Manual 1 - 2do Torneo de Programación Competitiva

Lions R.C.

Junio 2019

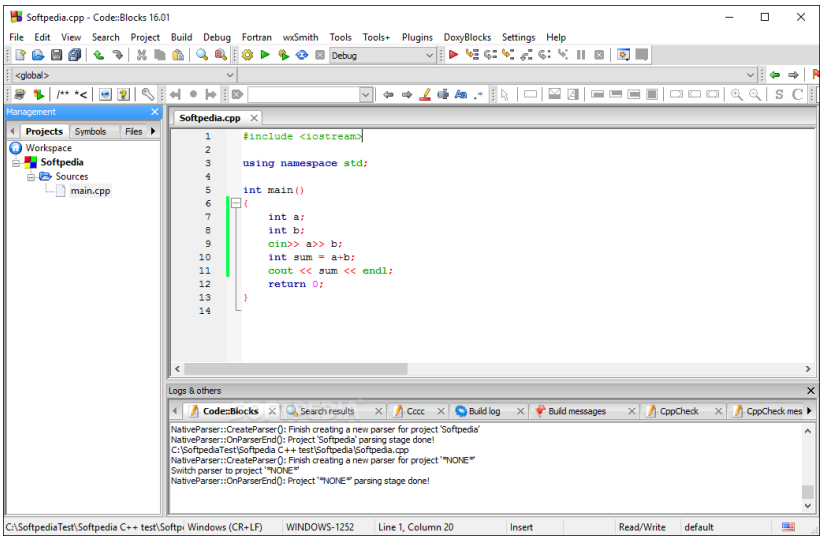
Contents

1 Instalación de IDE para editar y compilar C++

1.1 Code::Blocks [Windows]

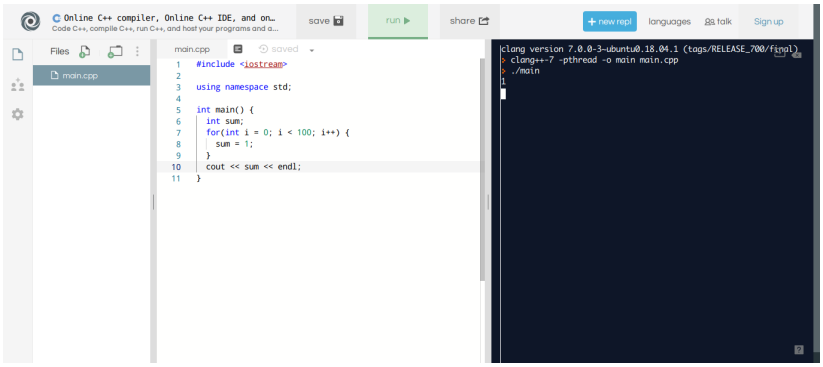
Si desea instalar un programa para compilar código facilmente, puede descargar Code::Blocks de la página oficial: http://www.codeblocks.org/

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 la primera opción (codeblocks-17.12-setup.exe).



1.2 Repl.it [Página web]

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

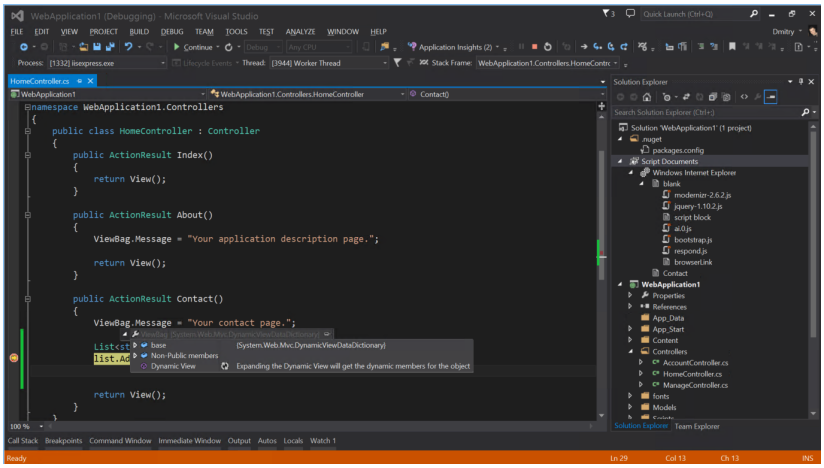


1.3 Visual Studio Community [Windows y Mac]

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 analisis 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 buena si estas interesado en programar cosas más adelante.

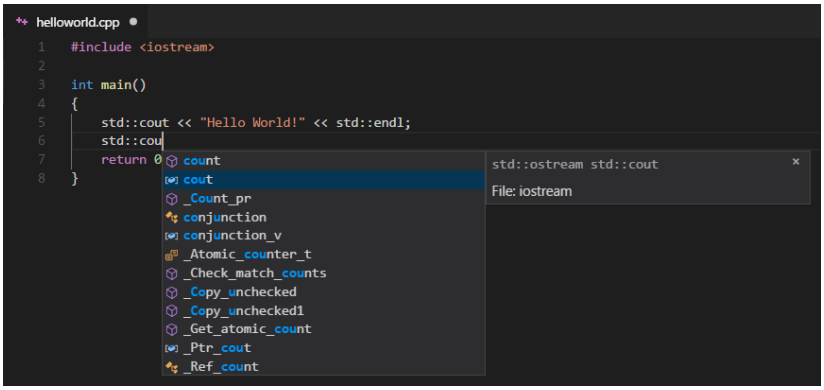


1.4 Visual Studio Code + Terminal [Mac y Linux]

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 puede compilar código.

Si estas en Windows, puedes utilizar este programa pero debes instalar algún emulador de terminal de sistemás Unix como mingw para poder compilar tu 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 $c++$



2 ¿Mi primer programa?

2.1 Donde empezar

Antes de crear un programa, debes asegurarte de tener en mente lo que quieres que haga. Pero esto es algo muy dificil si vas empezando, ¡porque no tienes idea de lo que puedes hacer con un programa!

Asi que durante estas proximas seis semanas, les daremos problemas de programación similares a los que aparecen en competencias más dificiles para que puedan aprender como crear sus propios programas y resolver problemas reales.

El programa más sencillo es uno que no hace nada; se abre y se cierra instantaneamente. Pero eso es aburrido y no tiene sentido. ¿Para qué quiero un programa que no hace nada?

Tú tienes que decirle al programa que quieras que haga, y el programa seguira tus instrucciones sin cuestionarlas, de tal grado que a veces cometera errores porque tú no esperabas que fuera a hacer todo al pie de la letra de como le dijiste.

Vamos a intentar construir un programa que sume dos números. Antes de empezar tu programa tendras que escribir cuatro líneas muy importantes. Si quitas alguno de estos cuatro líneas, tu programa dejara de funcionar y te arrojara un error, y ahorita es muy temprano para poder decirles exactamente lo que hacen estas líneas. Cuando tengan más experiencia, podrán cambiarlos a su gusto.

Listing 1: Las cuatro líneas esenciales

```
#include <iostream>

using namespace std;

int main() {

}
```

Ahí esta su primer programa, él que les dije que se abre y se cierra sin hacer nada. Por ahora, todo su código debe ir entre los dos llaves.

Tenemos que decirle a nuestro programa que queremos dos números enteros, los cuales seran sumados después. Para cada número entero que queremos, tenemos que asignarle un nombre.

Para hacer esto, escribimos una línea para cada variable con int seguido por el nombre de la variable después (sin espacios y sin caracteres especiales). Finalmente, le ponemos un punto y coma después de cada línea para decirle que eso es lo unico que queremos hacer en esa línea.

Listing 2: Dos variables

```
#include <iostream>
```

```
using namespace std;

int main() {
    int a;
    int b;
}
```

Y asi tenemos dos números. Podemos nombrarlos como queramos, por ejemplo: numeroA y numeroB, alfa y beta, n1 y n2, etc. Si corremos el código, no pasa nada. Esto es porque no le hemos dicho al programa que queremos que haga con estos dos números, ni le hemos dicho que valores tienen.

En una línea, podemos escribir a = 5; para decirle que queremos que la variable a valga 5. Tambien es valido modificar la línea int b; para que sea int b = 2;

Listing 3: Guardando valores

```
#include <iostream>

using namespace std;

int main() {
    int a;
    a = 5;
    int b = 2;
}
```

Es de gran importancia saber que el programa lee el código desde arriba hacia abajo, asi que no podemos poner la línea que dice a = 5; antes de int a; porque no sabra que a es una variable que definimos.

Ahora solo falta decirle que queremos que sume a y b.

Listing 4: Guardando valores

```
#include <iostream>

using namespace std;

int main() {
    int a;
    a = 5;
    int b = 2;
    a + b;
}
```

Este nuevo programa parece sumarlos, pero si se corre tampoco pasa nada. Esto es porque tenemos que decirle al programa que además de sumarlos, queremos ver esta suma. Si no, calculará a + b, no sabrá que hacer con esa suma y lo descartará.

Para hacer que nos muestre este dato, tenemos que utilizar algo llamado cout, que no es nada mas que un comando que nos muestre datos.

Para usar cout, se debe escribir cout << valor; donde valor es lo que quieres ver.

Listing 5: Tu primer programa

```
#include <iostream>

using namespace std;

int main() {
    int a;
    a = 5;
    int b = 2;
    cout << a + b;
}
```

Liga al código

Y de esa manera, tenemos nuestro primer programa. Hay miles de maneras de hacer este mismo código, y todos son iguales de validos. Aqui mostraremos algunos ejemplos:

Listing 6: ¿El mismo programa?

```
#include <iostream>

using namespace std;

int main() {
    int a = 5, b = 2;
    cout << a + b;
}
```

Listing 7: ¿El mismo programa?

```
#include <iostream>

using namespace std;

int main() {
    int a;
    int b;
    int c;
    a = 5;
    b = 2;
    c = a + b;
    cout << c;
}
```

Listing 8: ¿El mismo programa?

```
#include <iostream>

using namespace std;

int main() {
    cout << 2 + 5;
}
```

2.2 Comentarios

C++ te permite escribir comentarios en medio de tu código y estos pueden servir para describir lo que hace cierto código, para recordarte de hacer algo en cierta parte o para mantener tu código ordenado.

El tipo de comentario más popular es el de una sola línea, Estos se inician con dos // seguidos por tu nota.

Listing 9: Comentario

```
#include <iostream>

using namespace std;

//Los comentarios pueden ir en lineas separadas
int main() {
    //En cualquier parte del codigo
    int a; //Incluso aqui!
    int b;
    int c; //Esta variable es la suma de a y b
    a = 5;
    b = 2;
    c = a + b; //Aqui se hace la suma
    cout << c; //Aqui se imprime el resultado
}
```

Tambien existen los comentarios de multiples líneas, estos se deben iniciar con /∗ y se deben terminar con ∗/

Listing 10: Comentario

```
#include <iostream>

using namespace std;

/*  Esto es un comentario de multiples lineas
    Puedes escribir cuantas lineas quieras aqui
    Tambien sirve para temporalmente deshabilitar codigo
*/
```

```
int main() {
    int a; /* Tambien debes saber que pueden ocupar una sola linea */
    int b;
    int c;
    a = 5;
    b = 2;
    c = a + b;
    cout << c;
}
```

3 Leyendo y escribiendo

A la hora de compilar y ejecutar tu programa, seguramente notaste que se abre una consola con todo lo que decidiste ver. Esto es porque C++ corre dentro de esa consola (Tu compilador ejecuta comandos para abrir el programa sin que tú veas).

Esto te permite leer y escribir datos mientras que tu programa corre. Ya se ha visto que para escribir datos, se puede utilizar cout (que significa c + out o output de C) seguido por la variable que quieres "imprimir" a la consola.

Para leer datos, se ocupa tener una variable para almacenar ese dato de entrada, luego se debe utilizar cin (que significa c + in o input de C) seguido por esa variable.

Estas dos herramientas te permiten crear programas generales que resuelven problemas especificos.

Digamos que queremos crear un programa que lee dos números enteros (sin importar cuales sean) y que imprime su suma. Podemos crear este programa utilizando cin y cout para que el usuario diga cuales dos números quiere que sume.

Listing 11: Suma

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b << endl;
}
```

Liga al código

Cuando ejecutas tu código, verás que la consola esta esperando una entrada. Puedes ingresar dos números separados por espacios y picarle enter o poner el primer número, picar enter, poner el segundo y picarle enter denuevo. Al

terminar de hacer eso, verás que tu programa hace la suma instantaneamente y que lo despliega debajo de los números que le diste.

4 Operaciones básicas

4.1 Suma, resta, multiplicación, división

Como se sabe, puedes sumar, restar, multiplicar y dividir con los símbolos +, -, * y /.

Además de estos, si quieres guardar el resultado de una operación con una variable en esa misma variable, puedes simplificar la notación utilizando los operadores +=, -=, *= y /=.

int x += 5;
int x -= 8;
int x *= 2;
int x /= 6;

Para sumar y restar solo uno, la notación es aun mas sencilla:

x++;
x–;

4.2 Modulación

Una operación muy util e importante es la que se llama modulación, este se utiliza para sacar el residuo de una división con el simbolo %.

Digamos que queremos el residuo de 13 dividido por 5, esto lo sacamos simplemente con **13 % 5;**

5 Tipos de datos

A comparación de otros lenguajes de programación, en C++ tútienes que definir que tipo de dato quieres guardar en una variable.

5.1 Númericos

En C++, los números enteros y los números decimales se guardan en dos tipos de variables distintos. Esto es porque las variables que manejan números decimales no son completamente precisos (tienen un margen de error), mientras que los datos enteros sí lo son. Además, los números decimales estan representados de una manera completamente distinta en binario que los enteros, y esto tiene sus ventajas y desventajas.

Los números enteros se conocen como **int** o integers, y son capaces de guardar cualquier número desde -2,147,483,648 hasta 2,147,483,647.

Los números decimales se conocen como **float** o números flotantes, en teoria son capaces de guardar números hasta $3.4028235 * 10^{38}$, con la única desventaja que el error incrementa dependiendo del tamaño del número.

También existe una variante de **float** conocido como **double**, que tiene el doble de precisión y es capaz de guardar números hasta 10^{308}

Estos son los tipos de datos númericos más comunes, pero existen más con distintos propositos. Tambien existe **long long int**, una variante de **int** que es capaz de guardar números muchisimo más grandes (desde -9,223,372,036,854,775,808 hasta 9,223,372,036,854,775,807), a cambio que ocupa el doble de memoria.

Tambien existe **long long double**, con más precisión que un **double**.

Raramente, tendrás que hacer cálculos con números más grandes que los permitidos por estos datos, o con mayor precisión. Existen librerias externas que pueden ser utilizados para manejar números de escala inmensa.

5.2 Signed y unsigned

Existe una posibilidad que trabajarás con tipos de datos que son signed y unsigned, y estos definen si una variable puede guardar números negativos o solo positivos.

Todos los tipos de datos númericos empiezan como signed a menos que defines lo contrario. Esto significa que puede guardar números negativos y números positivos. Si quisieras convertir un tipo de dato en unsigned para que solo maneje números positivos, entonces puedes escribir unsigned antes del tipo de dato. Por ejemplo: **unsigned int a = 5;**

A la hora de hacer una variable unsigned, el máximo se convierte en el doble de lo que era antes. Esto es porque la mitad de los números fueron utilizados para representar los números negativos en la representación signed, y esa mitad se junta con la otra mitad para hacer el doble. Es decir, si el mayor número que se puede guardar en un **int** es 2,147,483,647, el mayor número que se puede guardar en un **unsigned int** es 4,294,967,295.

Esto se tiende a usar para manejar datos que se saben que nunca seran negativos, como temperaturas.

5.3 Caracteres y cadenas

C++ es capaz de guardar texto letra por letra en variables que se llaman caracteres y cuerdas. Un carácter o **char** es una variable que contiene una sola letra o simbolo conforme al estandar ASCII, mientras que una cuerda o un **string** es una variable que guarda mensajes completos.

Para asignar un valor a un carácter, se debe colocar una sola comilla antes y después de la letra: **char estrella = '*';**

Para asignar un valor a un string, se debe utilizar doble comilla antes y después del texto: **string mensaje = "Hola mundo!";**

5.4 Booleanos

Estos tipos de datos son muy utiles porque solo guardan dos valores: verdadero o **true** y falso o **false**.

Para asignar un booleano, se utiliza el termino **bool**. Por ejemplo: **bool decirMentira = false;**

5.5 Arreglos

Cuando estamos trabajando con muchas variables a la vez, las cosas se pueden complicar y se puede requerir guardar un número grande de cosas. Para no tener que escribir cientos de variables para una sola tarea, se pueden crear arreglos, que guardan una cantidad predeterminada de datos.

Se pueden hacer arreglos de cualquier tipo de dato y su nomenclatura consiste en poner dos corchetes [después del nombre con el número de elementos que se desea tener en ese arreglo.

Digamos que queremos un arreglo de 100 enteros, eso se podría definir como **int valores[100];**

Luego, para modificar o leer uno de estos valores, se puede llamar **valores[N]** donde N es el indice del elemento.

Es muy importante mencionar que el primer elemento tiene indice 0 y el ultimo elemento tendría indice 99. Los arreglos siempre empiezan desde cero y incrementan hasta el valor del tamaño menos uno. Si se intenta guardar o ver un valor que no esta en este rango, podrá causar errores en su programa.

Las cuerdas funcionan como arreglos, y se puede ver el carácter en la posición N utilizando el mismo sintaxis.

6 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 12: Función de suma

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

Como se puede ver, se inicia una función con un tipo de dato que corresponde al 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 paréntesis. Dentro de los paréntesis 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 números en ese orden, los cuales serán guardados en a y b.

Luego, siguen llaves 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 13: 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 dandole los parametros que queremos:

Listing 14: 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 despliegara un 8.

Otro ejemplo es una función que regresa la división de dos enteros:

Listing 15: 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.

7 Flujo de datos

La pieza fundamental detrás de cualquier programa es su lógica, y no puede haber lógica sin flujo de datos. El flujo de datos consiste en correr ciertos piezas de código bajo ciertas condiciones. Por ejemplo, si queremos saber si un número es par el código debe de tener dos secciones, uno que corre cuando sí es par y otro que corre cuando no es par.

7.1 Condicional if

La manera mas sencilla de comparar datos es con el condicional **if**. Este es capaz de correr código solo cuando se cumple cierta condición. Para utilizar este, se debe iniciar escribiendo **if** y se debe incluir una condición entre paréntesis, luego el código que se desea correr entre llaves.

Listing 16: Condiciones

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a + b > 5) {
        cout << "La suma es mayor a cinco" << endl;
    }
```

```
}
```

Aqui, tenemos una condición que checa si la suma de A y B es mayor a 5. En el caso de que sí se cumple, se imprimirá "La suma es mayor a cinco" y en caso que no, no se imprimirá nada.

Ahora modificaremos el código para que se imprima "La suma es menor o igual que cinco" cuando no se cumple la primera condición.

Listing 17: Condiciones

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a + b > 5) {
        cout << "La suma es mayor a cinco" << endl;
    }
    if(a + b <= 5) {
        cout << "La suma es menor o igual a cinco" << endl;
    }
}
```

Podemos simplificar este código con el condicional **else**, que se ejecuta cuando una condición **if** NO se cumple. Este se puede agregar después de la ultima llave y deberá tener otro par de llaves después.

Listing 18: Else

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a + b > 5) {
        cout << "La suma es mayor a cinco" << endl;
    } else {
        cout << "La suma es menor o igual a cinco" << endl;
    }
}
```

Digamos que ahora queremos checar si la suma es menor, igual o mayor que cinco. Se puede agregar un tercer condicional **else if** que ocurre cuando la primera condición falla pero otra condición se cumple. Se puede tener varios **else if** encadenados.

Listing 19: Else if

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    if(a + b > 5) {
        cout << "La suma es mayor a cinco" << endl;
    } else if (a + b == 5) {
        cout << "La suma es igual a cinco" << endl;
    } else {
        cout << "La suma es menor a cinco" << endl;
    }
}
```

En este caso, primero se checa si el número es mayor que cinco, luego se checa si es igual a cinco, y finalmente por lógica si no cumple ninguna de estas dos condiciones debe ser menor que cinco.

7.2 Condiciones

Como se ha visto, existen maneras de comparar datos para verificar condiciones. Para checar si una variable es menor que otro, se debe usar el simbolo <, mientras que para checar si es mayor, se usa el simbolo >.

Tambien se puede ver cuando un valor es menor o igual a otro valor con <= y cuando es mayor o igual con >=. Nota que el signo de igual siempre va despues de la comparación de mayor o menor.

Para preguntar si dos valores son iguales, se utilizan dos signos de igual **==**, ya que al utilizar solo uno se estaria asignando un valor a la variable de la izquierda. Para checar si no son iguales, se debe usar un signo de admiración seguido por un signo de igual **!=**.

Finalmente, existen los operadores lógicos **&&** y || que pueden ser utilizados para juntar condiciones. El primero se utiliza para correr código siempre y cuando las condiciones pegadas a él sean ciertos, mientras que el segundo sirve cuando cualquiera de los dos son ciertos.

Listing 20: Condiciones

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
```

```
    if(a == b) {
        cout << "A es igual a B" << endl;
    }
    if(a > 5) {
        cout << "A es mayor a 5" << endl;
    }
    if(b <= 10) {
        cout << "B es menor o igual que 10" << endl;
    }
    if(b != -6) {
        cout << "B no es igual a -6" << endl;
    }
    if(a < 1 && b > 1) {
        cout << "A es menor que 1 y B es mayor que 1" << endl;
    }
    if(a == 1 || b == 1) {
        cout << "Una de las variables A o B es igual a 1" << endl;
    }
}
```

Liga al código

7.3 Ciclo for

Digamos que quieres repetir un proceso cierta cantidad de veces pero no quieres tener que volver a escribir el código cientos de veces con cambios pequeños. Se puede automatizar este proceso con el uso de ciclos for.

Un ciclo for repite código cierta cantidad de veces y tiene una variable que incrementa o decrementa con cada turno del ciclo, esta variable puede ser util porque puede indicar el ciclo actual.

Digamos que ocupas saber la suma de 1 a 100. Esto se puede escribiendo cien líneas que suman cada valor o escribiendo una línea dentro de un ciclo for.

Aqui les mostramos el ejemplo con cien líneas:

Listing 21: Suma línea por línea

```
#include <iostream>

using namespace std;

int main() {
    int suma = 0;
    suma += 1;
    suma += 2;
    suma += 3;
    ...
```

```
    suma += 98;
    suma += 99;
    suma += 100;
    cout << suma << endl;
}
```

Y esto es la simplificación con el ciclo for:

Listing 22: Suma con ciclo for

```
#include <iostream>

using namespace std;

int main() {
    int suma = 0;
    for(int i = 1; i <= 100; i++) {
        suma += i;
    }
    cout << suma;
}
```

Como se puede ver, el ciclo for ahorra muchas líneas y puedes cambiar los parametros de iteración.

Se debe notar que el for tiene un sintaxis similar al if, excepto que tiene tres divisiones en la parte condicional.

La primera parte corresponde a la asignación del valor inicial, aqui se debe declarar e igualar una variable al valor inicial del ciclo. A esta variable se le tiende a dar el nombre **i**, o **j** cuando existe otro ciclo que este usando **i** al mismo tiempo.

Despues, sigue una condición que al no cumplirse terminará el ciclo. En este ejemplo, se utilizó **i <= 100** para asegurarnos que una vez que **i** llegue a 100, se termine el ciclo.

Finalmente, la tercera parte debe modificar la variable del ciclo para que progrese hasta que deje de cumplirse la condición en la segunda parte.

Con esta información, podemos ver que primero se guarda 1 en la variable **i**, luego se checa en cada iteración del ciclo que no sea mayor que 100, luego se suma 1 a la variable **i**.

Si queremos hacer un ciclo que nos de los números 0 a 4 en orden ascendiente, podemos modificar el código de la siguiente manera:

Listing 23: Ciclo for

```
#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i < 5; i++) {
```

```
        cout << i << endl;
    }
}
```

Y en caso de que queremos que nos den los valores de 4 a 0 en orden descendiente, podemos modificarlo para que reste en lugar de sumar.

Listing 24: Ciclo for

```
#include <iostream>

using namespace std;

int main() {
    for(int i = 4; i >= 0; i--) {
        cout << i << endl;
    }
}
```

7.4 Ciclo while

Si no te interesa la cantidad de veces que corra un ciclo y simplemente quieres repetir algo mientras que una condición sea cierto, se puede utilizar el ciclo while.

Se debe de tener cuidado con los ciclos while debido a que pueden correrse infinitamente.

Para crear un ciclo while, se utiliza el mismo sintaxis que el condicional if con el nombre **while**. El código dentro de las llaves seguira corriendo hasta que la condición sea falsa.

Listing 25: Ciclo while

```
#include <iostream>

using namespace std;

int main() {
    int potenciaDos = 1;
    while(potenciaDos < 100) {
        potenciaDos *= 2;
    }
    cout << potenciaDos << endl;
}
```

Como se ve en este ejemplo, el código calculará todas las potencias de dos hasta llegar a uno que no sea menor a 100, en este caso 128, luego lo imprimirá en la consola.

7.5 Break

Si tienes algún ciclo y quisieras salirte de ese ciclo bajo una condición que no esta incluida en la parte donde se declara ese ciclo, se puede utilizar el termino **break**.

Digamos que queremos hacer un ciclo que termine después de 100 iteraciones o después de que se encuentren tres números consecutivos que suman 20. Esto se puede implementar con un ciclo for y un break de la siguiente manera:

Listing 26: Break

```
#include <iostream>

using namespace std;

int main() {
    bool existeSuma = false;
    for(int i = 0; i < 100; i++) {
        int suma = i + (i + 1) + (i + 2);
        if(suma == 20) {
            existeSuma = true;
            cout << "La suma es: " << endl;
            cout << i << ", " << i + 1 << ", " << i + 2 << endl;
            break;
        }
    }
    if(existeSuma == false) {
        cout << "La suma no existe!" << endl;
    }
}
```

Liga al código

Como se puede ver, combinamos break con un booleano para ver si se llega a cumplir la condición de la suma, y en caso de que sí se cumple imprimimos la suma y dejamos de buscar más secuencias. En el caso contrario, no modificamos el booleano e imprimimos que la suma no existe.

7.6 Continue

A veces queremos evitar correr cierto código bajo una condición, pero no queremos tener que incluir nuestro código en muchas condicionales if. En un ciclo for o while se puede utilizar **continue** cada vez que se quiere saltar al siguiente ciclo.

Digamos que estamos iterando sobre todos los números menores a 100 y queremos saltarnos todos los números pares y divisibles entre 3. Esto se puede hacer de distintas maneras:

En la primera se puede tener un solo caso if con dos condiciones unidos con el operador &&

Listing 27: Condiciones con and

```
#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i < 100; i++) {
        if(i % 2 != 0 && i % 3 != 0) {
            cout << i << endl;
        }
    }
}
```

Para la segunda manera se puede tener dos ifs anidados:

Listing 28: Dos condiciones

```
#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i < 100; i++) {
        if(i % 2 != 0) {
            if(i % 3 != 0) {
                cout << i << endl;
            }
        }
    }
}
```

Y para la tercera podemos usar dos casos if con continue:

Listing 29: Continue

```
#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i < 100; i++) {
        if(i % 2 == 0) {
            continue;
        }
        if(i % 3 == 0) {
            continue;
```

```
        }
        cout << i << endl;
    }
}
```

Liga al código

Al correr estos tres códigos, se veran los mismos resultados.

8 Errores comunes

8.1 Falta de inicialización

Este error es uno de los más graves y más dificiles 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 30: 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 después, 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 sobreescritos por otro valor o leidos desde la consola usando *cin*.

8.2 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 túquieres 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 31: 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.

8.3 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 $1^2 + 2^2 + 3^2 ... + n^2$, y creas un programa para calcularlo.

Listing 32: 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 ¡= 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.

8.4 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 33: 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 qué 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.

8.5 Sobreflujos

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

Ahora vamos a mejorar el código que habiamos 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 34: 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.

8.6 Errores de redondeo

Este error pasa cuando se dividen dos enteros.

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

Listing 35: 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 división 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.

8.7 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 asi!) y te darás cuenta que tu programa deja de funcionar.

Listing 36: 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, asi 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 preguntandole a tu sistema si puedes guardar estos 4 megabytes en una area especial extra de memoria llamado el heap, que viene siendo una area publica con muchisimo 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.

9 Otros materiales

Si quisieras saber más acerca de como programar en C++, existen muchas páginas y canales de Youtube que explican sus temas a detalle. Aqui mencionaremos algunos ejemplos que nos han servido:

9.1 TutorialsPoint [Inglés]

Esta página es bastante buena porque condensa muchos detalles en pequeñas páginas detalladas.

https://www.tutorialspoint.com/cplusplus/

9.2 HackerRank [Inglés]

Es un canal de Youtube con un playlist de algoritmos y otro playlist de estructuras de datos bastante buena debido a la manera gráfica en la que enseñan las temas.

HackerRank - Algoritmos

HackerRank - Estructuras de Datos

9.3 CPlusPlus [Inglés]

Esta es una página dedicada a documentar todo lo que se relaciona con el lenguaje de C++ y sirve como una referencia avanzada que describe detalles desconocidos por la mayoria de sus usuarios. Además tiene documentación completa de todas las funciones, librerias y tipos de datos.

http://www.cplusplus.com/doc/tutorial/