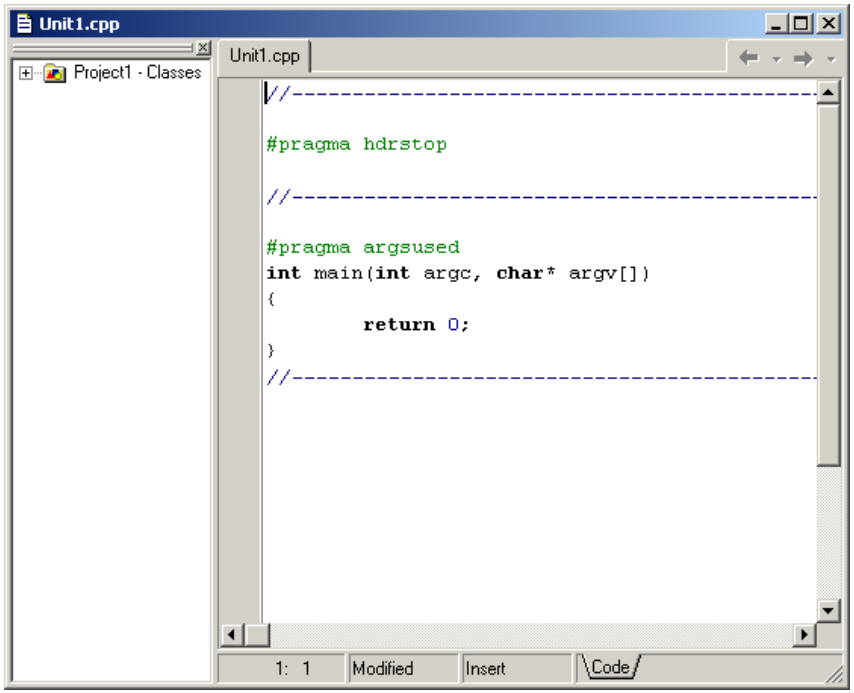


- Pulsemos sobre el botón de OK, ubicado abajo a la izquierda de dicha ventana. Esperemos unos momentos. Al finalizar, en la pantalla deberá desaparecer las dos ventanas de diálogo que abrimos anteriormente y se abrirá una ventana grande del lado derecho de la pantalla. Vea la siguiente figura para una idea de cómo se debería ver esto una ver cargado el nuevo proyecto.



- Ya estamos listos para empezar a escribir nuestro programa en modo consola. Acabamos de finalizar la creación de un nuevo proyecto. Lo que nos queda ahora es empezar a escribir nuestro código fuente en la ventana que se nos muestra a la derecha del monitor (conocida esta como editor de código fuente). Allí es en donde estaremos ocupados la mayor parte de nuestro tiempo como programadores.

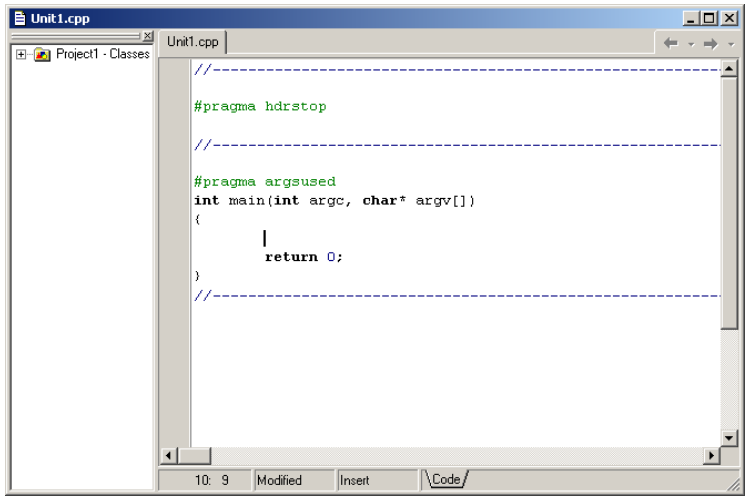
> **NOTA:** En la ventana del editor de código fuente veremos que existen palabras escritas en diversos formatos y estilos. Por ejemplo, hay algunas líneas en que las palabras aparecen en color verde (por ejemplo, las instrucciones del preprocesador), unas azules (los comentarios y los literales), otras en negrita (por ejemplo, las palabras reservadas por el CPP) y otras normales (por ejemplo, el nombre de las variables que declaremos, comandos y funciones tanto propias como de librerías). Esto no es más que una ayuda visual para los programadores, para que a simple vista se conozcan cuales de esas líneas de código son comentarios, palabras reservadas, literales, etc.



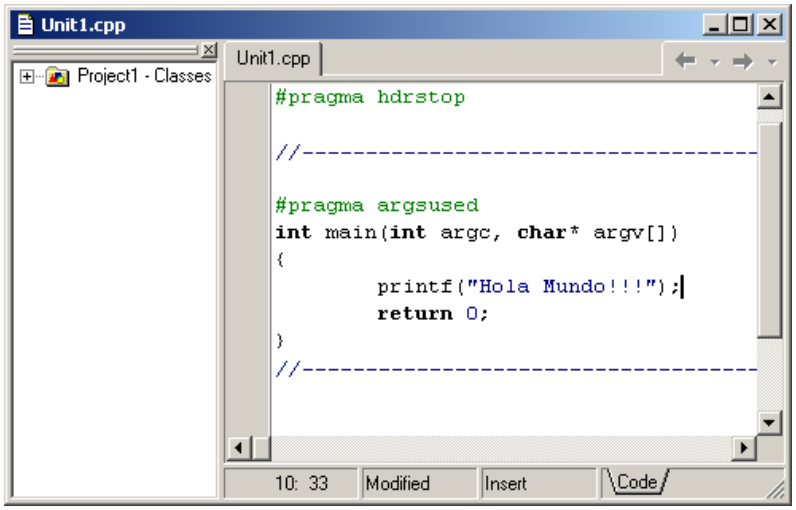
1.2. Nuestro primer programa… Hola Mundo!!!

Bien, ahora que sabemos como crear un proyecto nuevo en CPP, hagamos un simple programa: el famoso Hola Mundo. Este programa lo único que hará será escribir por la pantalla del computador la frase "Hola Mundo!!!". Aprenderemos en este apartado a como empezar a escribir nuestro código, a compilar un programa, depurar algún error de tipo léxico y finalmente a ejecutarlo. Para esto, prosigamos en donde quedamos en el apartado anterior) y haremos lo siguiente:

- Ubiquemos el cursor de edición (se verá sobre el editor como una línea semigruesa, de manera vertical y en constante parpadeo) dentro del cuerpo de la función principal (o sea, dentro de la función **main**, específicamente después de la llave que abre dicha función).
- A continuación, pulsaremos la tecla **ENTER**. Esto lo haremos con el propósito de dejar una línea entre la llave que abre el cuerpo de la función y lo que empezaremos a escribir dentro de ella:

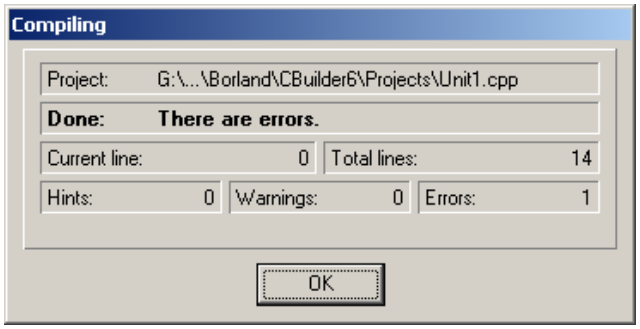


- Ahora escribimos nuestro código, el cual consistirá en utilizar la función **printf**, que, como ya sabemos, es una de las funciones que nos permite escribir en el monitor. Como argumento utilizaremos el literal de cadena "Hola Mundo!!!", quedando algo como:

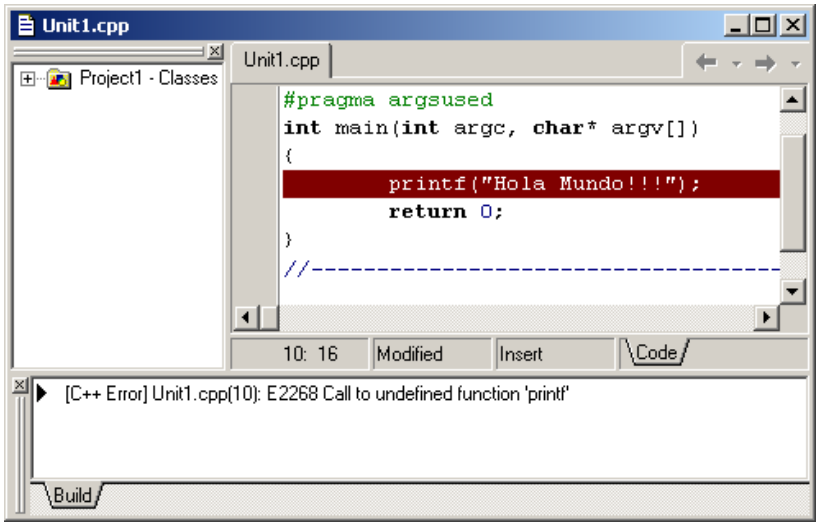


- Como eso es lo único que queremos que haga el programa, lo dejaremos así y procederemos a **compilarlo**. Para esto deberemos seleccionar del menú principal la opción *Project* y de allí, cuando se

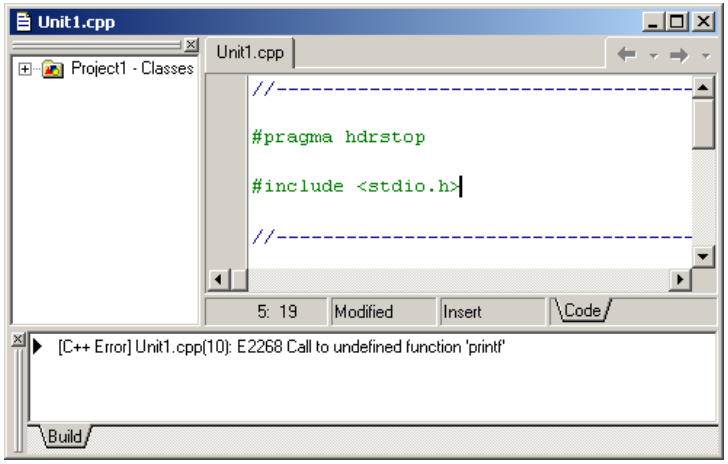
abra el submenú, seleccionar el ítem *Compile Unit* (o pulsar simultáneamente las teclas *Alt-F9*). En caso de existir errores sintácticos y/o léxicos aparecerá una ventana emergente en la cual nos indicará cuántos de ellos hay. En este caso SI hay un error, por lo que CPP mostrará una ventana como la siguiente (en ella se ve claramente que nos indica que existe un error):



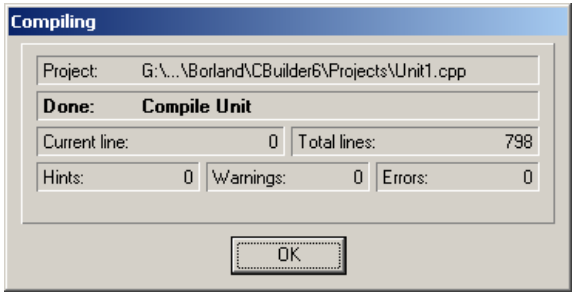
- Pulsemos sobre el botón OK. Al hacerlo, la ventana que nos indicaba la cantidad de errores encontrados se cerrará para darle paso de nuevo a la ventana del editor, sobre la cual podremos ver la línea de código en donde consiguió el error sombreada en rojo, y debajo de dicho editor se muestra una subventana en la cual nos mostrará la lista de errores encontrados:



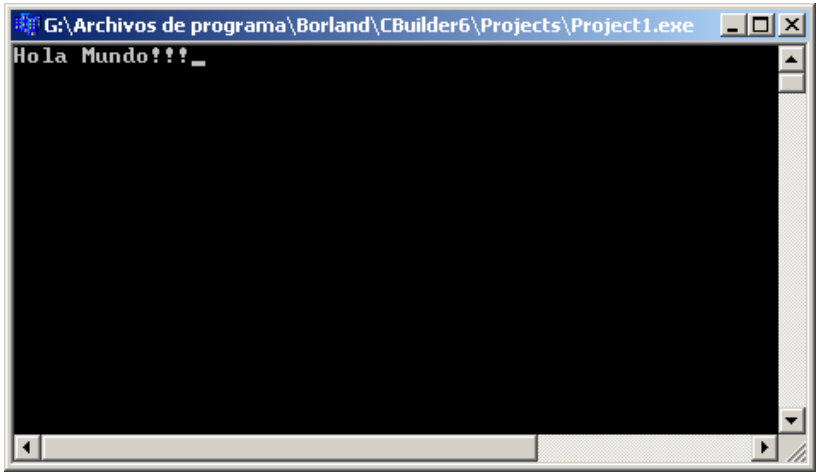
- De querer inspeccionar cualquiera de los errores mostrados en dicha subventana, sólo tendremos que hacer doble clic con el ratón sobre la línea del error para que CPP ubique el cursor sobre el código fuente que lo contiene. En este caso, el único error que nos muestra nos dice que **E2268 Call to undefined function 'printf'**. La primera palabra (E2268) es el código interno de CPP para dicho error, y lo siguiente es una breve descripción del mismo. Como conocemos ya un poco de inglés, sabemos que en la descripción nos indica que se está haciendo uso de la función *printf*, la cual no tiene el prototipo. Esto ya sabemos como arreglarlo ¿verdad? Correcto.
- Ubiquemos el cursor ahora a la línea siguiente después de la instrucción del preprocesador **#pragma hdrstop**. Allí deberemos incluir nuestra librería **stdio.h**, quedando algo como:



- Si ahora volvemos a compilar el programa, la ventana emergente nos debería aparecer de la siguiente manera:



- Como vemos, no hay errores. Pulsemos sobre el botón OK de la ventana y la subventana que anteriormente nos mostraba los errores se deberá desaparecer.
- Una vez que no tengamos errores en nuestro programa, podremos **ejecutarlo**. Para esto seleccionamos en el menú la opción *Run* y en ella el ítem con el mismo nombre (o pulsamos la tecla *F9*). Al hacerlo, CPP por defecto tratará de compilar de nuevo el programa, verificando que no haya habido cambios desde la última vez que se compiló hasta ese momento. Al no haber errores, pasará a ejecutar el programita, dando como salida la siguiente pantalla:

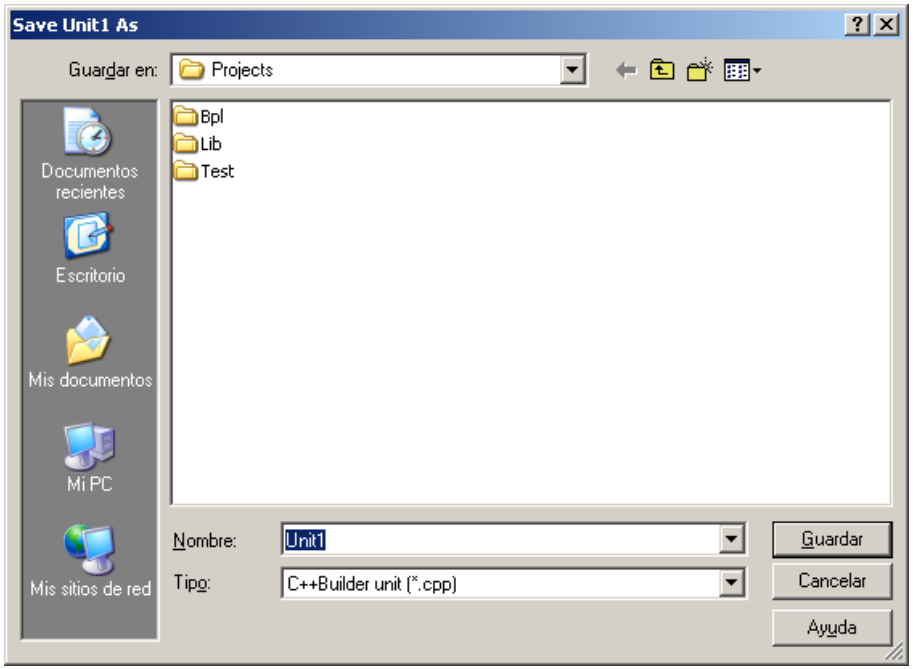


- Como vemos, aparecerá una ventana con el fondo negro (consola) y en ella la frase "Hola Mundo!!!", que fue lo que nosotros escribimos como argumento en la función *printf*. La línea o raya horizontal al final de la línea viene siendo el cursor de la consola.

1.3. Hicimos algo… ¿lo perderemos?

Bien, ahora que ya hemos terminado nuestro primer programa, podremos proceder a **guardarlo**. Esto significa que lo estaremos archivando en algún dispositivo de almacenamiento físico (por ejemplo, el disco duro, un diskette, un pendrive, etc). Esto no significa que para guardar un proyecto se deberá compilar y ejecutar primero. Esto lo podremos hacer en cualquier momento mientras estemos desarrollando el programa. De hecho, se sugiere que por lo menos cada tres minutos se guarde lo que se lleve hecho a manera de resguardar la información y evitar el peligro de perderla en caso de una falla eléctrica, o que apaguemos el computador sin darnos cuenta, etc.

Para esto, lo que debemos hacer es seleccionar del menú principal la opción *File* y de allí el ítem *Save All* (como es la primera vez que se a guardar ESTE proyecto, deberemos guardar tanto el archivo del programa como tal y el archivo del proyecto. Posteriormente lo único que estaremos guardando serán los archivos con los programas). Se abrirá una ventana de diálogo como la mostrada a continuación, en donde se nos pregunta en donde queremos guardarlo y con que nombre (el asume un nombre por defecto, que es este caso es *Unit1.cpp*).

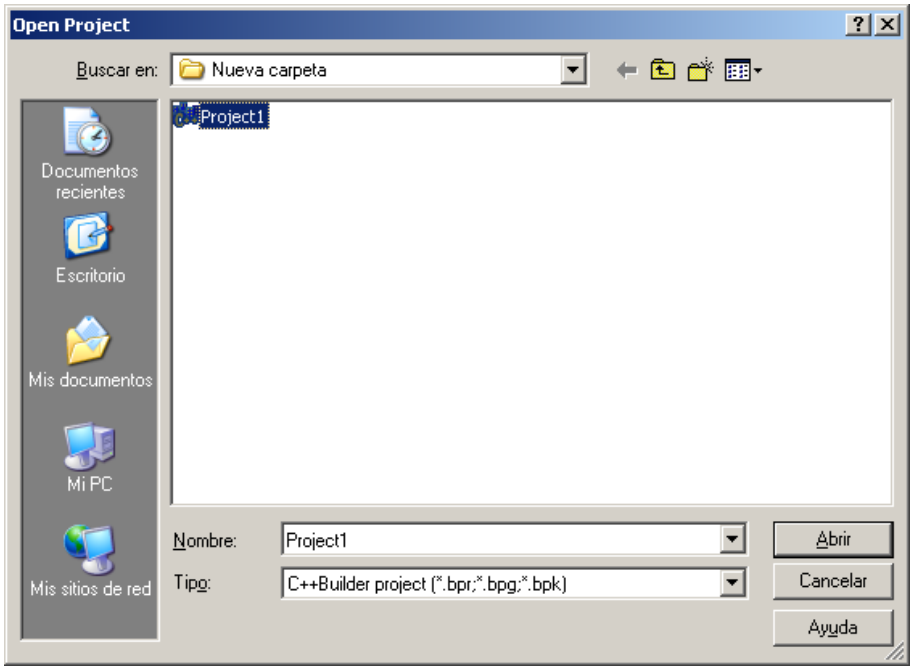


Luego de seleccionar en donde lo queremos guardar, debemos pulsar sobre el botón *Guardar*. Lo mismo ocurrirá para el archivo del proyecto, el cual el asume con el nombre de *Project1*.

Una vez que hayamos guardado todo, podremos sin cerrar CPP sin peligro de pérdida de nuestro código. Para esto seleccionaremos la opción *File* del menú y a continuación el ítem *Exit*. En caso de que tengamos algún

proyecto abierto y modificado, CPP se dará cuenta de ello y nos avisará, antes de cerrarse, de dicha condición, para lo cual deberemos responderle ante un aviso que él nos hará mediante una ventana emergente, si lo que queremos hacer es: guardar los archivos modificados antes de que CPP se cierre, o que se cierre perdiendo con ello cualquier cambio que hayamos hecho o cancelar la acción y quedarnos con el CPP abierto sin cerrase.

Ahora bien, si alguna vez queremos volver a abrir nuestro proyecto, lo que debemos hacer será iniciar el CPP, y a continuación seleccionar desde el menú principal la opción *File* y luego el ítem *Open Project*. Al hacer esto se abrirá un ventana de diálogo en la cual nos pide que le indiquemos donde y cual es el archivo del proyecto que queremos abrir. Ubicaremos entonces nuestro proyecto y a continuación le damos al botón *Abrir*. Listo, ya tenemos el proyecto listo para ser modificado una vez más.



Terminado todo esto, ya seremos capaces de crear un proyecto (modo consola), editar algo de código, compilar el código fuente, depurar algún error, ejecutar un programa, guardar el proyecto conjuntamente con los archivos de los programas que lo conforman, cerrar el entorno de programación y finalmente cerrar el CPP.

NOTA: Es de hacer saber que, en cualquier momento, estando el cursor ubicado en la ventana del editor, podremos hacer uso de los archivos de Ayuda que nos brinda el CPP. Para activarlo, basta con presionar la tecla *F1*. Al hacerlo, se abrirá una ventana desde la cual podremos buscar y ver todos los comandos con los que cuenta el C++, así como también algunos ejemplos sencillos de su uso y en cual librería lo podremos encontrar. Se recomienda el uso prolífico de dicha ayuda.

2. Resolución de ecuaciones cuadráticas

Ahora que sabemos como crear proyectos, compilarlos, ejecutarlos y guardarlos, pasemos entonces a realizar un programa que en realidad haga algo útil para nosotros.

El programa que haremos a continuación será algo sencillo, pero práctico: encontrar las raíces de una ecuación cuadrática (o de segundo grado). Para esto vamos a hacer uso algunas de las instrucciones que ya conocemos, como lo son el *printf*, el *scanf* y una función de la librería matemática *sqrt*.

2.1. Definición del problema

Se requiere hacer un programa en el cual se resuelva una ecuación cualquiera de segundo grado. Para esto, debemos encontrar las raíces de dicha función.

2.2. Solución

Sabemos que las ecuaciones de segundo grado vienen expresadas de la siguiente manera:

$$a x^2 + b x + c = 0$$

También sabemos que la solución a dicha ecuación viene dada por:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Por lo cual podemos deducir que de la ecuación anterior se obtienen dos posibles soluciones. Los valores tanto de *a*, como de *b* y *c* son conocidos. Se tratará de encontrar los dos valores para *x*. Estos valores podrán tener, como en la vida real, cantidades decimales. Por esto es que vamos a utilizar, como tipo de variables, el *double*, ya que este posee un amplio rango de valores y son más precisos que los *float*.

Habiendo ya analizado básicamente el problema, hagamos entonces el programa para resolverlo. Sigamos los siguientes pasos:

- Inicie el entorno de programación del CPP.
- Posteriormente, cree un nuevo proyecto como ya lo sabe hacer.
- Antes de empezar a escribir código, guardemos el proyecto bajo el nombre de *EcuCuad*, y el archivo del programa con el nombre de *CPrincipal*.
- Ahora ubiquemos el cursor en la ventana de edición para empezar a escribir lo siguiente (que será la declaración de las variables que pensamos vamos a estar utilizando), después de la llave que abre la función *main*:

```
// declaramos tres variables, las cuales nos representarán
// las tres constantes en la ecuación cuadrática
double a, b, c;

// ahora, declaramos las dos variables en las cuales se
// guardarán los resultados
double x1, x2;
```

- Seguidamente escribiremos el código necesario para poder permitirle a los posibles usuarios del programa ingresar los valores de las constantes de la ecuación, ya que en un principio estos valores podrán ser cualesquiera y no son conocidos por nosotros. De esta manera hacemos un programa "genérico" que nos permita poder resolver cualquier ecuación cuadrática.

```
// escribimos en pantalla una solicitud de que se ingrese
// el valor de 'a', para que el usuario sepa que es lo que
// se le está pidiendo de entrada
printf("Ingrese el valor de a: ");

// ahora leeremos el valor que el usuario haya ingresado
scanf("%lf", &a);

// repitamos los mismos pasos para las otras dos variables
// de entrada

printf("Ingrese el valor de b: ");
scanf("%lf", &b);
printf("Ingrese el valor de c: ");
scanf("%lf", &c);

// debemos saber que, para hacer uso de la instrucción
// 'scanf', debemos utilizar el operador de dirección para
// las variables, de esta manera scanf sabrá en donde
// guardar los valores leídos por el teclado
```

- Llegado a este punto, ya tenemos en memoria los valores de las constantes de la ecuación a resolver. Ahora debemos utilizar dichos valores en la implementación de la ecuación que la resuelve y listo. Veamos como sería:

```
// ahora, resolvamos la ecuación...
x1 = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
x2 = (-b - sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
```

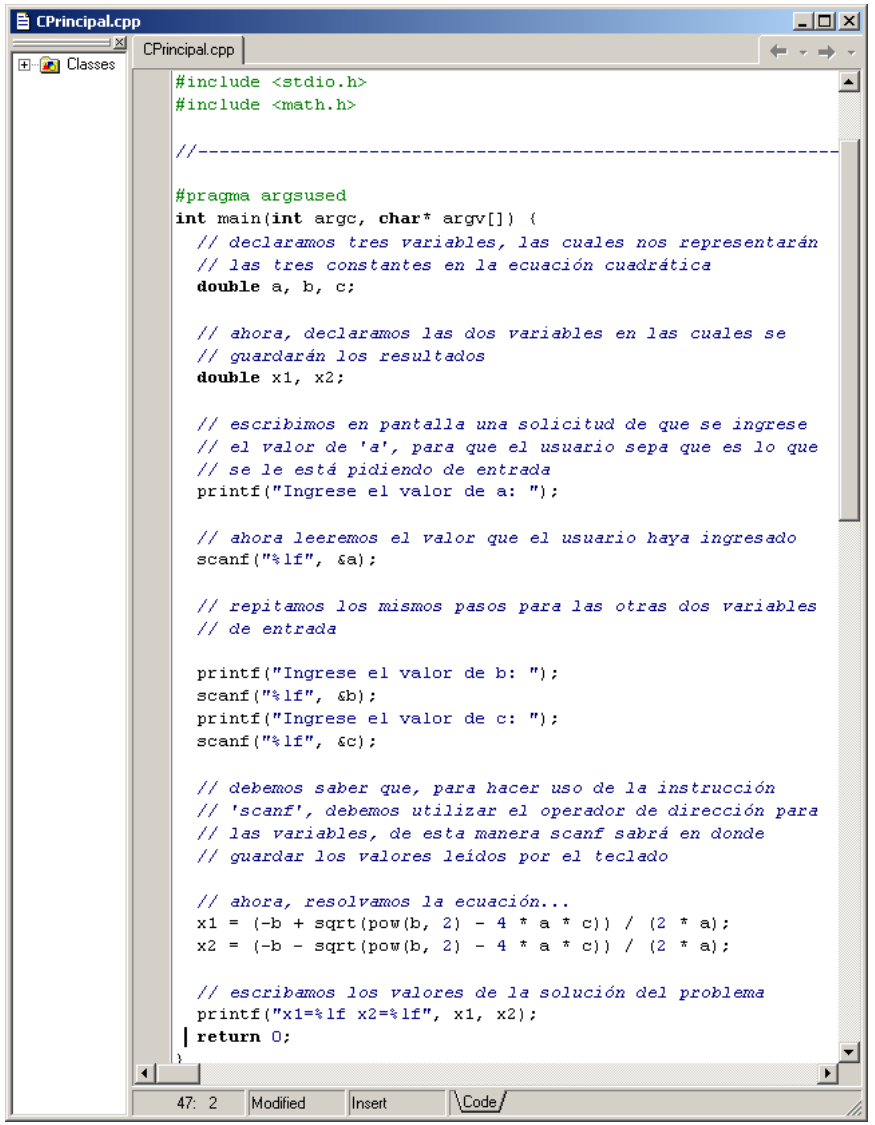
- Hemos utilizado las funciones *sqrt* y *pow*. La primera de ellas nos devuelve la raíz cuadrada del valor que le enviemos como argumento (tanto el argumento como lo que la función devuelva se esperan sean de tipo *double*). La segunda función que utilizamos es para calcular una potencia. Se le envían como argumentos el valor de la base y, separado con una coma, el valor del exponente. Esta función, al igual que la anterior, también esperan que tanto sus argumentos como el valor retornado sean del tipo *double*. Ahora bien, una vez calculadas las raíces de la ecuación procederemos a dárselas a conocer al usuario del programa. Para ello escribamos lo siguiente (después de las líneas que escribimos anteriormente):

```
// escribamos los valores de la solución del problema
printf("x1=%lf x2=%lf", x1, x2);
```

- Ahora, si nos atrevemos a compilar el programa, nos daremos cuenta de que vamos a obtener cuatro errores. Esto, como ya lo sabemos, viene de la falta de las declaraciones de las librerías para las funciones que queremos utilizar (printf, scanf, sqrt y pow). Entonces, ubiquemos el cursor al comienzo del programa, y debajo de la instrucción del preprocesador **#pragma hdrstop** escribamos:

```
#include <stdio.h>
#include <math.h>
```

- Ahora si, si compilamos el programa veremos con satisfacción que este no generará errores. Lo que nos queda en estos momentos es ejecutar el programa que acabamos de hacer y validar sus resultados. No nos olvidemos de ir guardando el programa cada cierto tiempo para evitar pérdidas mayores en nuestro código fuente.
- En la siguiente figura se muestra como ha de quedar aproximadamente el código fuente de este programa que acabamos de hacer.



- Si ejecutáramos este programa y los valores ingresados fuesen de tal manera que *b* al cuadrado sea menor que el producto de *4ac* los resultados obtenidos no serán los esperados, ya que nos encontraremos con un problema matemático: raíces cuadradas de números negativos. Para resolver dicho problema podemos modificar nuestro código para

que valide el argumento de la raíz antes que esta sea calculada. Para esto hagamos lo siguiente. Declararemos una nueva variable de tipo *double*. Aunque esto lo podemos hacer en cualquier parte del programa, se prefiere que se haga al inicio de las funciones y/o métodos. De esta manera se tiene mayor control y conocimientos de las variables que se utilizan dentro de ellas. Coloquemos entonces la declaración de esta variable debajo de las que ya tenemos en el programa:

```
// variable temporal para cálculos intermedios
double temp;
```

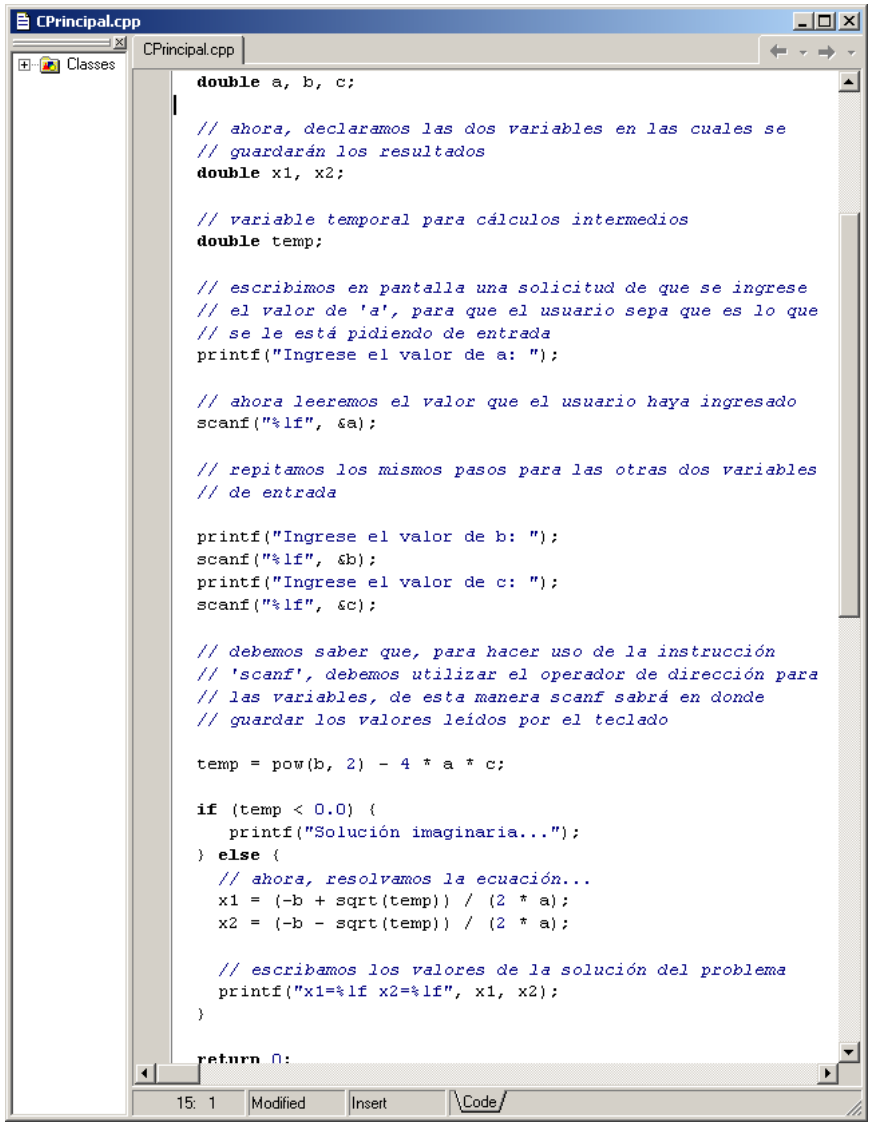
- Ahora sustituiremos las dos líneas del cálculo de las variables x1 y x2 con el siguiente código, con la finalidad de conocer si el valor del argumento de la raíz es negativa o no. En caso de ser negativa escribimos un mensaje por pantalla indicando que la solución es imaginaria, sino, hacemos el cálculo normal y mostramos los resultados:

```
temp = pow(b, 2) - 4 * a * c;

if (temp < 0.0) {
  printf("Solución imaginaria...");
} else {
  // ahora, resolvamos la ecuación...
  x1 = (-b + sqrt(temp)) / (2 * a);
  x2 = (-b - sqrt(temp)) / (2 * a);

  // escribamos los valores de la solución del problema
  printf("x1=%lf x2=%lf", x1, x2);
}
```

- Luego de los cambios propuestos en los ítems anteriores, pasemos a compilar el programa. No debería generar ningún tipo de errores. En caso de que los hubiese, vuelva atrás y revise paso a paso lo que se debe hacer. En la siguiente figura podremos ver gran parte de cómo debería quedar el código fuente de nuestro pequeño programa con las modificaciones introducidas en el apartado anterior. Luego de compilar el programa y validar de que no tenga errores, pase a ejecutarlo. Ingrese valores, en la ejecución, de tal modo que *b* al cuadrado sea menor que el producto de *4ac*. Notará ahora la diferencia entre el resultado que se muestra ahora con los mostrados por el programa antes de las modificaciones, ya que ahora nos mostrará una cadena literal que nos indica que la solución es imaginaria.



3. Y ahora, ¿qué sigue?

Una vez adquirida ya la lógica de programación, que aunada a la lógica inducida en materias anteriores, podremos ir desarrollando cada vez más y mejores programas y de mayores niveles de complejidad.

Lo que nos queda a continuación es ir practicando (acuérdese de que la práctica hace al maestro) e ir desarrollando y entregando las asignaciones en las fechas previstas, sin olvidar que nos pasemos de dicha fecha de entrega, por cada semana que se dejen de entregar se penalizará con dos (2) puntos menos del total de la asignación.

También no nos podremos olvidar de que, en el momento en que el profesor lo requiera, usted tendrá el deber de defender su asignación frente a él, lo cual se tomará en cuenta a la hora de la evaluación.

3.1. Cronograma de entrega

El profesor hará entrega, en el aula de clases, del cronograma de recepción de las asignaciones que le corresponden. Dichas asignaciones se encuentran enumeradas a continuación (de la lista, el profesor indicará cuáles son los que se deberán entregar, el resto quedará como ejercicios propuestos), esto con el fin de que puedan ir trabajando y/o adelantando sobre ellas a medida de que las clases teóricas se vayan dictando y acercándose las fechas de entrega de las mismas.

Acuérdese de que las asignaciones deberán ser entregadas en algún medio o dispositivo de almacenamiento físico, asegurándose de que estos se puedan leer, y sin problemas, tanto los archivos del proyecto como el proyecto en si. También deberá asegurarse de que estos medios no tengan virus ni troyanos.

Se recomienda así mismo que cuando trabajen sobre un proyecto nuevo, este, y los archivos del programa, se encuentren contenidos en una misma carpeta aparte, para que de esta manera cuando lo quieran copiar en algún otro medio sólo tengan que copiar la carpeta completa.

El código fuente del programa deberá estar debidamente identificado y documentado (hacer uso de los comentarios para esto), con al menos los nombres y cédulas de los integrantes de la asignación, así como el número de la asignación y fecha de entrega del mismo. Cualquier otra información adicional que quieran colocar dentro del mismo, como por ejemplo una breve descripción de lo que hace el programa y los procesos que se llevan a cabo, serán tomados en consideración.

3.2. Asignaciones

A continuación se muestran las posibles asignaciones que se deberán entregar semanalmente. Queda sobreentendido que dichas asignaciones deberán ser hechas en C++, y en todos los casos, los resultados deberán ser mostrados por la pantalla:

3.2.1. Realice un programa en el cual se le dé solución a una ecuación de segundo grado, tomándose en cuenta también las soluciones imaginarias (que ocurre cuando el argumento de las raíces cuadradas son negativas).

3.2.2. Leer 2 números enteros positivos y determinar el Mínimo Común Múltiplo entre ellos dos.

3.2.3. Hacer un programa que permita calcular las 5 primeras parejas de números primos gemelos (dos números son primos gemelos si, además de ser primos, la diferencia entre ellos es exactamente igual a 2) a partir de un número entero cualquiera ingresado por el usuario.

3.2.4. Diseñar un programa que permita calcular los 5 primeros números perfectos (un numero es perfecto cuando la suma de sus divisores, sin incluirlo, es exactamente el mismo número. Por ejemplo, el 6 es un número perfecto, porque sus divisores son 1, 2 y 3), a partir de un número entero positivo cualquiera ingresado por el usuario.

3.2.5. Dada una figura cuadrilátera cualquiera, leer los valores de los cuatro lados (iniciando desde el lado izquierdo y siguiendo el sentido contra reloj), e indicar si dicha figura forma un cuadrado, un rectángulo o simplemente es una figura inválida.

3.2.6. Realizar un programa que lea un número natural y lo convierta en números romanos.

3.2.7. Leer 4 notas (del 0 al 10) de exámenes prácticos y 3 notas de exámenes parciales. Luego, indique cuál es el promedio de los prácticos y el promedio de los parciales. Además, mostrar un mensaje si el estudiante tiene derecho de asistir al examen final, de reparación o a ninguno de estos. NOTA: el estudiante tendrá derecho al examen final si el 20% del promedio de los prácticos más el 50% del promedio de los parciales da un valor mayor o igual

a 3,5. Así mismo, tendrá derecho de asistir al examen de reparación si dicha suma anterior da como resultado un valor mayor o igual que 2,0.

3.2.8. Realice un programa que lea por consola una oración y una palabra cualesquiera. El programa deberá dar como resultado cuantas veces aparece la palabra ingresada en la oración. Ejemplo:

Oración: Esta casa es solo mía y es de mamá
Palabra: es
Resultado: Aparece 2 veces

3.2.9. Realice un programa que lea por la consola una oración y tres palabras. Luego deberá buscar, dentro de la oración, la primera palabra leída. Si se consigue, se deberá sustituir por la segunda palabra ingresada. En caso de que aparezca una segunda o más veces, se deberán sustituir por la tercera palabra ingresada por el usuario. Ejemplo:

Oración: esta materia esta cada vez más difícil, pero esta vez…
Palabra 1: esta
Palabra 2: la
Palabra 3: es
Resultado: la materia es cada vez más difícil, pero es vez…

3.2.10. Realice un programa que lea por consola una cadena de tipo "123+567" y efectuar la operación que indique la misma. En caso de que el usuario ingrese una cadena no válida, deberá mostrarse un mensaje de error indicándolo. Una cadena es válida si contiene dos operandos y en medio de estos un operador de los siguientes: suma (+), resta (-), multiplicación (*) o división (/). El programa deberá terminar su ejecución sólo cuando el usuario ingrese una cadena vacía.

3.2.11. Realice un programa que lea por la consola un párrafo y que de cómo resultado otro párrafo, con las mismas palabras, pero invertidas. O sea, la primera palabra será la última, la segunda palabra será la penúltima y así sucesivamente. Ejemplo:

Ingrese párrafo: esta casa es solamente mía
Resultado: mía solamente es casa esta

3.2.12. Hacer un programa que consistirá en situar ocho fichas en un tablero de ajedrez, de forma tal que ninguna de ellas se crucen con otra cualquiera (o sea, que no se encuentren vertical, horizontal ni diagonalmente).

3.2.13. Se desea realizar un programa que realice las siguientes tareas, en momentos independientes: a) Leer una lista de números enteros b) Visualizar dichos números c) Preguntar al usuario si desea ordenar la lista de forma creciente o decreciente.

3.2.14. Diseñar un programa que calcule y almacene en un arreglo los 15 primeros números pares anteriores a un número entero dado por el usuario N, en donde N deberá ser mayor a 51 y menor que 100.

3.2.15. Realizar un programa que lea los datos de 15 trabajadores (Nombre, Horas laboradas y el Costo de cada hora). Luego, calcule por cada uno de ellos lo siguiente: a) Sueldo bruto b) Sueldo neto (sueldo bruto menos 5% de deducciones) c) Imprimir el sueldo bruto y el neto.

3.2.16. Dado un arreglo de N elementos, hallar la media, la moda y su frecuencia.

3.2.17. Hacer un programa que permita el ingreso de 20 registros ($Nombre_X$, $Telefono_X$, $Cedula_X$). Luego, dado un número de cédula cualquiera, mostrar teléfono y nombre de la misma.

4. Al final, un ejercicio completo, paso a paso

Hagamos a continuación un ejercicio, aplicando todos los conceptos de POO aprendidos. Para esto, vamos a ir desarrollando el problema que se plantea en varias fases, para irle incorporando paso a paso todos los elementos aprendidos a los largo del semestre.

4.1. Creación de la clase principal y el método de inicio

En este primer problema abarcaremos lo que es la declaración la función de entrada a la aplicación (la función especial llamada *main*), y la asignación de valores primitivos a variables declaradas dentro del mismo.

El problema a resolver es el siguiente: Escribir un programa, utilizando para esto el lenguaje de programación C++, en el cual se lea el ancho y el alto de un triángulo:

```
// empezamos por incluir las librerías necesarias
#include <stdio.h>

// ahora, declaramos lo que será nuestra función
// principal
void main(/* no importa el argumento es opcional */) {
  // declaramos dos variables de tipo double, en las
  // cuales se almacenarán tanto el alto como el ancho
  // de nuestro triángulo
  double alto;
  double ancho;
  // nótese que son de tipo double ya que estos valores
  // podrían contener números con decimales

  // escribimos un literal de string en el cual se avisa
  // al usuario que se va a leer el alto
  printf("Ingrese el alto.: ");

  // se procede a leer el valor de la variable 'alto'
  scanf("%lf", &alto);

  // hacer lo mismo, ahora con la variable de 'ancho'
  printf("Ingrese el ancho: ");
  scanf("%lf", &ancho);
}
```

Como se ve, fue sencillamente fácil hacer esto. Se ve entonces cómo se utilizó la función *scanf* para guardar los valores introducidos por el usuario a través de lo que escriba por el teclado (siendo este la entrada estándar de un computador). Ahora, ¿que tal si ahora le empezamos a colocar restricciones al programa?

4.2. Uso de la instrucción condicional *if*

Tomemos el problema anterior, pero ahora deberemos validar que ambos valores (ancho y alto) deban ser números positivos, ya que no es lógico que las medidas de un triángulo (o figura cualquiera) sean 0 o negativas. En caso de que alguno (o ambos) valores sean negativos, se desplegará un mensaje de error indicándolo.

```
#include <stdio.h>

void main() {
  double alto;
  double ancho;

  printf("Ingrese el alto.: ");
  scanf("%lf", &alto);

  printf("Ingrese el ancho: ");
  scanf("%lf", &ancho);

  // validemos que sean positivos ambos
  if ((ancho > 0.0) && (alto > 0.0)) {
    // hacer algo acá… aún no
  } else {
    // uno de los dos (o ambos) valores son negativos

    // probamos primero el alto
    if (alto <= 0.0) {
      printf("el valor ALTO no puede ser negativo");
    }
    // probamos ahora el ancho
    if (ancho <= 0.0) {
      printf("el valor ANCHO no puede ser negativo");
    }
  }
}
```

Ahora ¿sabemos cómo validar datos? Correcto! Utilizando la instrucción condicional **if**, conjuntamente con una operación lógica y operadores relacionales. Una vez que ya tengamos esto claro, podremos continuar a la siguiente fase, que será calcular y mostrar al usuario el área de dicha figura geométrica, que en este caso es representada por el triángulo.

4.3. Uso de operaciones aritméticas

A este punto, el programa en sí no hace nada. Tan solo lee un par de valores, los almacena en memoria a través de dos variables y realiza unas validaciones, pero no realiza ningún proceso de los datos introducidos por el usuario.

En este paso, vamos a agregarle una funcionalidad, que será la de calcular el área de la figura en cuestión. Posteriormente a esto se deberá mostrar al usuario el cálculo realizado.

Como ya sabemos, el área de un triángulo viene dada por:

$$A = \frac{base \times altura}{2}$$

Ahora, para incorporar esto a nuestro programa, debemos agregar una nueva variable, que será la encargada de almacenar este cálculo en la memoria.

```
#include <stdio.h>

void main() {
  double alto;
  double ancho;
  double area;  // nuestra variable para el área

  printf("Ingrese el alto.: ");
  scanf("%lf", &alto);

  printf("Ingrese el ancho: ");
  scanf("%lf", &ancho);

  if ((ancho > 0.0) && (alto > 0.0)) {
    // todo OK… hacemos el cálculo del área
    area = (alto * ancho) / 2.0;

    // mostrar resultado por cónsola
    printf("Área del triángulo: %lf", area);
  } else {
    if (alto <= 0.0) {
      printf("el valor ALTO no puede ser negativo");
    }
    if (ancho <= 0.0) {
      printf("el valor ANCHO no puede ser negativo");
    }
  }
}
```

4.4. Uso de funciones creadas por el usuario

Llegado a este punto, nuestro programa ya hace algo… calcula el área de un triángulo. Llegó el momento de separar las funcionalidades de nuestro programa, porque como podemos ver, nuestro programa tiene un comportamiento "lineal", sin ningún subproceso (o métodos y/o función) propios de nuestro programa.

Para hacer la "separación", podemos empezar por la entrada de datos. Si observamos bien, en la entrada de ambas variables se repiten los pasos: mostrar un literal por la consola para luego hacer una lectura de una variable de tipo *double*. Hagamos entonces una función en la cual se reciba un argumento de tipo cadena (que vendría siendo el literal a mostrarle al usuario) y que retorne el valor que el usuario ingrese para el mismo.

Acordémonos de que las cadenas son un arreglo de caracteres y no un tipo primitivo en C++.

Esta nueva función, como ya se dijo, deberá devolver un valor de tipo *double*, para que de esta manera poderle asignar, desde donde se la llame, el valor retornado, siendo este el ingresado por el usuario.

Como funcionalidad adicional, podremos hacer que esta nueva función realice la validación del valor ingresado por el usuario, indicándole cuando exista un error en el valor ingresado (en caso de que este sea un número negativo).

```
#include <stdio.h>

// declaramos la cabezera de la función
double Lee(char var[]);

void main() {
  double alto;
  double ancho;
  double area;

  // hacemos uso de nuestro nuevo método
  alto = Lee("alto.");
  ancho = Lee("ancho");

  if ((ancho > 0.0) && (alto > 0.0)) {
    area = (alto * ancho) / 2.0;

    printf("Área del triángulo: %lf", area);
  }
  // la cláusula else ya no hace falta, ya que el
  // mensaje de error lo mostramos en el nuevo método
}

// creamos nuestra nueva función acá… esta recibirá una
// cadena como entrada, la cual será mostrada por
// consola al usuario, leerá también por la consola un
// valor numérico ingresado por el usuario y este se
// retornará al punto en donde fue invocada esta función
double Lee(char var[]) {
  double valor;

  printf("Ingrese el %s: ", var);
  scanf("%lf", &valor);

  if (valor <= 0.0) {
    printf("El valor %s no puede ser negativo", var);
  }
  return valor;
}
```

Bien!!! Ya tenemos un programa que lee un par de variables y realiza el cálculo pertinente al caso. Pero ¿qué pasaría si quisiéramos continuar haciendo otra serie de cálculos? Por ejemplo, calcularle el perímetro del mismo y otras tantas cosas más. Tendríamos que declarar tantas variables dentro de nuestro método *main*, a tal punto que se perdería la perspectiva del mismo.

4.5. Creación y uso de objetos

Lo mejor que podremos hacer en casos como lo explicado al final del punto anterior es ir separando funcionalidades, ya no por funciones, sino por unidades funcionales (y métodos en consecuencia) inherentes o comunes a cada caso. Por ejemplo, podríamos comenzar por separar lo que es el manejo abstracto de un triángulo de la función principal. De este modo tendremos en nuestro caso una clase nueva: la que representaría a nuestro triángulo.

Hagamos pues entonces una separación de funcionalidades… "Divide y vencerás" reza una máxima antigua… y dicha máxima no se podría aplicar mejor al caso de la Programación Orientada a Objetos, que mientras más separadas (en un orden lógico funcional, por supuesto) estén las cosas, más fácil será su comprensión y mantenimiento.

```
#include <stdio.h>

double Lee(char var[]);

// nuestra nueva clase que representará a la figura de
// un triángulo
class CTriangulo {
  // recordemos que si queremos acceder a los atributos
  // de una clase, estos deberán ser declarados como
  // públicos
  public:
    double alto;
    double ancho;
    // el atributo de 'area' lo podremos ignorar debido
    // a que haremos en su lugar un método que nos
    // devuelva dicho valor

    // método público en el cual se calculará el área
    // del triángulo
    double Area();
};
```

→

```
// desarrollo del método 'Area' de la clase 'CTriangulo'
double CTriangulo::Area() {
  return ((alto * ancho) / 2.0);
}

void main() {
  // declaramos una variable de tipo CTriangulo
  CTriangulo tri;

  // ahora, leemos sus valores de alto y ancho, a través
  // de sus atributos homónimos
  tri.alto = Lee("alto.");
  tri.ancho = Lee("ancho");

  if ((tri.ancho > 0.0) && (tri.alto > 0.0)) {
    // accesamos al método que nos retorna el área
    printf("Área del triángulo: %lf", tri.Area());
  }
}

double Lee(char var[]) {
  double valor;

  printf("Ingrese el %s: ", var);
  scanf("%lf", &valor);

  if (valor <= 0.0) {
    printf("El valor %s no puede ser negativo", var);
  }
  return valor;
}
```

Ahora, hemos separado la funcionalidad de la figura de un triángulo en una clase aparte, la hemos instanciado y finalmente utilizado sus atributos y métodos. Pero para poder cumplir con una de las máximas de la POO (la encapsulación), no deberíamos dejar que los atributos se hagan visibles a todo nivel. Para esto, debemos ahora recodificar este programa para convertir dichos atributos en métodos de acceso. Si hacemos esto, nos tropezaríamos con un inconveniente: ¿cómo hacemos entonces para colocarles valores a estas propiedades?. La solución, en el próximo apartado.

4.6. Creación de propiedades y uso de constructores

Bueno, modifiquemos nuestro programa anterior para que ahora, en vez de utilizar los atributos públicos, accedamos a las características del objeto a través de métodos, y mantener así el encapsulamiento del mismo.

```
#include <stdio.h>

double Lee(char var[]);

class CTriangulo {
  // ahora, los atributos son privados
  private:
    double alto;
    double ancho;
  public:
    // constructor por defecto
    CTriangulo() {
      this->alto = 0.0;
      this->ancho = 0.0;
    }

    // escribamos código de inicialización de los
    // atributos
    CTriangulo(double alto, double ancho) {
      this->alto = alto;
      this->ancho = ancho;
    }

    // escribamos código para nuestra propiedad Alto
    double getAlto() { return alto; }

    // escribamos código para nuestra propiedad Ancho
    double getAncho() { return ancho; }

    double Area();
};

double CTriangulo::Area() {
  return ((alto * ancho) / 2.0);
}
```

```
void main() {
  // declaremos un par de variables para su uso interno
  double ancho; // variable para el ancho
  double alto;  // variable para el alto

  // ahora será un puntero a memoria SIN instanciar
  CTriangulo* tri;

  // utilizaremos un par de variables 'normales' para la
  // lectura de las medidas
  alto = Lee("alto.");
  ancho = Lee("ancho");

  if ((ancho > 0.0) && (alto > 0.0)) {
    // ahora SI podremos instanciar el objeto con los
    // valores conocidos
    tri = new CTriangulo(alto, ancho);

    printf("Área del triángulo ");
    printf("de %lfx%lf es: ",
           tri->getAncho(),
           tri->getAlto());
    printf("%lf", tri->Area());

    // eliminamos el objeto de la memoria
    delete tri;
  }
}

double Lee(char var[]) {
  double valor;

  printf("Ingrese el %s: ", var);
  scanf("%lf", &valor);

  if (valor <= 0.0) {
    printf("El valor %s no puede ser negativo", var);
  }
  return valor;
}
```

Ahora vemos como, un simple problema de cálculo de área de un triángulo, se ha convertido en algo "largo y difícil"... pero sabemos que esta es la mejor manera de aplicar la POO, ya que todas las operaciones inherentes a algún objeto que envuelva a la figura geométrica de un triángulo, se hará solamente en una clase, encapsulando sus características, y la interacción con el "mundo exterior" se hará a través de sus métodos.

Bien, ¿qué pasaría si ahora el usuario desea que, en vez de leer y procesar un triángulo, quisiera procesar N triángulos? Según como están las cosas, no le quedaría más remedio, al usuario, que ejecutar nuestra aplicación N veces, lo cual no es muy convincente ni estratégico, y más aún dados los adelantos tecnológicos existentes hoy día (¿para qué tener entonces un computador, si la tarea repetitiva se debe hacer manual?).

Deberemos entonces modificar nuestro programa para cumplir con este nuevo requerimiento del usuario.

4.7. Uso de ciclos iterativos: *for*

Bien, nuestro próximo paso será ahora el de tratar de hacer que el programa realice el proceso N veces. Para esto, debemos saber si la cantidad N es conocida. De ser así, procederemos a leer N y posteriormente, en un ciclo iterativo *for* haremos el proceso.

```
#include <stdio.h>

double Lee(char var[]);

class CTriangulo {
  private:
    double alto;
    double ancho;
  public:
    CTriangulo() {
      this->alto = 0.0;
      this->ancho = 0.0;
    }

    CTriangulo(double alto, double ancho) {
      this->alto = alto;
      this->ancho = ancho;
```

```
    }

    double getAlto() { return alto; }

    double getAncho() { return ancho; }

    double Area();
};

double CTriangulo::Area() {
  return ((alto * ancho) / 2.0);
}

void main() {
  // usaremos una variable para guardar cuantos
  // triángulos desea procesar el usuario: N
  int N;
  double ancho;
  double alto;
  CTriangulo* tri;

  // leemos la cantidad de triángulos que se quiere
  // procesar
  printf("Ingrese N: ");
  scanf("%i", &N);

  // recurrimos al ciclo for
  for (int i = 0; i < N; i++) {
    // usamos el '\n' para poder 'bajar' una línea el
    // cursor sobre el monitor
    printf("Ingrese datos de la figura %i\n", i + 1);

    alto = Lee("alto.");
    ancho = Lee("ancho");

    if ((ancho > 0.0) && (alto > 0.0)) {
      tri = new CTriangulo(alto, ancho);

      printf("Área del triángulo ");
      printf("de %lfx%lf es: ",
             tri->getAncho(),
             tri->getAlto());
      printf("%lf\n", tri->Area());

      delete tri;
    }
  }
}

double Lee(char var[]) {
  double valor;

  printf("Ingrese el %s: ", var);
  scanf("%lf", &valor);

  if (valor <= 0.0) {
    printf("El valor %s no puede ser negativo", var);
  }
  return valor;
}
```

Resuelto… pero claro, tiene sus fallas. Por ejemplo, ¿que pasaría si el usuario se equivoca y coloca uno de los datos de algún triángulo negativo? Simplemente que se perdería ese intento de lectura y, si el usuario deseaba procesar 5 triángulos, sólo le quedarán para procesar 4. Pero eso es algo que ya usted debería saber como solucionar.

El siguiente paso es ¿cómo haríamos para, en vez de calcular el área de triángulos, también lo podamos hacer para rectángulos? La solución, en el próximo apartado. Los invito a que continúen…

4.8. Apoyo de otra clase y uso del ciclo *do-while*

Acá estaremos ocupados ahora en la implementación de otra nueva clase: la clase para el manejo de la figura geométrica rectangular. Se nos presenta, a primeras de cambio, independientemente de cómo se haga la clase para esta nueva figura, el cómo reconocer que los datos que esté ingresando el usuario sean los de un triángulo o los de un rectángulo.

Para resolver esto, vamos a modificar nuestro programa principal (función *main*) para actúe también como un menú, en el que el usuario elegirá cuál es la figura que desea procesar en ese momento en específico (dentro de la lista de N figuras).

Entonces, lo que se hará serán dos cosas: crear una nueva clase denominada, en este ejemplo, como *TRectangulo* y lo otro que nos queda hacer es el menú del usuario. Para la confección de este menú utilizaremos el ciclo iterativo *do-while*. Veamos como sería:

```
#include <stdio.h>

double Lee(char var[]);

class CTriangulo {
  private:
    double alto;
    double ancho;
  public:
    CTriangulo() {
      this->alto = 0.0;
      this->ancho = 0.0;
    }

    CTriangulo(double alto, double ancho) {
      this->alto = alto;
      this->ancho = ancho;
    }

    double getAlto() { return alto; }

    double getAncho() { return ancho; }

    double Area();
};

double CTriangulo::Area() {
  return ((alto * ancho) / 2.0);
}

// he aquí nuestra nueva clase para el manejo de
// rectángulos... todas las propiedades y atributos
// permanecerán iguales, lo que debe cambiar en el
// método del cálculo del área
class CRectangulo {
```

→
←

```
  private:
    double alto;
    double ancho;
  public:
    CRectangulo() {
      this->alto = 0.0;
      this->ancho = 0.0;
    }

    CRectangulo(double alto, double ancho) {
      this->alto = alto;
      this->ancho = ancho;
    }

    double getAlto() { return alto; }

    double getAncho() { return ancho; }

    double Area();
};

// este método deberá cambiar, ya que el área se calcula
// de manera diferente
double CRectangulo::Area() {
  return (alto * ancho);
}

void main() {
  int N;
  double ancho;
  double alto;

  // nueva variable, que será usada para la opción del
  // menú
  char op[10];

  CTriangulo* tri;
  CRectangulo* rec; // nuestra nueva variable para
                    // rectángulos

  printf("Ingrese N: ");
```

→
←

```
  scanf("%i", &N);

  for (int i = 0; i < N; i++) {
    // ciclo iterativo en el cual se le pedirá al
    // usuario cual es la siguiente figura en procesar
    do {
      printf("1.- Triángulo\n");
      printf("2.- Rectángulo\n");
      printf("Escriba su opción y pulse ENTER");

      scanf("%s", op);
    } while ((op[0] != '1') && (op[0] != '2'));

    printf("Ingrese datos de la figura %i\n", i + 1);

    alto = Lee("alto.");
    ancho = Lee("ancho");

    if ((ancho > 0.0) && (alto > 0.0)) {
      printf("Área de la figura ");

      if (op[0] == '1') {   // la figura es un triángulo
        tri = new CTriangulo(alto, ancho);

        printf("de %lfx%lf es: ",
               tri->getAncho(),
               tri->getAlto());
        printf("%lf\n", tri->Area());

        delete tri;
      } else {              // sino, será un rectángulo
        rec = new CRectangulo(alto, ancho);

        printf("de %lfx%lf es: ",
               rec->getAncho(),
               rec->getAlto());
        printf("%lf\n", rec->Area());

        delete rec;
      }
    }
```

→
←

```
  }
}

double Lee(char var[]) {
  double valor;

  printf("Ingrese el %s: ", var);
  scanf("%lf", &valor);

  if (valor <= 0.0) {
    printf("El valor %s no puede ser negativo", var);
  }
  return valor;
}
```

Como se ve, es sumamente sencillo. Lo único que hicimos fue, prácticamente, copiar y pegar la clase *CTriangulo* y nombrarla *CRectangulo*. El otro cambio fue la introducción de un bucle *do-while* para que fungiera como una especie de menú del sistema, en el cual el usuario podrá escoger entre ingresar los datos de un triángulo o los de un rectángulo.

Pero, como se ve (y se dijo), la clase *TRectangulo* no es más que una vulgar copia de la clase *CTriangulo*. Y si vemos bien, entre estas dos clases existe gran similitud. ¿Será que se podrá hacer una clase base, en la que se englobe lo común a estas dos figuras y, que de ella se hereden ambas clases?... Pues si...

4.9. Herencia y uso del objeto especial *base*

Tomemos ahora el código resultante del ítem anterior y modifiquémoslo para implementar la herencia. Debido, y es lógico pensar así, a que ambas clases representan a una figura geométrica en particular, no dejan de ser precisamente eso: una figura geométrica. Si analizamos bien la situación, tenemos que las figuras geométricas que estamos utilizando de ejemplo, tienen mucho en común: ambas tienen medidas de altura y anchura, así como también a ambas se les puede calcular el área ocupada.

Entonces, podremos crear una clase base (*TFigura*) en la que estarán los atributos comunes a las dos figuras en particular. Posteriormente, hagamos del método de cálculo del área un método que podamos sobrecargar, para que de esta manera poder utilizar a plenitud otro de los conceptos de la

POO: el *polimorfismo*.

Otro de los cambios que le podríamos introducir a este programa, para terminar con este tema, es eliminar el ingreso de la cantidad de figuras que el usuario desea procesar y a cambio agregamos una nueva opción a nuestro menú para que el mismo usuario escoja en que momento desea terminar la ejecución del programa.

```
#include <stdio.h>

double Lee(char var[]);

// la nueva clase: será la base (padre) de las otras dos
// clases
class CFigura {
  private:
    double alto;
    double ancho;

  public:
    // cuando se vaya a heredar una clase de otra, la
    // clase padre DEBERÁ tener definido el constructor
    // por defecto
    CFigura() {
      alto = 0.0;
      ancho = 0.0;
    }

    CFigura(double alto, double ancho) {
      this->alto = alto;
      this->ancho = ancho;
    }

    double getAlto() { return alto; }

    double getAncho() { return ancho; }

    // el método de la clase base devolverá SIEMPRE 0,
    // ya que lo ideal es que este método no sea llamado
    // desde ninguna parte... por esto es declarada como
    // VIRTUAL, para que desde las clases hijas se pueda
                                                    →
                                                    ←
```

```
    // sobreescribir
    virtual double Area() {
      return 0.0;
    }
};

// heredamos ahora de la clase CFigura, por lo que no
// hacen falta las declaraciones de los atributos y
// propiedades
class CTriangulo : CFigura {
  public:
    // hacemos uso del llamado al constructor de la
    // clase padre
    CTriangulo(double alto, double ancho):
               CFigura(alto, ancho) { }

    // este método será sobreescrito
    double Area();
};

double CTriangulo::Area() {
  return ((getAlto() * getAncho()) / 2.0);
}

// heredamos ahora de la clase CFigura, por lo que no
// hacen falta las declaraciones de los atributos y
// propiedades... acá se aplica lo mismo que para la
// clase CTriangulo
class CRectangulo : CFigura {
  public:
    CRectangulo(double alto, double ancho):
                CFigura(alto, ancho) { }

    double Area();
};

double CRectangulo::Area() {
  return (getAlto() * getAncho());
}

void main() {
                                                    →
                                                    ←
```

```
// ya no necesitaremos más a la variable 'N'
double ancho;
double alto;

// cambiemos de char[] a integer
int op;

// tampoco requeriremos del uso de variables del tipo
// CTriangulo o CRectangulo... sólo usaremos CFigura y
// los demás salen por polimorfismo
CFigura* fig;

// bucle infinito hasta tanto el usuario no escoja la
// opción de salida del sistema
do {
  printf("1.- Triángulo\n");
  printf("2.- Rectángulo\n");
  printf("9.- Para salir del programa\n");
  printf("Escriba su opción y pulse ENTER ");

  scanf("%i", &op);

  if ((op == 1) || (op == 2)) {
    printf("Ingrese datos de la figura\n");

    alto = Lee("alto.");
    ancho = Lee("ancho");

    if ((ancho > 0.0) && (alto > 0.0)) {
      // fijarse bien en el uso ahora del objeto
      // 'fig'... este objeto es 'polimórfico'
      if (op == 1) {
        fig =(CFigura*)(new CTriangulo(alto, ancho));
      } else {
        fig =(CFigura*)(new CRectangulo(alto, ancho));
      }

      printf("Área de la figura ");
      printf("de %lfx%lf es: ",
             fig->getAncho(),
             fig->getAlto());
```

→
←

```
        printf("%lf\n", fig->Area());

        delete fig;
      }
    }
  } while (op != 9);
}

double Lee(char var[]) {
  double valor;

  printf("Ingrese el %s: ", var);
  scanf("%lf", &valor);

  if (valor <= 0.0) {
    printf("El valor %s no puede ser negativo", var);
  }
  return valor;
}
```

Con este ejemplo final hemos puesto de manifiesto los grandes beneficios que nos ofrece la Programación Orientada a Objetos, sobre todo en la funcionalidad que otorga la herencia de clases así como del concepto de polimorfismo.

Y bien, ya con esto hemos terminado este largo y gran capítulo. A lo largo de este, se han mostrado todas las facetas de la POO, así como también las instrucciones básicas del leguaje de programación denominado C++. Espero que les sirva de ayuda…