

Manual 1 - 2do Torneo de Programación Competitiva

Lions R.C.

Junio 2019



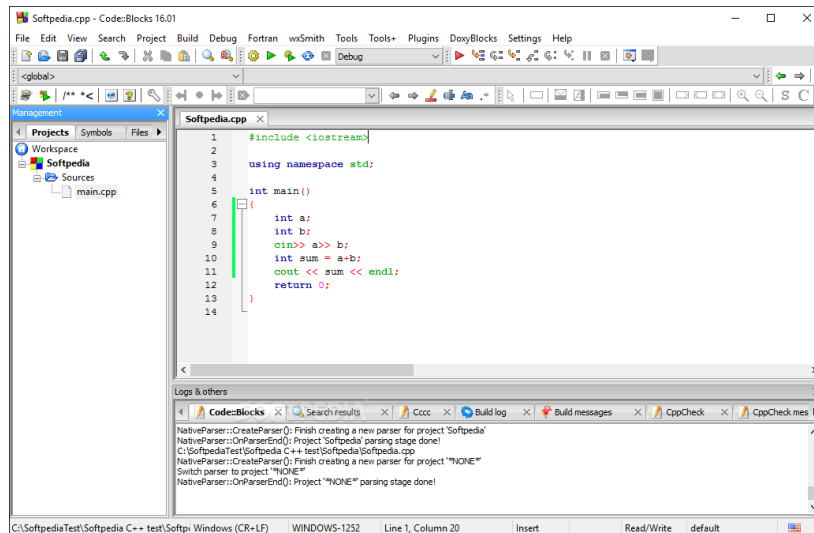
1 Introducción

2 Instalación de IDE para editar y compilar C++

2.1 Code::Blocks [Windows]

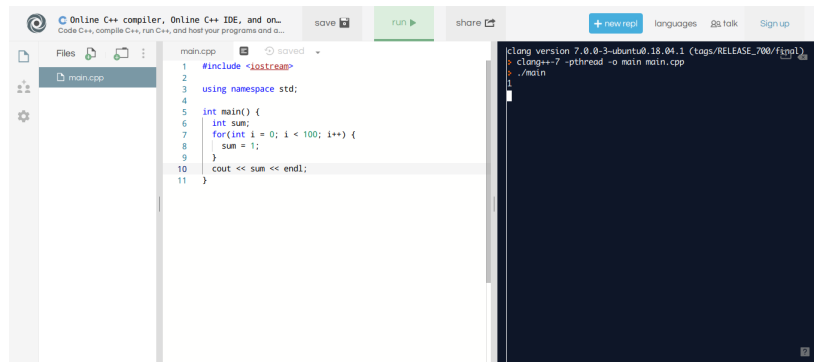
Si desea instalar un programa para compilar código fácilmente, puede descargar Code::Blocks de la página oficial: <http://www.codeblocks.org/>

Este programa permite al usuario escribir y compilar código en Windows sin tener que instalar las diversas herramientas de compilación a mano. Para descargarlo, debe ir a la sección de Downloads y escoger Binaries, y de ahí descargar cualquiera de las opciones que le aparece (le recomendamos la primera opción).



2.2 Repl.it [Página web]

Repl.it es una página web que te permite probar código desde cualquier plataforma, la única desventaja puede ser que ocupes una cuenta para guardar tu código. Para usarlo, debe entrar a la página <https://repl.it/languages/cpp>

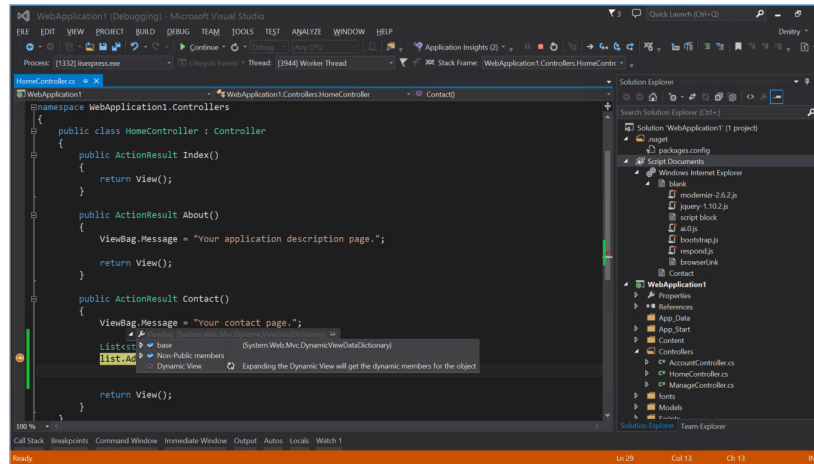


2.3 Visual Studio Community [Windows y Mac]

Esta es la versión gratis del software famoso de Microsoft llamado Visual Studio, te permite compilar código en muchos lenguajes y ofrece herramientas avanzadas de análisis de código.

Se puede descargar en la página oficial <https://visualstudio.microsoft.com/vs/community/> y te permite usarlo siempre y cuando estes ingresado con una cuenta de Microsoft. Si no tienes una cuenta de Microsoft y quisiera utilizar este programa, puedes registrarte en <https://account.microsoft.com/account?lang=es-MX>

A pesar de que esta instalación es más tardado, es una opción bastante profesional y bueno si estas interesado en programar cosas más grandes.

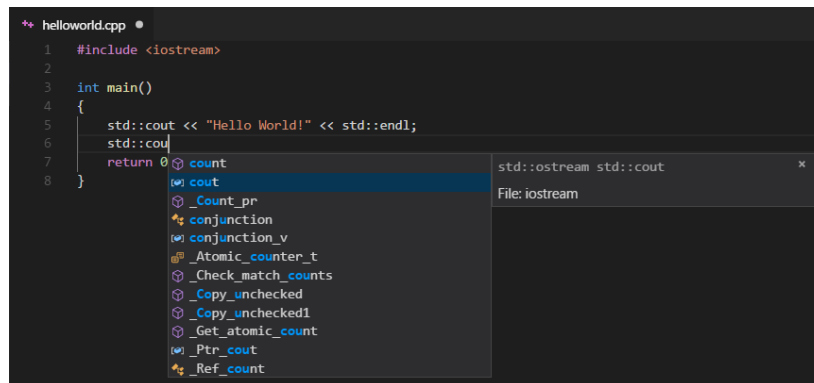


2.4 Visual Studio Code + Terminal [Mac y Linux]

El programa de Visual Studio Code es muy popular para la programación, tiene soporte para todos los lenguajes y es completamente gratis, con la desventaja de que no se puede compilar código.

Si estas en Windows, puedes utilizar este programa pero debes instalar algun emulador de terminal de sistemas Unix como mingw para compilar código.

Si estas usando Mac o Linux, puedes compilar un programa de C++ desde la terminal. Para hacer esto, debes asegurarte de tener instalado el paquete *gcc* y debes generar un archivo ejecutable a partir del código con el comando `c++`



3 Mi primer programa?

4 Leyendo y escribiendo

5 Errores comunes

5.1 Falta de inicialización

Este error es uno de los más graves y más difíciles de detectar debido a que la lógica puede estar completamente correcta, el programa se compila sin errores y en algunos casos tus variables te podrán dar los resultados correctos.

Esto pasa cuando defines una variable pero no le das un valor inicial. Por ejemplo:

Listing 1: Error de inicialización

```
#include <iostream>

using namespace std;

int main() {
    int i = 1;
    int suma;
    while(i <= 100) {
        suma += i;
        i++;
    }
    cout << suma << endl;
}
```

[Liga al código](#)

Este programa suma los números de 1 a 100 y los va guardando en la variable suma. Se esperaría que el resultado fuera 5050, pero si lo corres te despliega -1428474. Si vuelves a correr el código, ahora ves 82934. Este valor siempre cambia.

Esto ocurre debido a que nunca inicializamos o le damos un valor inicial a la variable suma. A la hora de declarar la variable, le estamos asegurando un espacio en la memoria del sistema pero esto no significa que ese espacio tenga guardado un cero. Puede ser que algún proceso anterior haya tenido que usar ese espacio y que haya dejado un valor que no nos interesa ahí.

La razón por la que siempre cambia es porque la memoria siempre esta siendo utilizado por otros procesos (Google Chrome, Word, etc.) y cada vez que corre el programa agarrara el primer espacio libre que encuentra para la variable suma, que no siempre será la misma.

Existe una posibilidad de que tu compilador automaticamente detecte que suma no fue inicializado y le asigna el valor de cero despues, esto puede provocar que el programa funcione en tu computadora pero a la hora de probarlo en otra falla.

Entonces para asegurar que esto nunca pase, siempre debes de tener la costumbre de inicializar tus variables a menos que sabes que serán sobrescritos por otro valor o leidos desde la consola usando *cin*.

5.2 Ciclos con otras variables

Este error pasa muy seguido cuando se utilizan ciclos for anidados y cuando uno olvida que haya declarado una variable antes.

Digamos que vas tan mal en matemáticas que se te olvidaron las tablas multiplicativas por completo, y tu quieres crear un programa que los genera. Puedes hacer esto con dos ciclos for anidados, uno que calcule las filas y otra que calcule las columnas.

Listing 2: Ciclos con otras variables

```
#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i <= 12; i++) {
        for(int j = 0; i <= 12; i++) {
            cout << i * j << "\t";
        }
        cout << endl;
    }
}
```

[Liga al código](#)

Al parecer este código esta bien, pero si miras fijamente verás que en el segundo ciclo estamos modificando *i* en lugar de *j*. El compilador asume que sabes lo que estas haciendo, asi que no te arroja ningun error al tener distintos variables en un solo ciclo.

Para asegurar que esto no pase, siempre debes asegurarte de que solo estes modificando una variable por ciclo y que no hayas usado una variable con ese mismo nombre antes.

5.3 Error por uno

Este error tiende a ser muy facil de arreglar y ocurre por fallas de lógica.

Digamos que quieres encontrar la suma de todos los cuadrados $1^2 + 2^2 + 3^2 \dots + n^2$, y creas un programa para calcularlo.

Listing 3: Error por uno

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    int suma = 0;
    for(int i = 0; i < n; i++) {
        suma += i * i;
    }
    cout << suma << endl;
}
```

[Liga al código](#)

Corres tu programa con $n = 3$, y ya sabes que $1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$, pero te dice que es 5. Ahora corres tu programa con $n = 4$ y ahora observas que la respuesta es 14 cuando debía ser 30. Esto ocurre porque estas contando de 0 a $n - 1$ (nota que el for tiene la expresión $i < n$ cuando debía ser $i \leq n$).

Esto pasó porque el for loop cuenta hasta n pero no lo incluye en la suma, y estos tipos de errores suelen ocurrir cuando se utilizan ciclos que inician desde 1 o cuando se debe incluir el ultimo valor en el proceso.

5.4 Segmentation fault

Este error es algo común y suele pasar cuando quieres sacar un valor que esta fuera de un arreglo.

Digamos que tenemos un código que le pregunta al usuario un valor y luego despliega el factorial de ese valor. El código asume que el usuario seleccionará un número entre 0 y 10, para luego desplegar su factorial el cual esta guardado en un arreglo.

Listing 4: Error de inicialización

```
#include <iostream>

using namespace std;

int main() {
    int factoriales[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800};
    cout << "Que factorial quisieras?" << endl;
    int deseado;
```

```

    cin >> deseado;
    if(deseado < 0) {
        cout << "No_existe_" << deseado << "!";
    } else {
        cout << deseado << "_es_" << factoriales[deseado] << endl;
    }
}

```

[Liga al código](#)

Ahora que pasa si el usuario quiere saber el valor de 25565 factorial? Si ejecutas el código con tal valor verás que el programa se cierra inesperadamente y regresa el siguiente error: [1] 12937 segmentation fault (core dumped).

Esto ocurre porque estas intentando accesar el elemento 25565 de un arreglo que solo tiene 10 valores, lo que pasa es que estas intentando ver un lugar que no existe en la memoria de tu programa.

El 95% de las veces que verás un segmentation fault, será porque estas intentando modificar o ver un valor que esta fuera de un arreglo. Siempre debes asegurarte que un arreglo tiene suficiente espacio para guardar todos los posibles valores que ocupes.

5.5 Sobreflujos

Este error pasa cuando tienes un valor que es demasiado grande para una variable.

Ahora vamos a mejorar el código que habíamos hecho del factorial. En lugar de guardar los primeros 10 valores en un arreglo, vamos a generar cada valor con un ciclo.

Para checar que todo este bien, voy a desplegar los primeros 18 factoriales.

Listing 5: Sobreflujo

```

#include <iostream>

using namespace std;

int main() {
    int factorial = 1;
    for(int i = 1; i < 18; i++) {
        factorial *= i;
        cout << factorial << endl;
    }
}

```

[Liga al código](#)

Corres este código y te topas con una sorpresa, $17!$ es aparentemente -288522240. ¿Cómo fue que un factorial te dio un número negativo si siempre incrementa?

Lo que esta pasando es que el valor de $17!$ es demasiado grande para ser guardado en un valor de tipo `int`, si buscas en línea encontraras que un `int` solo puede guardar valores de -2,147,483,648 a 2,147,483,647, y $17!$ tiene un valor de 355,687,428,096,000.

El programa no te arroja ningun error, simplemente lo que hace es que cuando un número es más grande del valor máximo de 2,147,483,647, se traslapa a -2,147,483,648 y sigue sumando desde ese valor. Eso significa que si quiero guardo el valor de 2,147,483,649 en un `int`, me guardará en realidad el valor -2,147,483,647.

Si sigues imprimiendo más números, verás que se vuelven a hacer positivos y negativos debido a este traslape.

Para solucionar esto, debes reemplazar `int` por otro tipo de dato que pueda guardar números más grandes, como `long long int` o `double`.

5.6 Errores de redondeo

Este error pasa cuando se dividen dos enteros.

Sabes que $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = 1$ y quisieras comprobarlo creando un programa. Lo implementas con el siguiente código:

Listing 6: Error de redondeo

```
#include <iostream>

using namespace std;

int main() {
    int suma = 0;
    int potencia = 2;
    for(int i = 1; i < 30; i++) {
        suma += 1 / potencia;
        potencia *= 2;
    }
    cout << suma << endl;
}
```

[Liga al código](#)

Pero cuando lo corres, te dice que tu suma es cero. Revisas la variable de potencia y te das cuenta que ese sí esta bien, pero la variable de suma se queda en cero.

Esto ocurre debido a que 1 y potencia estan siendo guardados como enteros, y cuando se dividen dos enteros el resultado siempre será otro entero redondeado

para abajo. Es decir si hago $5/2$, esto me dará 2 porque el valor de 2.5 fue redondeado hacia abajo. Como potencia siempre es mayor que 1, esta division siempre se redondeara para abajo y dará cero.

Para corregir esto, se deben cambiar las variables de potencia y de suma de enteros a un tipo de dato flotante, es decir un número que permite puntos decimales.

5.7 Faltas de memoria

Esto sucede cuando tu programa ocupa demasiado espacio continuo y ocasiona un error.

A veces sucede que tienes que crear un arreglo con 10 millones de valores (es verdad, sí vas a ocupar algo así!) y te darás cuenta que tu programa deja de funcionar.

Listing 7: Faltas de memoria

```
#include <iostream>

using namespace std;

int main() {
    int arreglo[10000000];
    for(int i = 0; i < 10000000; i++) {
        arreglo[i] = i;
    }
    cout << "Arreglo_llenado" << endl;
}
```

[Liga al código](#)

Por suerte, esto se puede arreglar. Si cuentas los bytes, te darás cuenta que son 4 millones de bytes o 4 megabytes, y las computadoras modernas tienden a tener hasta 16 gigabytes de memoria RAM, así que no debe haber problema para conseguir 4 megabytes de memoria.

El problema viene del hecho que a la hora de crear un arreglo, el sistema busca 4 megabytes de espacio que estén libres y juntos dentro de la memoria del programa. Esto se puede arreglar preguntándole a tu sistema si puedes guardar estos 4 megabytes en una área especial extra de memoria llamado el heap, que viene siendo una área pública con muchísimo espacio, en lugar del espacio limitado que fue asignado a tu programa.

Para hacer que el programa busque dentro del heap, debes cambiar el arreglo a un puntero y utilizar malloc, new o alguna función que te permite obtener espacio del heap.

6 Tipos de datos

6.1 Signed y unsigned

6.2 Numericos

6.3 Caracteres y cadenas

6.4 Booleanos

6.5 Long long

6.6 Arreglos

7 Funciones

Las funciones son muy utiles para cuando tienes que repitir código similar muchas veces. Estos son similares a funciones matemáticas, reciben cuantas variables de entrada quieres que tengan, haran una operación con estas entradas y regresará una sola variable de salida.

Para una función, se dice que se "regresa" su salida porque para indicar que se quiere terminar la función, se utiliza el operador *return* seguido por el valor de la variable de salida.

Un ejemplo sencillo es una función que suma dos números enteros A y B y que regresa otro entero C, la suma. Esto lo podemos implementar de la siguiente manera:

Listing 8: Función de suma

```
int suma(int a, int b) {  
    int c = a + b;  
    return c;  
}
```

Como se puede ver, se inicia una funcion con un tipo de dato que corresponde a el tipo de dato de la salida. Como int va antes que el nombre de la función (suma), nos dice que queremos que regrese un entero.

Luego, sigue el nombre de la función y dos parentesis. Dentro de los parentesis van las variables de entrada separadas por comas. Como tenemos un entero a y otro entero b, estamos diciendo que para correr esta función el usuario debe dar dos numeros en ese orden, los cuales serán guardados en a y b.

Luego, siguen corchetes que encapsulan el código de la función. Aqui es donde hacemos nuestros cálculos con nuestras variables y regresamos el valor de la salida.

Declaramos otro entero c y le asignamos la suma de a y b, luego lo regresamos, es decir estamos diciendo que queremos que c sea nuestra salida.

Todas las funciones tienen que estar colocados antes de main de tal manera:

Listing 9: Función de suma

```
#include <iostream>

using namespace std;

int suma(int a, int b) {
    int c = a + b;
    return c;
}

int main() {
```

Ahora, podemos llamar nuestra función dentro de main dándole los parámetros que queremos:

Listing 10: Función de suma

```
#include <iostream>

using namespace std;

int suma(int a, int b) {
    int c = a + b;
    return c;
}

int main() {
    cout << suma(3, 5) << endl;
}
```

[Liga al código](#)

Al ejecutar este código, nos desplegará un 8.
Otro ejemplo es una función que regresa la división de dos enteros:

Listing 11: Función de división

```
#include <iostream>

using namespace std;

float divide(int a, int b) {
    return (float)a / b;
}
```

```
int main() {  
    cout << divide(5, 2) << endl;  
}
```

[Liga al código](#)

Este código imprime 2.5, lo que hace es que recibe dos enteros a y b, y luego regresa su división como si fuera un número flotante. El (float) antes del a significa que quieres convertir la división en un número flotante antes de que redondea.