MANUAL RÁPIDO DE USO RECOMENDADO DE BUILDER C++

INTRODUCCIÓN

Aunque a lo largo del curso 2010/2011 se puede usar cualquier compilador estándar de C++ para hacer las prácticas, hay dos que se recomiendan especialmente: el Dev C++ (por ser el que se utilizó mayormente en primero) y el Builder C++ (la versión del laboratorio de prácticas es la 6.0 Enterprise Suite).

La razón de recomendar el Builder C++ 6.0 Entreprise es porque incorpora una herramienta de validación del uso de memoria de cualquier programa que realicemos (el **CodeGuard**). Hay otras herramientas comerciales e incluso gratuitas que permiten esto mismo, pero vamos a usar esta por su sencillez.

Controlar el uso racional de la memoria de los programas se valorará en las prácticas de este año.

INSTALACION

1) Usar la instalación por defecto, aunque no hace falta instalar bases de datos ni otras historias.

2) No hace falta registrarse. Lo único, poner el número de serie. Puedes encontrar el Builder C++ 6.0 y el número de serie (además de otro mucho software) en la unidad *X* compartida en el laboratorio de prácticas.

EJECUCION

La forma recomendada de trabajar con el Builder C++ 6.0 es:

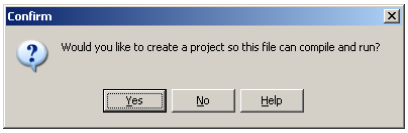
1) Crear una carpeta por cada programa que vayamos a realizar. Por ejemplo, creamos la carpeta “practica0”.

2) Dentro, creamos un fichero de texto vacío llamado *main.cpp*. Si lo pinchamos dos veces y el Builder C++ (en adelante BC++) está correctamente asociado a los archivos .cpp, se intentará abrir con este programa. De no ser así, hay que darle con el botón derecho del ratón y en “Abrir con” tenemos que buscar el ejecutable del BC++ que se llama `bcb.exe` y que en la instalación por defecto está en:

   ```
   C:\archivos de programa\borland\cbuilder6\bin
   ```

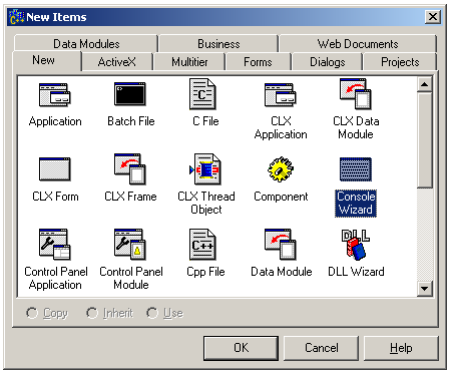
   Podemos establecer la asociación con los archivos .cpp si vamos a usar siempre este programa para trabajar con C++.

3) Si abrimos el *main.cpp* vacío, el Builder puede preguntarnos:

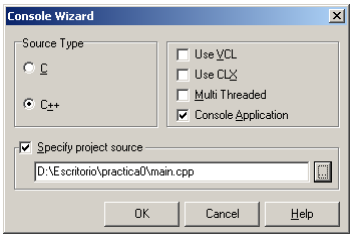


¿Queremos crear un proyecto que podamos compilar y ejecutar? Le decimos que sí (y vamos a paso 5). Si no lo pregunta, da igual, creamos el proyecto nosotros mismos (seguir en paso 4).

4) Para crear el proyecto: menú *File -> New -> Other* y elegimos:

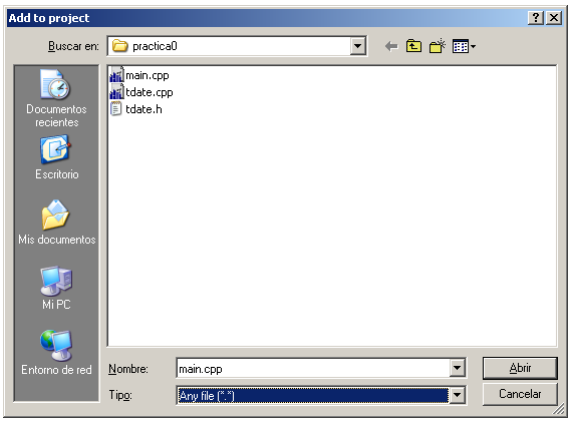


Console wizard. Todas nuestras aplicaciones van a ser en consola, no van a necesitar librerías gráficas ni van a ser multi-hilo. Así que en la siguiente ventana que nos presenta el Builder debería tener un aspecto como éste:



Acordáos de elegir la fuente del proyecto, el main.cpp vacío que hemos creado al principio.

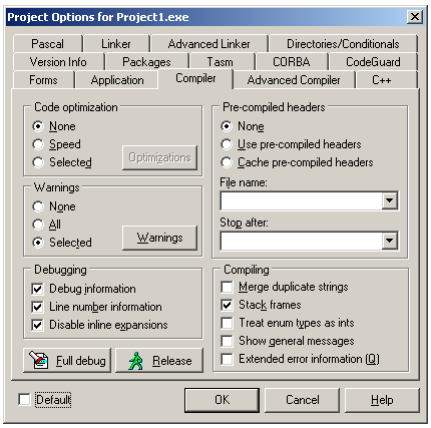
5) Si tenemos todos los archivos correspondientes de la práctica en el mismo directorio (por ejemplo como con la clase TDate de la práctica 0), podemos añadirlos al proyecto sin más que darle al menú *Project -> Add to Project* y elegir "ver todos los tipos de archivo":

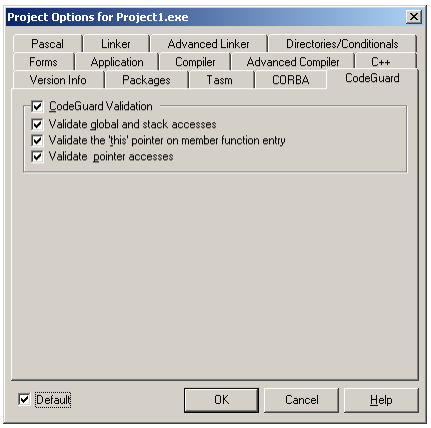


Ahora, usando la tecla CTRL seleccionamos los archivos que faltan de la práctica (*tdate.h* y *tdate.cpp*) y los añadimos al proyecto.

6) Si rellenamos el *main.cpp* (vacío, se supone) con el código adecuado, ya podemos compilar (CTRL+F9) y ejecutar (F9). Es MUY recomendable grabar antes el proyecto, usando para eso el botón  .

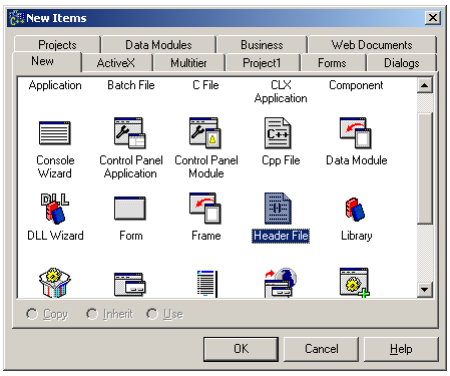
7) En las opciones de cualquier proyecto debemos asegurarnos de que NO haya cabeceras precompiladas (figura de la izquierda, *pre-compiled headers*) y que la herramienta *CodeGuard* esté siempre activada (figura de la derecha, seleccionarlo todo y dejarlo como *default*).





De todas maneras, incluso, de un ordenador a otro, se recomienda llevarse sólo el código fuente y crear el proyecto de nuevo. Por si acaso.

8) Si lo que estuviéramos haciendo fuera crear código nuevo, sólo hay dos posibilidades: crear un fichero de interfaz .h o un fichero de implementación .cpp. Así pues, le damos a menú *File -> New -> Other* y elegimos *Cpp file* o *header file* según corresponda.



Se creará un nuevo fichero File1.h o File1.cpp que no cogerá el nombre que nos interese hasta que pulsemos salvar  . Guardar en la carpeta del proyecto e IMPORTANTE, añadirlo al proyecto tal y como se ha descrito en el apartado 5).

DEPURACIÓN

Una opción típica en cualquier IDE es la de depuración (*debug*). Las acciones más interesantes son las siguientes:

1) **Break** (marcar donde nos interesa que el programa se detenga), para ir hasta ahí de golpe (al darle a F9) y desde ahí paso a paso. Para eso señalamos con el ratón en el margen izquierdo del programa, y hacemos click izquierdo para que salga el punto rojo.

main.cpp

```
#include <iostream>
#include <stdio>
#include "TDate.h"

using namespace std;


int main(int argc, char* argv[]) {
        TDate fecha1(31,12,2000);
        TDate fecha2(3,3,2001);
        TDate fecha3("24/01/2001");
        fecha1.print();
        cout << "\n";
        fecha2.print();
        cout << "\n";
        fecha3.print();
        cout << "\n";

        cout << "fecha2 menos fecha1 = ";

        TDate res = fecha2 - fecha1;

        res.print();
        cout << endl << "La misma fecha es " << res;

        fecha1++;
        ++fecha2;

        cout << "\n";
        fecha1.print();
        cout << "\n";
        fecha2.print();

        getchar();

        return 0;
}
```

2) ***Step over*** (“paso por encima”), para que se ejecute completamente la línea siguiente del programa. Para ejecutar este comando pulsamos en  o en F8. En el ejemplo anterior, ejecutará `fecha1.print()` completamente (dará un resultado) y el programa se detendrá en la línea siguiente.

3) ***Trace into*** (“recorrer por dentro”), para que el programa pase el control a la primera línea de una función que queramos recorrer paso a paso. Por ejemplo si tras varios *step over* nos detenemos en la línea `fecha3.print()` y entonces hamos *trace into*, el control del programa pasará al método *print* de la clase *TDate*.

main.cpp

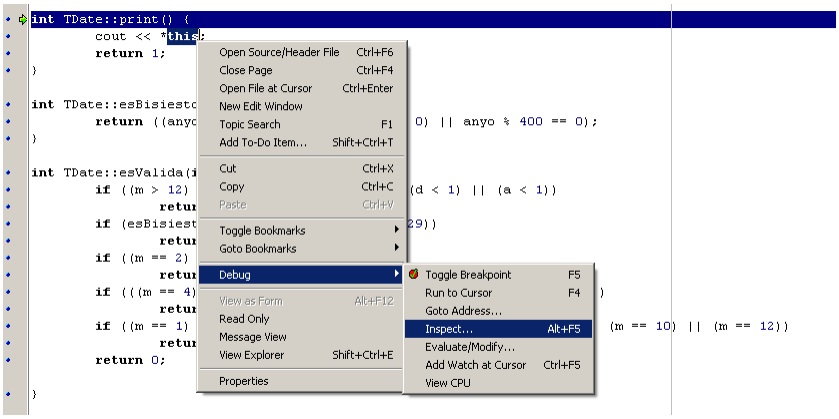
```
#include <iostream>
#include <stdio>
#include "TDate.h"

using namespace std;


int main(int argc, char* argv[]) {
        TDate fecha1(31,12,2000);
        TDate fecha2(3,3,2001);
        TDate fecha3("24/01/2001");
        fecha1.print();
        cout << "\n";
        fecha2.print();
        cout << "\n";
        fecha3.print();
        cout << "\n";

        cout << "fecha2 menos fecha1 = ";

        TDate res = fecha2 - fecha1;

        res.print();
        cout << endl << "La misma fecha es " << res;

        fecha1++;
        ++fecha2;

        cout << "\n";
        fecha1.print();
        cout << "\n";
        fecha2.print();

        getchar();

        return 0;
}
```

main.cpp | tdate.cpp

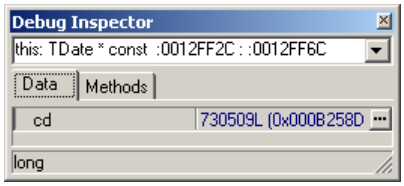
```
}

int TDate::setDate(const char *c) {
        // ejercicio
        // posiblidad con lo que tenemos
        TDate aux = TDate(c);
        cd = aux.cd;
        return 1;
}

int TDate::setDate(long d) {
        cd = d;
        return 1;
}

int TDate::print() {
        cout << *this;
        return 1;
}

int TDate::esBisiesto(int anyo) {
        return ((anyo % 4 == 0 && anyo % 100 != 0) || anyo % 400 == 0);
}
```

Notar que cuando estamos depurando en el estado de *break* podemos conocer el valor de cualquier variable asignada actualmente en la memoria, sin más que pasar por encima la

flecha del ratón. Aún más interesante es la opción de seleccionar la variable que nos interesa conocer su estado, sus métodos y hasta la posición de memoria que ocupa. Para ello seleccionamos con el ratón la variable (en el ejemplo siguiente el puntero *this*) y vamos a *Debug -> Inspect*.



Nos saldrá una ventanita como ésta:



El puntero *this* es un puntero al objeto de la propia clase. En este caso estábamos imprimiendo un TDate (fecha). Así pues, nos sale como datos los atributos de la clase TDate (el atributo cd de la cuenta de días) y su valor. En otra pestaña tenemos los métodos disponibles para esta clase.

También tenemos las direcciones de memoria de todo, lo cual es muy útil por ejemplo si tenemos una clase relativamente compleja cuyos atributos son de otra clase diferente. Al pulsar en la dirección de memoria podemos recorrer los valores de los atributos de diferentes tipos, en cadena. Por ejemplo, piénsese en una carrera de corredores de Fórmula 1. Podemos tener una clase Carrera que sea una lista de elementos de la clase Corredor. Pues bien, accediendo en modo *debug* a la clase Carrera podríamos acceder también a los corredores de la misma, que son de otra clase, sin más que pinchar en la dirección de memoria donde se almacenan estos datos (pestaña Data).

CODEGUARD

Esta herramienta es el chivato que nos avisa si una práctica se ha hecho con malas prácticas de programación o con poco cuidado. Si el programa compila no significa que todo esté del todo bien. Dependiendo de las opciones de la compilación, aún siendo más o menos estándar, es posible que errores tan peligrosos como acceder a una posición del array que no existe pasen inadvertidos.

Es cuestión de leer el mensaje de error en inglés. Los más típicos son:

1) Acceder a una posición no válida de memoria (*array out of bounds*). Ejemplo típico: un array del que nos hemos salido de sus límites.
2) Acceder a una posición de memoria donde había algo pero ya no (*access on free memory*). Ejemplo típico: reservar memoria para un objeto o variable (con *new*), liberarlo (con *delete*) y luego pretender volver a acceder a él.

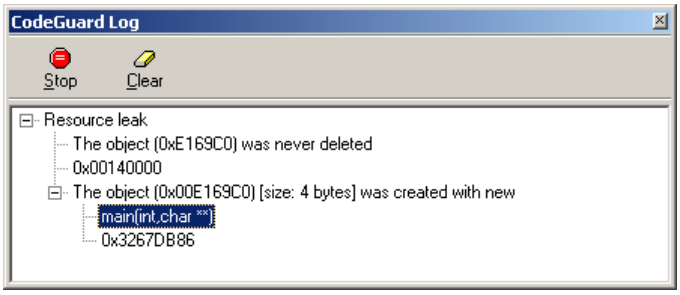
3) Fuga de memoria (resource *leak*). Ejemplo típico: no liberar memoria reservada expresamente, no escribir el código adecuado en el destructor de alguna clase para liberar su memoria, en ambos casos antes de que termine de ejecutarse el programa.

El CodeGuard sirve de ayuda para detectar estos errores, a veces de difícil comprensión. Por ejemplo, imaginemos el código siguiente, en el que hemos modificado el *main* del programa TDate para declarar `fecha1` como un puntero:

main.cpp | tdate.cpp

```
#include <iostream>
#include <stdio>
#include "TDate.h"

using namespace std;


int main(int argc, char* argv[]) {
        TDate* fecha1 = new TDate(31,12,2000);
        TDate fecha2(3,3,2001);
        TDate fecha3("24/01/2001");
        fecha1->print();
        cout << "\n";
        fecha2.print();
        cout << "\n";
        fecha3.print();
        cout << "\n";

        cout << "fecha2 menos fecha1 = ";

        TDate res = fecha2 - (*fecha1);

        res.print();
        cout << endl << "La misma fecha es " << res;

        (*fecha1)++;
        ++fecha2;

        cout << "\n";
        fecha1->print();
        cout << "\n";
        fecha2.print();

        getchar();

        return 0;
}
```

Notar que hemos hecho reserva expresa de memoria para `fecha1`. Sin embargo, no la hemos liberado antes de terminar el programa. Como resultado, el programa se ejecuta normalmente pero al terminar…



Esta ventanita típica de problemas de memoria es el log de CodeGuard. En la misma nos dice el objeto que está causando problemas (el 0xE169C0) y por qué (nunca fue borrado). Para saber qué es lo que ha pasado, en la parte de abajo nos indica el ciclo de vida del objeto o dato que está dando problemas. Se lee de abajo a arriba, aunque en este caso sólo nos indica que el objeto fue creado con *new* (se reservó memoria) y lo más importante, nos dice dónde ocurrió esto. Fue en el *main*, si pinchamos en el texto del *main* nos va a la línea exacta donde reservamos la memoria que nunca liberamos.

La solución: añadir la línea delete `fecha1` antes de terminar el programa.

MUY IMPORTANTE: cuando el CodeGuard nos detecte un problema lo mejor es detener el programa volviendo a compilar (CTRL+F9). Si le damos a STOP es bastante probable que el programa se bloquee y haya que terminar con el proceso bcb.exe para poder volver a abrir el BC++. Evidentemente Borland no usó sus propias herramientas para construir el Builder C++…