

Programando con Code::Blocks

Una respuesta típica de los alumnos que pasan de Borland C++ 3.1 a code::blocks es:

“lo instalé pero los programas no compilan”

Las razones más comunes comprenden a la estructura de proyectos, que vimos en Introducción a Code::Blocks, y ciertas diferencias en las librerías disponibles en Gcc respecto al compilador de Borland.

```
prueba\main.c:25: warning: implicit declaration of function `getch'
```

```
Prueba/main.c:26: undefined reference to `__clrscr'
```

¿A qué se deben estos mensajes?

El primero, nos dice que tenemos una declaración implícita a una función llamada getch() . Aparece cuando invocamos esa función en nuestros programas. Esta no está incluida en las librerías por defecto en este IDE. Algo parecido sucede con la segunda.

El método system(char *);

Este procedimiento nos ayudará a reemplazar estas dos funciones de una forma simple, sin tener que programar o incluir ninguna librería adicional. Como su nombre lo dice, el procedimiento system (char *); hace llamadas a la línea de comandos del sistema operativo, sea este cual fuere.

Para reemplazar a getch(void); solo hay que reemplazar esa línea por system(“pause”); -en Windows-

Y en el caso de clrscr(void); si bien CodeBlocks abre una ventana nueva para ejecución, en programas más extensos puede interesarnos limpiar la consola. Entonces podemos reemplazar esa función con system(“cls”);

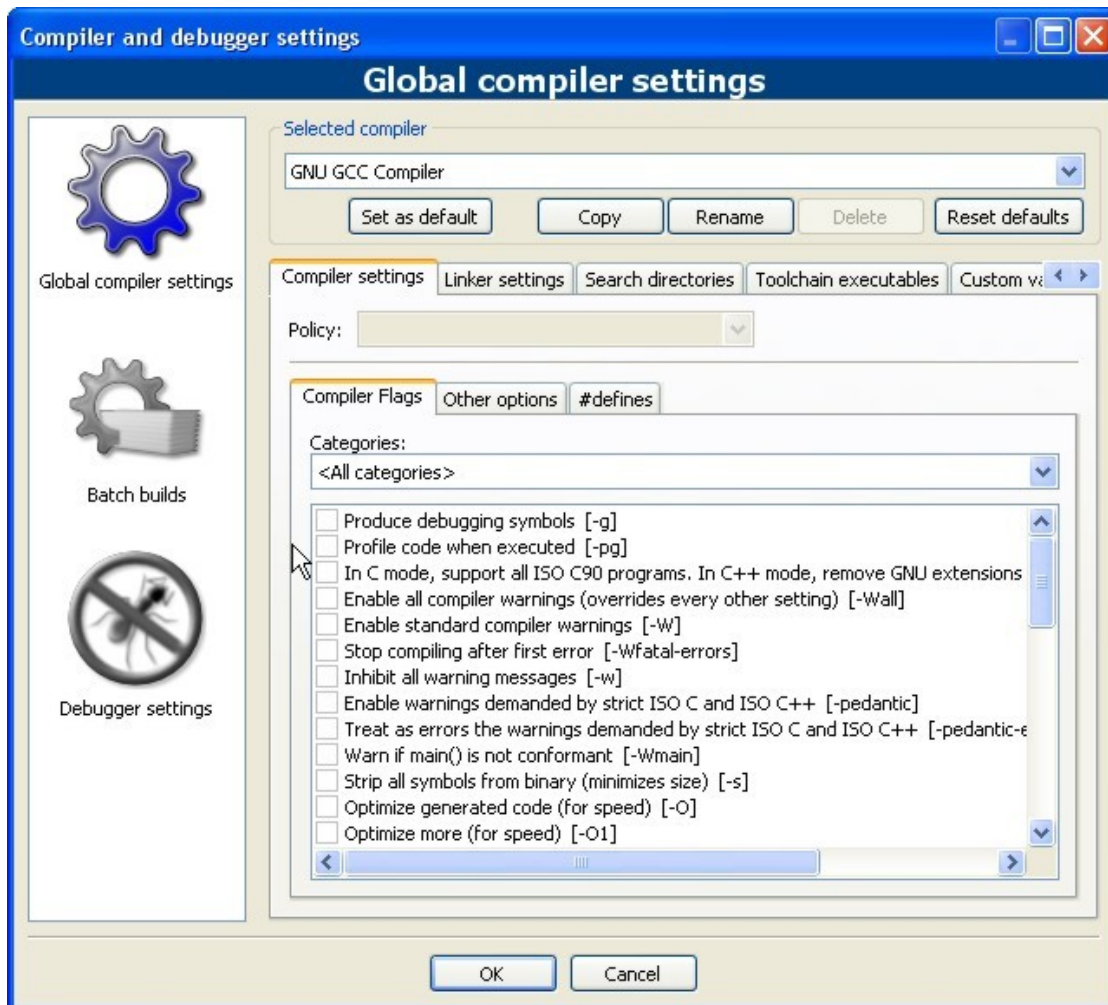
Debug: depurar programas

Para poder depurar nuestros programas, necesitamos hacer que el compilador produzca símbolos de depuración en las versiones de “debug” de nuestras compilaciones. Esto no es algo que venga por defecto pero es muy sencillo de activar.

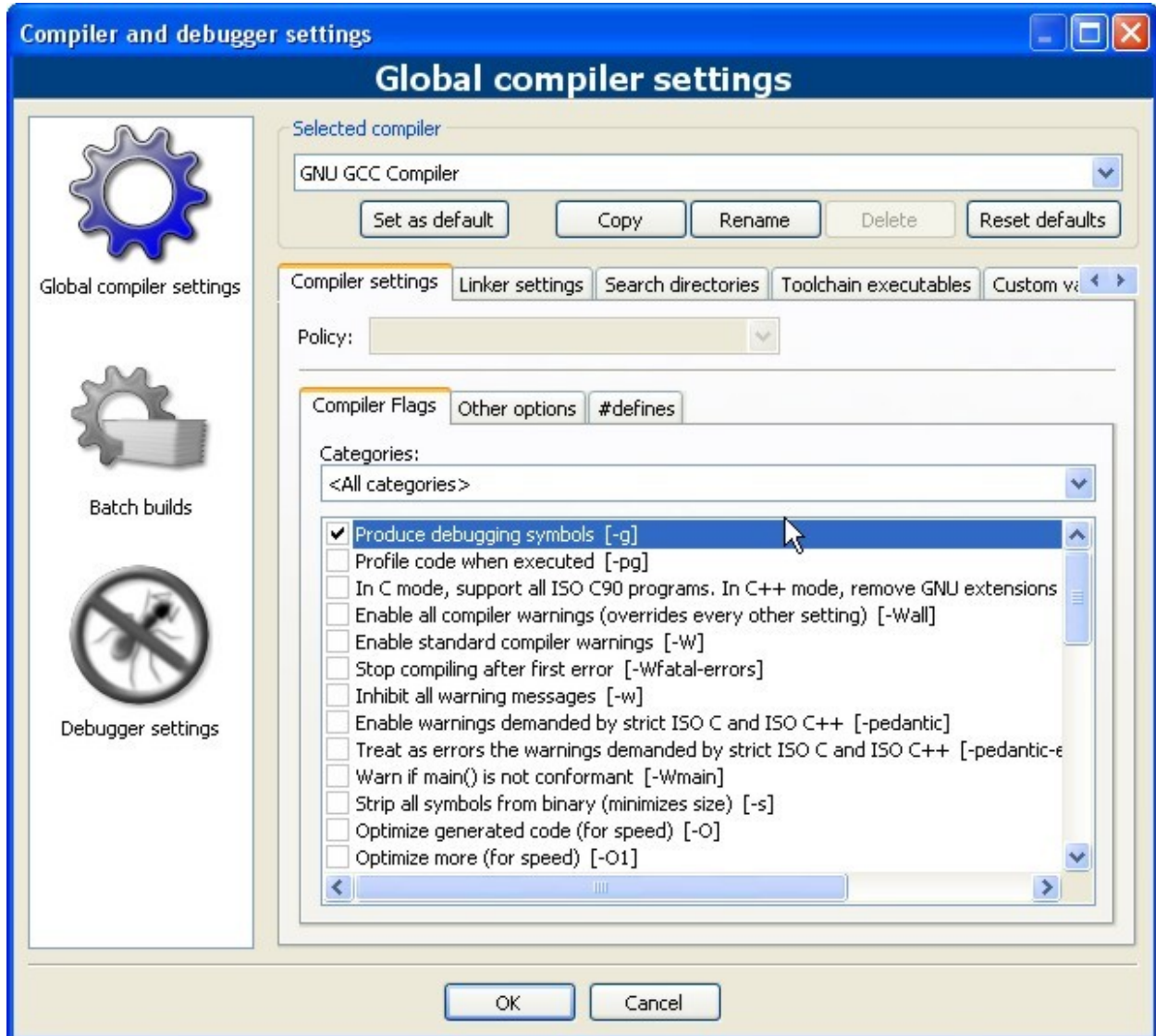
Primero tenemos que ir a Settings-> Compiler and Debugger



Luego tildar en la primera opción



Así debería quedar



Ahora que tenemos esto podemos ver qué contienen las variables de nuestros programas en modo Debug.

Depurando paso a paso

¿De qué sirve depurar un programa y cuando debo hacerlo? La respuesta lisa y llana es: cuando no funcionan como debería y/o tiene comportamientos inesperados, freezeos, etc.

Para depurar tenemos que poner nuestro “build target” en modo “debug”, y una vez que tenemos eso, podemos compilar el programa y ejecutarlo línea por línea (step into), o bien normalmente con ciertos puntos de quiebre de la ejecución (breakpoints) donde el programa para su ejecución en una determinada línea para que podamos inspeccionarlo más detenidamente solo en ese punto o en las líneas siguientes.

Veamos ahora un poco de código, de ejemplo para ver resultados concretos

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Empleado{
    int legajo;
    char nombre[30] ;
    char direccion[50];
    float sueldo;
};

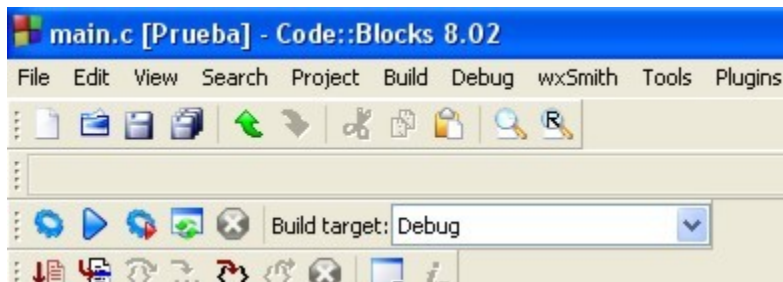
int main(void)
{
    printf("Hello world!\n");

    struct Empleado emp;
    scanf("%s",emp.nombre);
    strcpy (emp.direccion,"Balcarce 50");
    emp.sueldo=9000,1;
    emp.sueldo=9002,1;
    emp.sueldo=900232,1;
    emp.sueldo=9010,1;
    emp.sueldo=0,1;
    system("dir");

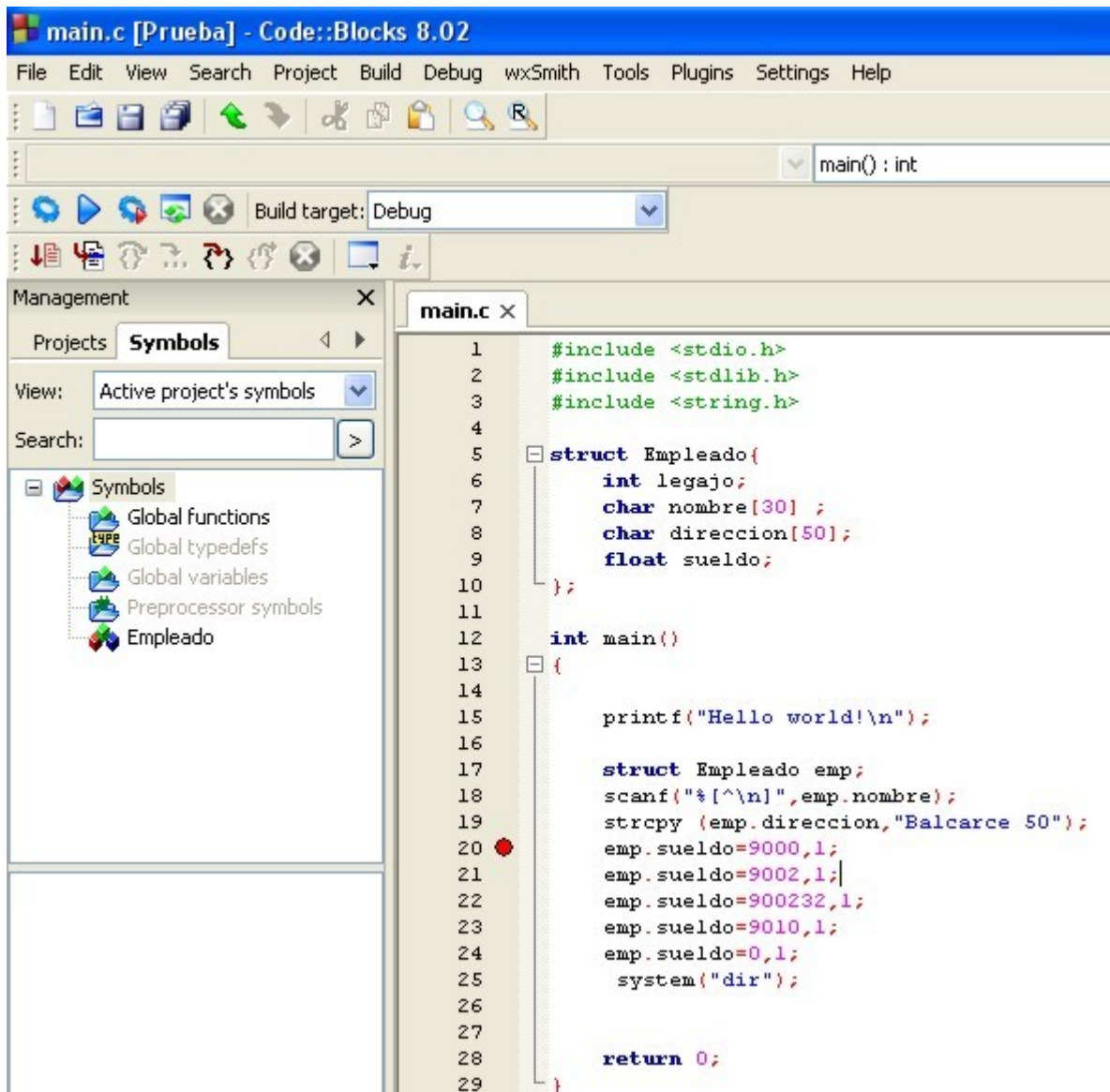
    return 0;
}
```

Este programa no tiene mucha utilidad en general pero si nos sirve en este caso para ver los valores internos en la ejecución (watches)

Paso 1: build target

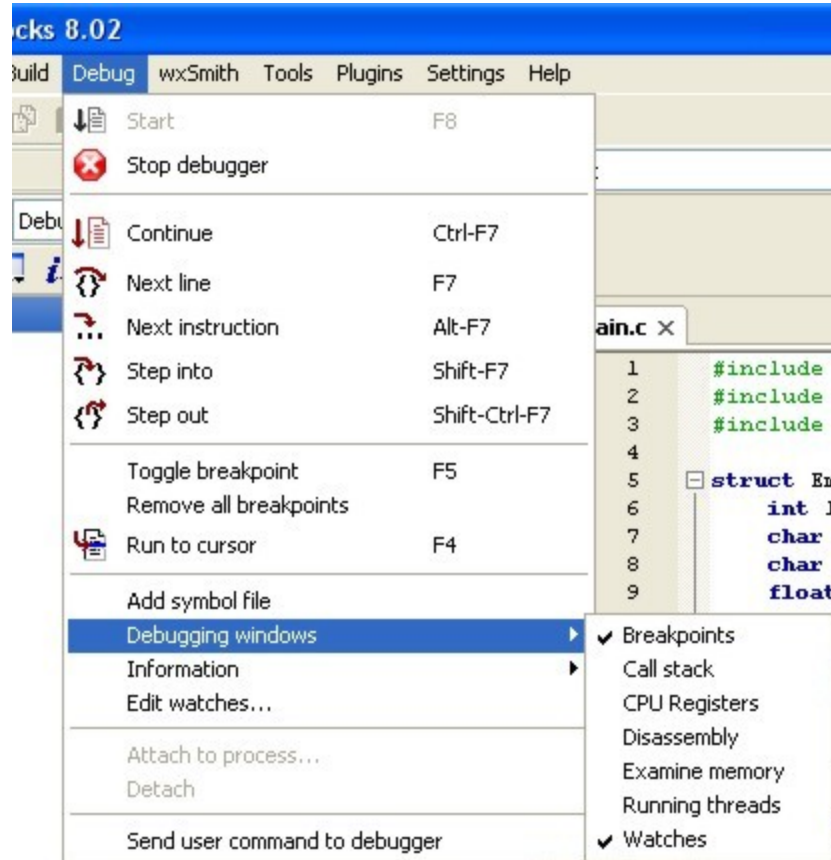


Paso 2: poner un breakpoint, clicar a la derecha del número de línea.



El punto rojo nos dice donde está puesto él o los breakpoints

IMPORTANTE: Activar la ventana de Watches



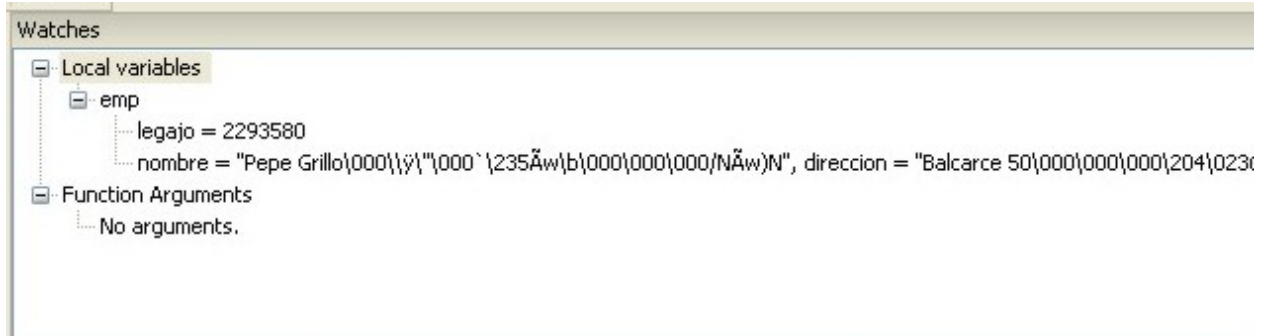



compilar

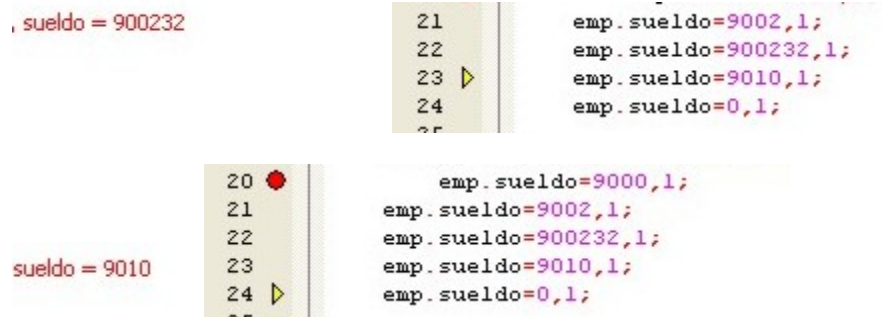


debug/continue

PASO 3: Ingresar un nombre y apretar enter, verán que la ventana watches tiene información.



PASO 4:  apretar step into. Verán que la línea cambia y los valores también, en la variable sueldo.



Se puede observar cualquier tipo de variable, ya sea de tipo simple (int, float, char, short, etc.) o tipo compuesto (string, struct, enum,etc.) y tipos definidos por el usuarios (nodos, listas, etc.).

La idea es que vayan investigando los demás comando de debug, y puedan familiarizarse con esta herramienta tan útil y poderosa.