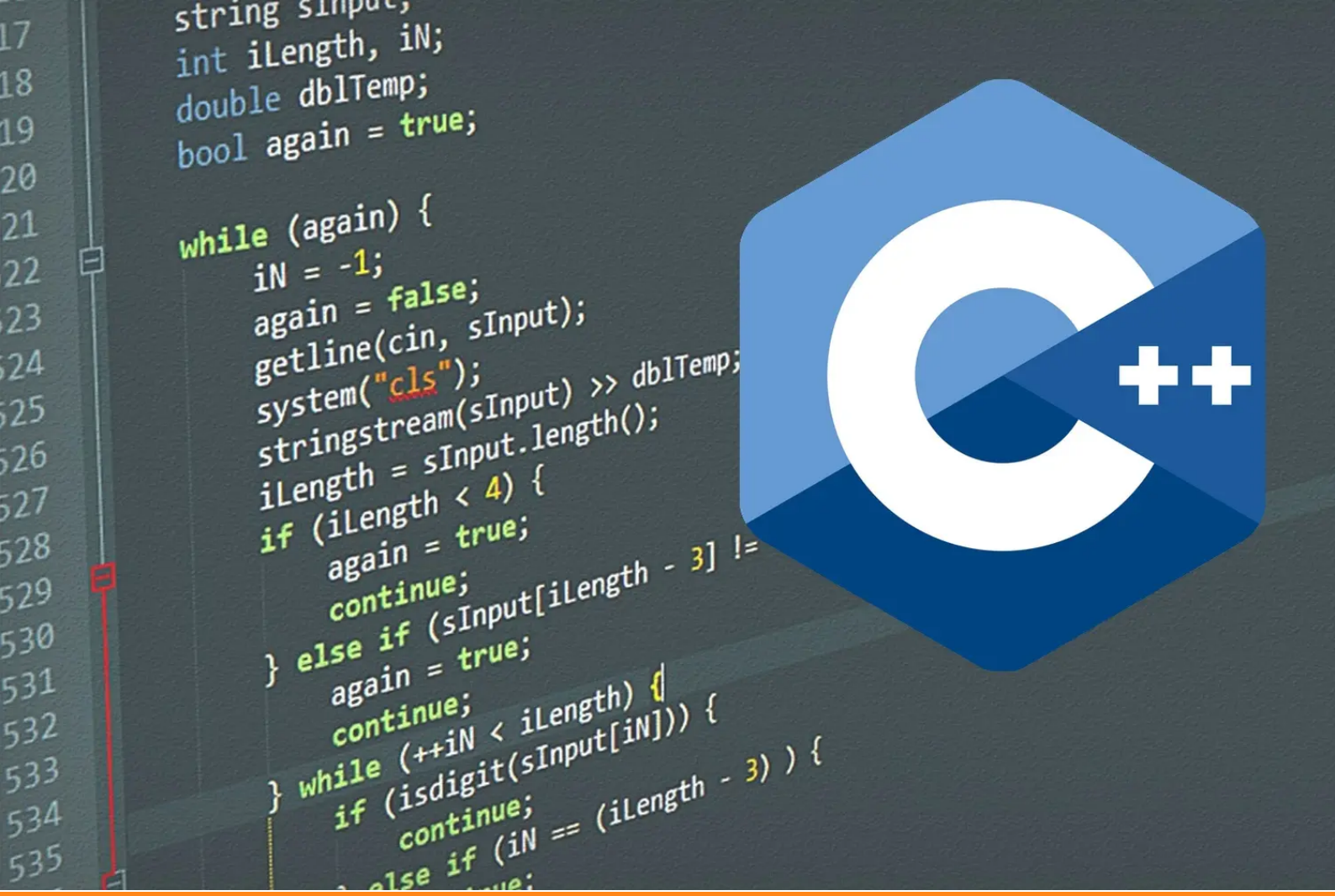

I²COM

Material introductorio para optimizar su desempeño en la materia

Cátedra de Introducción al Cómputo

Instituto Balseiro

Julio 2022

No se confíe, la materia es significativamente mas profunda que esto.

Contenido

1 Introducción

En primer lugar, felicitaciones por lograr ingresar al IB, sin dudas uno de los institutos educativos de mayor nivel y prestigio de la Argentina. Delante les espera una camino lleno de desafíos no menores a los que ya han superado, siendo esta materia uno de los primeros. Debido al rápido avance de la misma, y para optimizar su capacidad de absorber la información durante las primeras clases, les acercamos esta introducción a la materia. Leer este documento a consciencia y reproducir los ejemplos dados les dará una ventaja significativa, así que le recomendamos fuertemente que lo hagan.

1.1 Introducción al Cómputo

De las herramientas que fuimos creando a lo largo del tiempo, la computadora es, sin dudas, la más versátil de todas. Esta versatilidad se debe a su capacidad de realizar una infinidad de tareas distintas, mediante la combinación inteligente de operaciones relativamente simples. Y esto se logra mediante la programación.

El objetivo de esta materia es enseñarles la puerta al mundo de la programación. Algunos tal vez ya hayan estado incursionando, mientras que para otros será la primera vez que escriban un programa y lo vean correr. En ambos casos, les recomendamos que lean todo lo que sigue. Una vez comenzada la cursada, van a notar lo rápido del avance, y tener los conceptos fundamentales bien claros puede ser la diferencia entre seguir adelante o quedarse atrás.

1.1.1 Los lenguajes de programación

Todo el poder de las computadores se basa en un elemento relativamente simple: el transistor. El transistor tiene 3 conexiones con el mundo exterior: entrada, salida y compuerta. Dependiendo del estado de la compuerta, se definirá el comportamiento entre entrada y salida. Y ahí está la verdadera base de la programación: decidir qué estado ponerle a cada compuerta de cada transistor de cada procesador. Obviamente, imaginen que hacerlo a mano es inviable, por decirlo de alguna forma. Por lo tanto, durante la evolución de esta tecnología fueron surgiendo herramientas para facilitarle la vida a los programadores. Y estas herramientas son los múltiples lenguajes de programación que existen. Descartando los más viejos, de los cuales ya pocos escucharon hablar alguna vez, el lenguaje más paradigmático fue sin dudas C. Ahora bien, como toda herramienta creada por la humanidad, eventualmente se usa para crear otras más complejas y específicas. Y así, a partir de C surgieron lenguajes que hoy son ampliamente usados en distintos ámbitos. Entre ellos se encuentra C++, que es el lenguaje que utilizaremos a lo largo de esta materia.

1.1.2 Por qué C++?

Siendo que la idea es que aprendan las bases necesarias para poder encarar cualquier lenguaje de programación de los que se utilizan actualmente, durante años la materia fue dictada utilizando C. Sin embargo, se decidió cambiar a C++, debido a que como lenguaje aún preserva la eficiencia de un lenguaje de bajo nivel, pero se puede utilizar de forma similar a uno de más alto nivel (como Python) si uno quiere, lo que lo hace mucho más cómodo. La cuestión es que una vez que aprenden un lenguaje de bajo nivel, es sencillo aprender uno de alto nivel; al revés no tanto.

1.2 Pienso, luego escribo

1.2.1 Algoritmos

Antes de empezar a aprender cualquier lenguaje de programación, hay que aprender a pensar en algoritmos. Si no sabe qué es un algoritmo, es simple: una lista de pasos a seguir. Por ejemplo, si quiere que alguien encuentre el Instituto Balseiro (sin pasarle la ubicación de Google Maps), podría decirle algo por el estilo:

1. entrar a Bariloche;
2. seguir 20 de Octubre hasta su intersección con Bustillo;
3. seguir Bustillo hasta el km 9.5;
4. consultar a los guardias sobre cómo ingresar.

Este es un algoritmo para encontrar el IB. Cualquier persona que sepa leer español puede seguir las instrucciones; y si las sigue va a llegar. Un programa es un algoritmo, pero escrito en algún lenguaje de programación, en vez de en español. Simple, no?

Lo que no es tan simple es elegir el mejor algoritmo... De hecho existen casi infinitos algoritmos posibles para llegar al IB. Por ejemplo, otro algoritmo podría ser:

1. entrar a Bariloche;
2. seguir 20 de Octubre hasta Diagonal Capraro;
3. seguir Diagonal Capraro hasta su intersección con Moreno;
4. seguir Moreno hasta que se convierte en San Martín;
5. seguir San Martín hasta el obelisco;
6. agarrar Bustillo hasta el km 9.5;
7. consultar a los guardias sobre cómo ingresar.

Este algoritmo también es correcto, y si lo siguen al pie de la letra, van a llegar al IB. Pero es un poco más rebuscado, más propenso a equivocaciones. Y elegir el algoritmo correcto es justamente decidir, entre todos los posibles, el que sea más eficiente y menos propenso a errores.

Bueno, si es cuestión de que sea lo más rápido y menos propenso a errores posible, por qué no este:

1. entrar a Bariloche;
2. llamar un taxi;
3. decirle que lo lleve al Centro Atómico Bariloche;
4. consultar a los guardias sobre cómo ingresar.

A primera vista es el mejor, no? No hay forma de equivocarse, y el taxista conoce mejor que nadie el camino más rápido y corto. Efectivamente, pero le va a costar más dinero. Dependiendo de cuánto disponga, puede ser que le convenga este algoritmo, o el primero (y caminar un rato largo si no tiene vehículo), o el segundo si además de llegar quiere conocer un poco del centro... Y así ocurre siempre con los algoritmos, sean de lo que sean. Mientras más complejo el problema, más opciones hay, y más cosas hay que considerar antes de elegir uno. Así que para diseñar un buen algoritmo/programa, antes debe pensar en lo siguiente:

- Qué resultado quiero obtener?
- De cuántos recursos físicos dispongo?
- De cuánto tiempo dispongo?
- Quién va a utilizarlo?

Lo primero es obvio; lo segundo más o menos; lo siguiente es entendible: si debe crear un programa que va a correr dentro de un satélite, más le vale que haga lo que debe sin cometer errores y que use la menor cantidad de memoria y energía posible, sin importar demasiado el tiempo que le lleve programarlo. En cambio, si va a escribir un programa durante un final, ya no le va a importar tanto la eficiencia, pero si el tiempo que le lleve terminarlo... Y lo último? Eso tiene que ver más que nada con la prolijidad con la cuál escribe. Si sólo yo uso mi programa, yo me entiendo, no? Bueno, esa es una mala práctica que todos cometemos en la privacidad de nuestras PCs, pero que durante esta cursada deberá evitar, ya que nos toca a nosotros revisarlo!

1.2.2 Pseudocódigo

En la frontera entre un algoritmo y un programa propiamente dicho, existe algo llamado pseudocódigo. Qué es? Básicamente es ponerle el nombre que se nos de la gana a las funciones que pensamos usar para llevar a cabo nuestro algoritmo. Lo mejor para entender esto es un ejemplo: suponga que quiere escribir un programa que sume dos números que son ingresados por el usuario, y que muestre el resultado en la pantalla. El algoritmo que resuelve esto podría ser:

1. iniciar programa;
2. solicitar ingreso del primer número;
3. guardar el valor en una variable x;
4. solicitar ingreso del segundo número;
5. guardar el valor en una variable y;
6. calcular la suma x + y;
7. guardar el resultado en una variable z;

8. mostrar el valor de z;
9. terminar programa.

Y en pseudocódigo, podría ser:

```
inicio(programa_suma)
    pedir(x);
    leer(x);
    pedir(y);
    leer(y);
    z=sumar(x,y);
    mostrar(z);
fin(programa_suma)
```

Fácil, no? Otra forma de decir lo mismo podría ser:

```
programa_suma
{
    solicitar(x);
    guardar(x);
    solicitar(y);
    guardar(y);
    z=hacer_suma(x,y);
    imprimir(z);
}
```

O sea, el pseudocódigo es una representación PERSONAL de una idea. No importa qué palabras elija, mientras se entienda lo que debería hacer. Es el paso previo a escribir el programa, y es algo que generalmente uno hace mentalmente, aunque a veces si se complica puede que necesite escribirlo. Para pasar de pseudocódigo a código, sólo hace falta conocer el vocabulario y sintaxis correspondientes al lenguaje elegido! (y algunos detalles más que ya veremos más adelante).

Les dejamos un ejemplo adicional para mostrar otra cuestión fundamental de la programación: divide y conquista! Suena trillado, pero lo que significa es importante: un problema, por mas complicado que sea, siempre puede ser subdividido en problemas cada vez más chicos y simples de resolver.

Se desea escribir un programa que pregunte el nombre del usuario y lo salude.

Algoritmo:

1. iniciar programa;
2. solicitar nombre de usuario;
3. guardar nombre de usuario en variable N;
4. imprimir en pantalla “Hola ”;
5. imprimir en pantalla el valor de N;

6. finalizar programa;

Pseudocódigo:

```
programa_saludo
{
    N = solicitar_nombre;
    imprimir(‘‘Hola ’’);
    imprimir(N);
}

solicitar_nombre
{
   imprimir(‘‘Quien anda ahí?’’);
   leer(n);
   devolver(n);
}
```

Este es un pseudocódigo formado por un (pseudo)programa principal y una (pseudo)función. La (pseudo)función es llamada por el (pseudo)programa principal, a lo cual realiza las acciones de imprimir texto, leer n, y devolver el valor de n al (pseudo)programa principal, que lo guarda en N, para luego utilizarlo. Esto puede sonar rebuscado la primera vez; relea y piense lo que significa antes de seguir. La subdivisión de tareas es fundamental a la programación, téngalo siempre presente.

1.3 Representación

Si bien no es un requisito exclusivamente necesario para poder programar, entender la forma en que se representan los números y símbolos en general en la computadora, es una ventaja a la hora de generar código eficiente. Todo se basa en el bit, que puede tomar dos valores posibles: 0 y 1. Si bien puede tomar esos únicos valores, el bit puede usarse para representar números tan grandes como uno quiera. Cómo? Combinando bits en un número predefinido. Por ejemplo, la combinación más conocida es el byte, que consta de 8 bits. Y cuántos números/símbolos posibles pueden ser representados con 1 byte? $2^8 = 256$. Eso alcanza, por ejemplo, para representar las 26 letras del alfabeto inglés en minúsculas, las mayúsculas, los 10 números que conocemos, y un montón de símbolos más (busque código ASCII, si tiene curiosidad). Es decir, a cada combinación de 8 bits le asignamos un símbolo, de forma que, por ejemplo ’01000001’ = ’A’, ’01100001’ = ’a’, ’01000000’ = ’@’. Si alguna vez utilizó el comando alt+64, debería entender qué significa ese 64. Sino, ya lo va a entender.

Además de poder representar símbolos, los bits pueden usarse para representar números, tanto enteros cómo de punto flotante. Para entender cómo, repasemos primero lo que significa

la base de un sistema numérico.

Estamos acostumbrados a vivir en un mundo en base 10, así que es natural si el código binario es un poco confuso al comienzo. Pero qué significa esto de "base 10"? Que normalmente representamos los números mediante 10 símbolos diferentes, y que todo número puede representarse como una suma de potencias en base 10. Con un ejemplo quedará más claro: $123456.7 = 1\text{x}10^5 + 2\text{x}10^4 + 3\text{x}10^3 + 4\text{x}10^2 + 5\text{x}10^1 + 6\text{x}10^0 + 7\text{x}10^{-1}$. Fíjese cómo la representación '123456.7' se obtiene de tomar los coeficientes de cada elemento de la sumatoria y ponerlos concatenados (teniendo el cuidado de poner el punto al pasar a exponente negativo). Y cómo sería con 254? $2\text{x}10^2 + 5\text{x}10^1 + 4\text{x}10^0$. En esta base aprendimos desde que nacimos, y nos es tan natural, que olvidamos que la elección fue arbitraria; bueno, tan arbitrario como el hecho de que tenemos 10 dedos. Y cómo sería la representación en base 2? Imagine que sólo existieran dos símbolos numéricos: el 1 y el 0. Intentemos representar 254 (que así escrito esta en base 10), pero en base 2. Para ello, debemos poder obtener ese valor, pero sumando sólo potencias de 2, en vez de potencias de 10: $254 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 0$ Reescribiendo: $254 = 1\text{x}2^7 + 1\text{x}2^6 + 1\text{x}2^5 + 1\text{x}2^4 + 1\text{x}2^3 + 1\text{x}2^2 + 1\text{x}2^1 + 0\text{x}2^0$. Y entonces, siguiendo la lógica que usamos para la representación en base 10, si tomamos los coeficientes de la sumatoria y los concatenamos, nos da la representación en base 2, de 254: '11111110'. Preste atención al orden de los coeficientes, siempre a la izquierda van los de mayor orden, eso es así para todas las bases que se le ocurran.

Alguno se podrá estar preguntando, hay más bases además de 2 y 10 que se utilicen con frecuencia? De hecho sí, aunque en general no tendrá que trabajar con ellas, es importante que sepa que existen al menos estas: base 8 (representación octal) y base 16 (hexadecimal). Para representaciones con bases mayores a 10, nos quedamos sin símbolos numéricos preestablecidos, así que cómo lo solucionamos? Usando lo que ya tenemos: letras. Así que en base 16, los símbolos numéricos correspondientes son: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$. Esto significa que $A = 10, B = 11, C = 12, D = 13, E = 14, F = 15$. Tratemos de representar 254 en base 16: $254 = 15\text{x}16^1 + 14\text{x}16^0 = F\text{x}16^1 + E\text{x}16^0$. Es decir, 254 en base 16 se representa como 'FE'. Sabe en qué cosa relativamente cotidiana se utiliza la representación hexadecimal? El código de colores RGB (red, green, blue) consta de una escala que va de 0 a 255 para cada color. Por ejemplo, el rojo puro y totalmente saturado tendría, en representación decimal, el valor 255000000. En hexadecimal es más compacto: $FF0000$.

Llegados a este punto, retomemos un poco el tema de algoritmos. Cómo escribiría un algoritmo para pasar de una representación en base 10, a una en base b? Piénselo.

1.3.1 Representación de punto flotante: IEEE 754

A la hora de representar números con decimales, hay que tener en cuenta los requerimientos necesarios. Si sólo cambiamos de base 10 a binario, y tratamos de guardar ese número en la memoria, vamos a notar que el lugar necesario crece al aumentar la cantidad de decimales. Esto

es muy ineficiente, ya que no sólo se ocupa memoria en guardar ceros innecesariamente, sino que además la precisión es dependiente del número, y no de la representación! Este problema se solucionó hace muchos años a través de un estándar, el IEEE 754 a través de 2 formatos para representar números con distintas precisiones: precisión simple y precisión doble, y lo que se hace para obtener una representación de simple (o doble) precisión es: en el primer bit se guarda el signo s, los siguientes 8 (11) son para representar el exponente e y los últimos 23 (52) son para la fracción o mantisa m, descartando el 1 al comienzo, que siempre es igual; es decir, quedándose sólo con los decimales. Todos estos valores deben estar ya pasados a base 2, cuidado con eso. Entonces, el número que representamos mediante estos 32 (64) bits se puede obtener: $(-1)^s \text{x} 2^{(e-offset)} \text{x} (1+m)$, donde el $offset$ es necesario para poder representar exponentes positivos y negativos.

Este estándar para representar números de punto flotante es importante saber que existe y entender por qué, y cómo funciona. Y saber calcularlo de memoria? No realmente.

2 Empezando a programar

Para obtener un programa funcionando, hay varias etapas que se deben recorrer. En primer lugar, se debe escribir el código que guarda las instrucciones. Pero así como está al escribir un programa en un lenguaje determinado, la máquina no va a entender qué hacer. Para que pueda seguir las instrucciones es necesario primero traducir este lenguaje, a otro que la computadora sí pueda "entender". De eso se encarga el compilador. Además se encarga de buscar las bibliotecas utilizadas (conjunto de funciones que alguien ya programó, y que nos facilitan la vida enormemente), y conectarlas con nuestro programa. Así, si todo funciona bien, al final el compilador creará un archivo ejecutable. Ese es el producto final, el set de instrucciones en lenguaje de máquina que surge de nuestro código. Y si surgen errores durante la compilación? Entonces no se creará ese ejecutable, y el compilador nos dará pistas sobre qué hicimos mal, dónde está el error. Una vez corregidos los errores, volvemos a compilar, hasta que se logre generar el ejecutable. Pero incluso cuando tengamos el ejecutable funcionando, puede ser que haya algún error en las instrucciones que escribimos, y que el programa funcione de forma no deseada. A este tipo de errores que pasan la compilación, se los suele llamar informalmente "bugs". El término es histórico, de tiempos previos a la invención del transistor.

2.1 IDE

Llegó la hora de escribir el primer programa. Y acá surge una importante pregunta: dónde lo tengo que escribir?? Existen muchas opciones de IDE (entorno de desarrollo integrado) para elegir. De hecho, no es necesario ni siquiera usar uno de estos, podría escribir su código en un archivo de texto, y guardarlo con la terminación .cpp, y luego usar algún compilador para generar el ejecutable. Pero las herramientas y amabilidad con el usuario que proveen los IDE resultan en una gran ventaja a la hora de aprender, y también de generar y mantener proyectos grandes, por lo cual le recomendamos fuertemente que elija y utilice uno. Si ya tiene y/o conoce uno, adelante! Si no posee le recomendamos Code::Blocks, que es multiplataforma y libre.

2.1.1 Code::Blocks

Una vez descargado e instalado, utilizar las herramientas más básicas de este IDE es relativamente simple. Lo guiaremos a continuación, sobre cómo empezar un proyecto.

Al abrir Code::Blocks, le aparecerá una pantalla como la de la **Fig.** 2.1. Seleccione Crear Nuevo Proyecto, y le aparecerá una ventana nueva, consultando qué tipo de proyecto desea (**Fig.** 2.2). Allí elija la opción de Aplicación de Consola y siga las instrucciones (elija C++ cuando le pidan el lenguaje, obviamente). A continuación elija el nombre del proyecto, y el

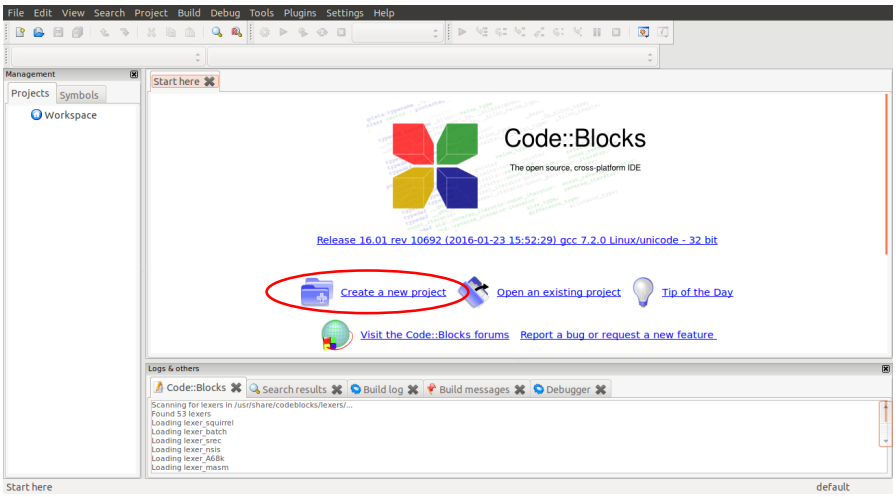

Fig. 2.1: Crear Nuevo Proyecto.

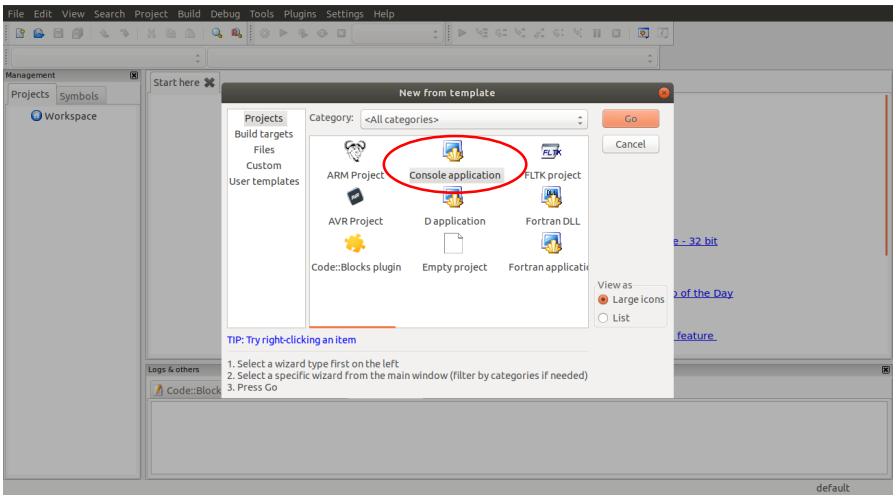

Fig. 2.2: Aplicación de consola.

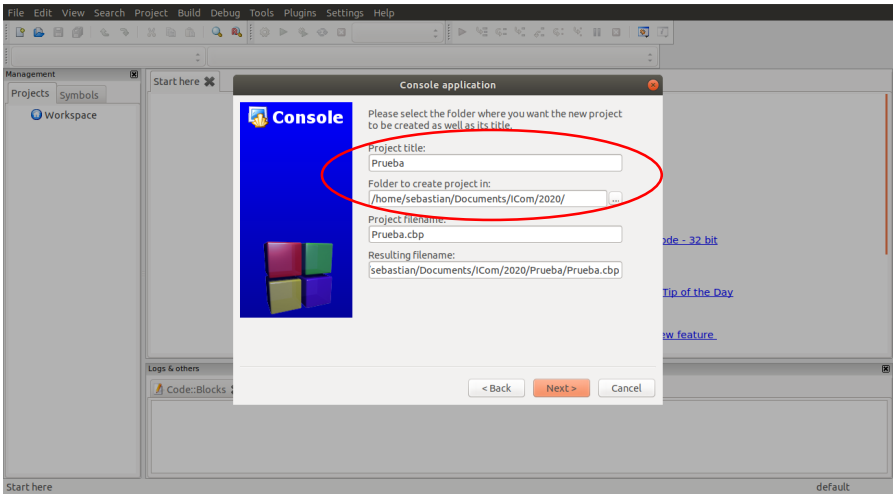

Fig. 2.3: Nombre y directorio del proyecto.

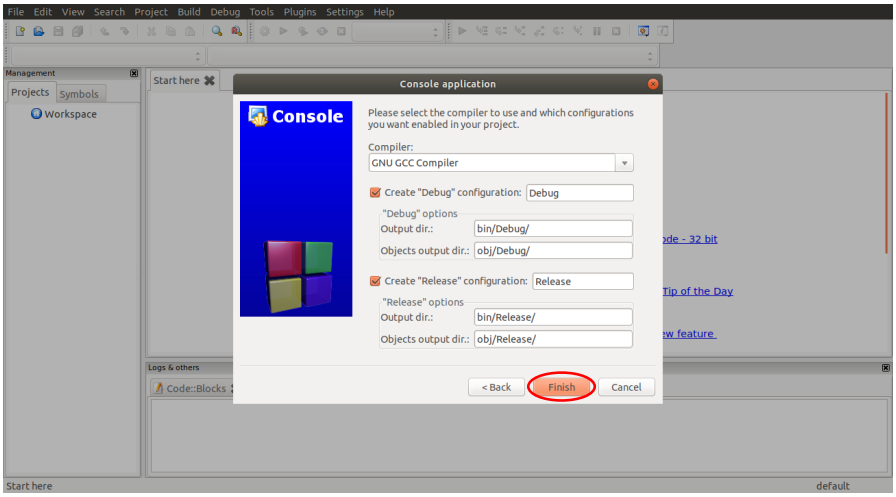

Fig. 2.4: Compilador.

directorio donde desea guardar los archivos relacionados (**Fig.** 2.3). Finalmente, seleccione el compilador que utilizará, y deje las otras opciones como aparecen en la imagen (**Fig.** 2.4). Al clickear Finalizar, el proyecto se creará, y podremos empezar a escribir nuestro programa! Por defecto, se creará un archivo llamado main.cpp, que contendrá el código necesario para el programa estándar de iniciación en la programación: el famoso "Hola mundo".

2.1.2 Hola mundo

Bueno, manos a la obra, escribamos un programa que haga algo muy simple: que escriba en la pantalla la frase "Hola mundo", y nada más. Cómo se haría esto en C++? Veamos:

```
#include <iostream>
int main()
{
    std::cout << "Hola mundo" << std::endl;
    return 0;
}
```

Sencillo, pero todo muy nuevo, qué es cada cosa!? A continuación va repetido lo mismo, pero con comentarios:

```
//En C++ se puede comentar una linea entera poniendo doble barra adelante.

/*La otra opcion para comentarios es encerrar varias lineas asi.
Esto tambien es comentario.
Y esto también, hasta aca*/

#include <iostream> /*Esto le dice al compilador que busque el archivo de
    encabezado (header) iostream, y la copie ahi.*/
```

```
int main() //este es el cuerpo del programa principal, que siempre debe
    llamarse así.
{
    std::cout << "Hola mundo" << std::endl;

    /*std:: este es un prefijo, que indica que cout y endl son parte de la
        biblioteca estándar de C++. Como se usa mucho std::, existe una
        forma de evitar la necesidad de escribirlo miles de veces, que
        veremos a continuación.*/

    // cout se encarga de mostrar en pantalla lo que le digamos con <<.
    // endl mete un salto de linea al final
    // los ; le dicen al compilador donde terminan las sentencias.
    // Toda sentencia termina con un ;

    return 0; /* Esto finaliza el programa y devuelve el valor 0, que por el
         momento no usaremos para nada, así que esta linea podría obviarse*/
} //las llaves marcan donde empiezan y terminan los bloques de codigo.
```

Y otra vez lo mismo, pero evitando usar std:: a cada rato:

```
#include <iostream>
using namespace std; /*Esto le indica al compilador que vamos a usar la
    biblioteca estandar, y por lo tanto no necesitamos agregar el prefijo
    std:: */
int main()
{
    cout << "Hola mundo" << endl;
    return 0;
}
```

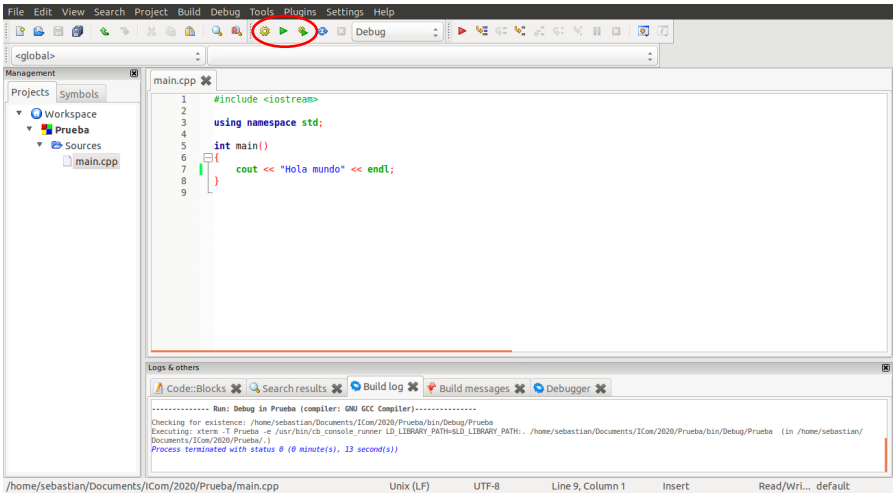

Fig. 2.5: Proyecto listo.

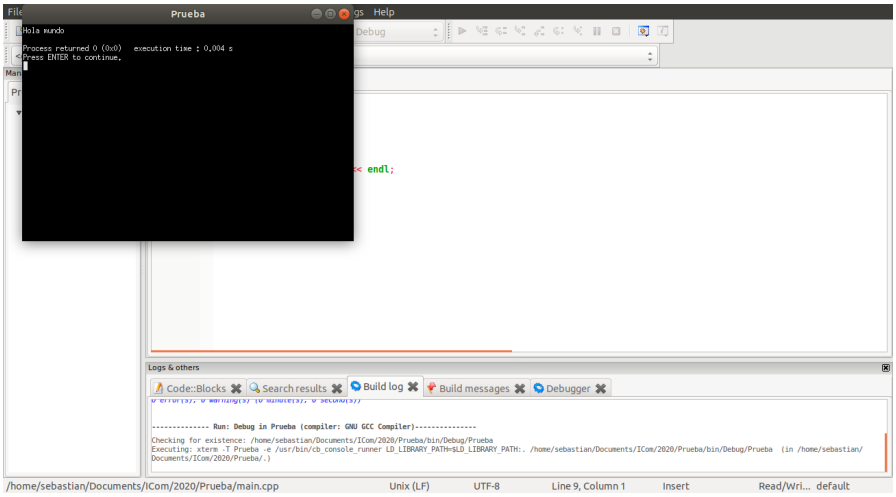

Fig. 2.6: Programa ejecutado.

Una vez que tenemos el código listo en Code::Blocks, se verá como en la **Fig.** 2.5. Sólo resta compilar y ejecutar. Para ello deberá clickear primero el engranaje amarillo (compilar) y una vez finalizada la compilación, el boton verde triangular (ejecutar). También puede utilizar la opción siguiente, representada por el engranaje y triangulo juntos. Esa opción compila y ejecuta en serie. Lo bueno: un sólo click. Lo malo: si hay algún problema de compilación, se ejecutará la última versión compilada exitosamente. Esto puede hacernos pensar que el programa está andando, cuando en realidad presenta errores, mucho cuidado al usar esta opción!

Una vez ejecutemos el programa, nos aparecerá una consola (**Fig.** 2.6, ventana negra) donde aparecerá lo que nuestro programa imprima por pantalla mediante cout, más alguna información extra y/o errores de ejecución si los hubiera.

Este simple programa se utiliza, por razones históricas, en la iniciación a cualquier lenguaje de programación. En el caso de C++, nos permite introducir varios conceptos importantes, que ya mencionamos, pero detallaremos a continuación. En primer lugar, el **main**:

```
int main()
{


}
```

Este bloque de código es el alma del programa. Cuando ejecutemos, lo que realmente hacemos es darle el control al **main** (bueno, a su versión compilada en realidad), y a partir de ahí empieza todo. Por eso es imprescindible que se llame así, y que no haya más de un **main** en el mismo programa! Adelante tiene el prefijo **int** separado por un espacio, que indica que al finalizar el bloque, **main** podrá devolver SOLO un valor de tipo **int** (entero). Las llaves marcan el comienzo ({) y final (}) del bloque de código. Esto no significa que siempre se ejecute todo lo que este entre las llaves, de hecho el bloque termina de ejecutarse por defecto al llegar a la llave final, pero lo mejor es terminarlo con la sentencia **return**.

```
int main()
```

```
{
    // Una sentencia aqui se ejecutaria

    return 0;

    /* Una sentencia aqui no, porque ya se finalizo el bloque mediante la
        palabra clave return*/
}
```

Allí donde pusimos un 0, usted puede poner cualquier valor en el rango que abarca el tipo **int** : -7 ,2 , 1000, etc. Más adelante veremos el significado que tiene ese valor retornado desde **main**. Justo en este caso no estamos utilizando el valor devuelto, así que mientras que cumpla con ser del tipo adecuado, está bien. Lo que no sería correcto, es devolver un valor de punto flotante, o de caracter, o cualquier otro, si ya declaramos el tipo de retorno como entero:

```
int main()
{
    return 0.5; //esto esta mal
    return 'y'; //esto tambien esta mal
    return "y esto esta muy mal";
}
```

Y los paréntesis al lado del **main**? Eso es para el uso de argumentos. Lo veremos en más detalle cuando hablemos de funciones, que comparten muchas características.

Ya hablamos de lo que es el **main**, lo que hace el **return** y lo que es el tipo de dato devuelto. Otra cosa fundamental es la Biblioteca Estándar, el famoso **std::**. Si bien es súper recomendable que si tiene tiempo investigue un poco al respecto, a los fines de esta materia, y de la generación de programas simples, sólo necesita saber lo siguiente: antes del **main**, escriba siempre **using namespace std;** Si hace eso, se podrá olvidar del tema, y sólo necesitará concentrarse en el problema que tenga en frente. Por lo tanto, la base absolutamente mínima que necesitará para empezar un programa de cero, será siempre así:

```
using namespace std;

int main()
{
    return 0;
}
```

Desde aquí podrá empezar a agregar todo lo que vaya necesitando. Sigamos con el ejemplo de Hola mundo, pero empezando con esto. Lo que queremos es utilizar alguna funcionalidad de C++ que nos permita imprimir texto en la consola. Allí nos toparemos con que lo que necesitamos es **cout**. Este elemento tiene un operador "especial" que permite que mandemos el texto a la

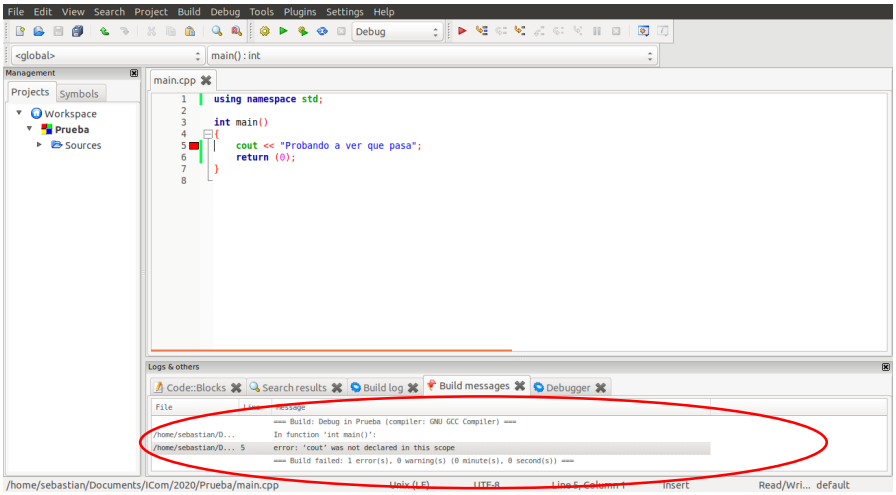

Fig. 2.7: Fallo en la compilación.

pantalla: <<. Listo, entonces si usamos eso debería estar completo el programa, no? No. Si utilizamos cout así sin más, nos va a surgir un error de compilación. El compilador nos dirá que no tiene idea de qué es **cout**. A ver, veamos entonces cómo se vería si intentamos compilar lo siguiente:

```
using namespace std;

int main()
{
    cout << "Probando a ver que pasa";
    return 0;
}
```

En la **Fig.** 2.7 se muestra lo que ocurre. Abajo de la pantalla hay un panel donde se muestra el resultado de la compilación. Allí se indica cuántos errores se encontraron, y se nos da pistas de cuál puede ser el problema. En este ejemplo, el problema es que **cout** está declarado en el header **iostream**, y si no lo incluimos en nuestro código, el compilador no sabe que existe. Y aquí entra algo muy importante: si va a usar cualquier funcionalidad propia de C++, no considere que ya esta incluida por defecto, fíjese qué headers necesita incluir. Y para agregarla, es muy sencillo, solo utilice la directiva **#include <>**, con el header adecuado entre los brackets. En nuestro caso:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Ahora si compila";
    return 0;
}
```

Note que las directivas al precompilador comienzan con el caracter **#** , en este caso la **#include** solicita de incluir el header y como todas las directivas no requiere de **;** al final.

Bueno, si entendió todo lo que se explicó hasta aquí, más allá de que se acuerde de memoria o no las palabras clave exactas, debería poder volver a ver el ejemplo de Hola mundo y contarle a alguien qué hace cada cosa.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hola mundo" << endl;
    return 0;
}
```

O tal vez aún le genere ruido ese segundo << y el **endl**. Utilizando el mismo comando **cout**, se puede mandar a pantalla en forma serial, todo lo que uno quiera, usando los << que sean necesarios. Y **endl** es un salto de línea, nada más. Entonces, podríamos lograr el mismo resultado con el siguiente programa:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hola" << " " << "mundo";
    cout << endl;
    return 0;
}
```

O también:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "H"<<"ola"
         <<" ";
    cout <<"mun"<<       "do";
    cout << endl;
    return 0;
}
```

Es decir, los espacios y saltos de linea entre << y lo que lo rodea no tienen ningún efecto sobre el resultado, sólo en la estética! Sólo cuando se encuentra un ; es que las sentencia termina.

Y hablando de estética, es muy importante que se tome el trabajo de hacerse el hábito de mantener prolijo y bien comentado su código. El uso de una única sentencia por línea y de

tabulaciones alineadas para que se vea rápidamente la estructura del programa, es fundamental. Pero mejor que explicar es mostrar. Fijándose cuál de las siguientes opciones entiende más fácilmente, se dará cuenta de lo que le decimos.

A)

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hola mundo" << endl;
    return 0;
}
```

B)

```
#include <iostream> using namespace std;int main(){cout<< "Hola mundo" <<
    endl;return 0;}
```

Sólo por las dudas, se espera que cuando entreguen, lo hagan como A...

3 Introducción a C++

A esta altura, ya debería tener andando su IDE de preferencia, y entender al menos la idea detrás de la representación numérica y simbólica en la computadora. Además, debería ser capaz de distinguir el bloque principal de un programa escrito en C++, y qué headers están incluidos. Suponiendo que todo esto es cierto, comencemos a aprender los distintos elementos que componen el lenguaje.

3.1 Los tipos de datos nativos

Como vimos en el capítulo anterior, los datos (numéricos o simbólicos) pueden ser almacenados mediante código binario (combinaciones de 1 y 0), mediante la utilización de sistemas de representación adecuados. La forma en que C++ nos permite seleccionar el tipo de representación, es el tipo de dato. Los tipos nativos en C++ se pueden separar en 4 grupos: caracteres, enteros, lógicos y de punto flotante.

3.1.0.1 Caracteres

Este grupo consta del tipo **char**. Un **char** ocupa 1 byte (8 bits) en la memoria, por lo que puede representar $2^8 = 256$ valores enteros. Estos se pueden relacionar a símbolos, o ser usados como meros números enteros de rango muy corto.

Existe un tipo ampliado **wchar_t**, que ocupa 2 bytes, pero en la práctica lo más probable es que por ahora no lo utilice. Este tipo de dato es útil en particular para idiomas con gran cantidad de símbolos, como el chino y el japonés, dónde no alcanza con 255 combinaciones para abarcar todos.

3.1.1 Lógicos

El tipo **bool** es el único miembro de este grupo, y puede tomar los valores **true** y **false**. Si bien se podría utilizar un único bit para representar esto, debido a que el byte, y no el bit, se utiliza como unidad mínima de memoria, el tipo **bool** ocupa 1 byte. Algo sumamente útil a tener en cuenta siempre que quiera utilizar variables u operadores lógicos: **false** $= 0$ y **true** $\neq 0$. Es decir, 0 es equivalente a **false**, mientras que cualquier valor entero distinto de 0 es equivalente a **true**. Téngalo presente siempre, ya que se utiliza muchísimo.

3.1.2 Enteros

Este es el grupo más grande, consta de 8 tipos distintos. Suena a mucho, pero en realidad todos se comportan igual, sólo cambia el rango (mínimo y máximo valor posible de representar).

Para entender bien esto, empecemos por el tipo más usado: **int**. Este tipo sirve para representar números enteros entre -2147483648 y 2147483647, ya que ocupa 4 bytes (32 bits). El resto de los 7 tipos, sólo cambian ese rango. Los tipos **short int**, **long int** y **long long int** son lo que su nombre indica, versiones de representación de enteros con un rango más corto, más largo, y mucho más largo, respectivamente. Bueno, ya mencionamos 4; y los otros 4? Esencialmente, agregando la palabra clave **unsigned** frente a estos tipos, se generan 4 más, que poseen un rango igual en valor absoluto, sólo que, como la palabra lo indica, sólo son números positivos o cero. Para el ejemplo de **int**, si usamos **unsigned int** el rango pasa a ser de 0 a 4294967295; es decir, siguen siendo 2^{32} valores posibles, sólo que elegimos representar sólo valores no negativos. En resumen, los tipos enteros son: **short int**, **int**, **long int**, **long long int**, **unsigned short int**, **unsigned int**, **unsigned long int** y **unsigned long long int**. La elección de uno frente a otro se basa únicamente en el rango que esperamos necesitar. Si bien podríamos utilizar siempre la versión más larga de entero, si no es realmente necesario para nuestro programa, estaremos desperdiciando memoria y tiempo de procesamiento innecesariamente.

Nota: En realidad, el ancho de cualquier tipo de enteros, o sea la cantidad de memoria asociada a una variable de cualquier tipo entero depende de la plataforma/compilador. Los valores expresados arriba son los que seguramente encontrará en la máquina que utilice.

3.1.3 Punto flotante

Ya vimos algo sobre representación de punto flotante en el capítulo anterior. La versión C++ de eso se refleja en los tipos posibles, los cuales, al igual que lo que vimos para enteros, sólo cambian el rango representable (que en este caso esta relacionado también con la precisión, cuidado!). Los tipos de este grupo son: **float**, **double** y **long double**. En general, uno utiliza el tipo **double** para casi todo lo que requiera variables de punto flotante. El tipo **float** se suele utilizar para casos donde la eficiencia sea realmente importante y se esté 100% seguro que alcanza con su precisión. Y el tipo **long double** para implementaciones donde no alcanza con la precisión de **double**.

3.2 La variable

Qué es una variable? Viniendo del ámbito de las Ciencias Exactas, todos saben lo que es una variable desde el punto de vista matemático. Sin embargo, en programación es ligeramente diferente. Ya no es sólo un símbolo que representa algún valor arbitrario. Tiene un lugar físico donde estará almacenado su *valor*, el espacio que ocupará estará determinado por su *tipo*, y a su vez esta porción de memoria posee una *dirección*. Es decir, cuando declaramos una variable en C++ lo que hacemos es reservar un espacio de memoria de tamaño adecuando para el tipo de la variable, en algún lugar de la memoria. De esa forma, podemos almacenar distintos valores allí, siempre y cuando sean del tipo adecuado. La variable entonces, además de almacenar el valor

que deseamos, nos provee acceso a esa información. En particular, para acceder a la dirección de memoria de una variable existe el operador **&**. Para entender cómo se utiliza este operador, lo más sencillo es mostrar un ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    int x; //Esta es la declaracion de una variable de tipo int

    x = 10; //Aqui inicializamos x con un valor de 10

    cout << "Este es el valor inicial de la variable: " << x << endl;
    cout << "Y esta es su direccion de memoria: " << &x << endl;

    x = 20; //Le damos un valor distinto a la variable

    cout << "Este es el valor nuevo de la variable: " << x << endl;
    cout << "Su direccion de memoria sigue siendo la misma: " << &x << endl;

    return 0;
}
```

Compile y ejecute este ejemplo para comprobar el funcionamiento. Si bien el valor de la dirección de memoria no tiene un significado obvio desde la perspectiva humana, la máquina sabe exactamente dónde se encuentra físicamente esa dirección. Por lo tanto, es una herramienta sumamente útil para manipular variables y datos en general. Pero no se preocupe, todo a su debido tiempo.

En lenguajes de programación orientados a objetos, como puede ser C++ si así se elije, el término *variable* es reemplazado por el *objeto*. Pero es lo mismo: un pedazo de memoria reservada, del tamaño adecuado para el tipo de dato, y en alguna dirección de memoria. Lo que cambia cuando se programa con orientación a objetos lo veremos en la cursada.

3.3 Operadores básicos

Los operadores en programación son similares a los operadores matemáticos, algunos de hecho iguales. Cada operador requiere de un número de argumentos para operar, pudiendo devolver un resultado o no. Veamos algunos ejemplos simples:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int x, y; //Defino dos variables, sin inicializar

    int w=2, z=5; //Defino dos variables inicializadas

    x = w + z;
    /*El operador + requiere de dos argumentos. Lo que hace es sumar esos
        argumentos, nada mas. Si el resultado no se utiliza ahi mismo, se
        pierde, ya que no tiene asignado un espacio de memoria. El operador
        = también necesita dos argumentos: lo que hace es copiar el VALOR
        del argumento de la derecha, en la DIRECCION DE MEMORIA del
        argumento de la izquierda.*/

    y = w - z;
    /*El operador - es muy similar al + .*/

    y = w * z;
    /*Si se utiliza con dos argumentos, lo que hace es multiplicar esos
        argumentos. Si el resultado no se utiliza ahi mismo, se pierde, ya
        que no tiene asignado un espacio de memoria. Pero el operador = se
        encarga de guardar ese valor en el espacio reservado para la
        variable y.*/

    y = z / w;
    /*Como ambos argumentos son de tipo entero, la division devolvera un
        entero tambien, sin decimales (en este caso devolvera 2). Si el
        resultado no se utiliza ahi mismo, se pierde, ya que no tiene
        asignado un espacio de memoria. Pero el operador = se encarga de
        guardar ese valor en el espacio reservado para la variable y.*/

    return 0;
}
```

Piense un minuto qué significa declarar una variable, si podemos hacerlo sin darle un valor. Qué estamos declarando? Si entendió los fundamentos explicados con anterioridad, debería ver que lo que declaramos es el espacio de memoria que vamos a utilizar. El símbolo que le damos, en este caso `x` e `y`, nos sirven para poder acceder allí más adelante, y guardar los valores que queramos. Si ya teníamos un valor guardado, como es el caso de `w` y `z`, también nos permite leer ese valor. La dirección de memoria que se reserva inicialmente es aleatoria (la R de RAM, *Random Access Memory*), pero una vez definida es constante hasta que la variable se destruya, por ejemplo al finalizar el programa.

3.3.1 Operadores de asignación especiales

Existen algunos operadores que cumplen la función de asignar un nuevo valor a una variable, ahorrando un paso. Una vez que comience a programar, va a ver que son muy útiles y cómodos. Pero para utilizarlos correctamente, debe entender qué es lo que hacen. Veamos un ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    int x=1, y=2;

    x = x + y; /*Esta tipo de operacion es muy comun, por lo que existe una
        forma mas compacta de hacer o mismo.*/

    x += y; //Esto es equivalente a lo anterior.

    return 0;
}
```

El operador **+=** toma el valor de la variable de la izquierda (x), le suma el valor de la variable de la derecha (y), y finalmente guarda el resultado en la dirección de memoria de la variable de la izquierda (x). Así como está este operador de "autosuma", existen equivalentes para el resto de los operadores algebraicos:

```
#include <iostream>
using namespace std;

int main()
{
    int x=1, y=2;

    x *= y;
    y /= x;
    x -= y;

    return 0;
}
```

Ahora bien, este tipo de operadores sólo son válidos cuando las variables ya poseen un valor guardado.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int x, y=2, z=0;

    x *= y; //Esto esta mal.
    y /= x; //Esto tambien.
    x -= y; //Y esto tambien.

    z += y; /*Esto esta bien, ya que ambas variables poseen un valor, es
        decir, estan inicializadas.*/

    y -= 2; //Esto tambien esta bien.

    2 += y; //Pero esto esta muy mal!

    return 0;
}
```

En los últimos dos casos de este ejemplo, se presenta una importante asimetría. Si ambos lados tienen un valor determinado, por qué en el primer caso está bien (y -= 2), mientras que en el segundo esta mal (2 += y)? Simple, como dijimos, estos operadores terminan guardando el resultado en la dirección de memoria del miembro de la izquierda. Pero los literales (valores sueltos de cualquier tipo, como pueden ser: 2, 15.1, 'a', true, "Hola mundo", etc.) no tienen asignada una dirección de memoria, sólo son un valor suelto. Por lo tanto, no hay donde guardar el resultado, ergo -> error.

Otros (auto)operadores son aún más simples, sólo requieren de una variable. Esencialmente, son 4, aunque mayormente usará 2. Veamos en un ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    int i=0, j=3;

    i++;   //Le sumo 1 a i
    ++i;   //Le sumo 1 a i
    j--;   //Le resto 1 a j
    --j;   //Le resto 1 a j

    return 0;
}
```

Pero si parece que hacen lo mismo, no deberían ser 2 operadores solamente? No. Tanto si se ponen los $++$ ($--$) al frente o detrás de la variable, sumarán (restarán) 1. Sin embargo, hay un detalle que los diferencia: el tiempo al cual se realiza la operación. Este tipo de operadores, cuando están sueltos así efectivamente dan igual, pero cuando son utilizados en expresiones, su comportamiento distinto se nota. Básicamente, si el operador está al frente, la operación tiene prioridad a las comparaciones o asignaciones, mientras que si está detrás, es al revés. Igualmente, no se preocupe, es algo muy sutil. En general, para la gran mayoría de los casos con los que se encontrará, con utilizar sólo las versiones i$++$ e i$--$ será suficiente.

3.3.2 Operadores relacionales

Otros operadores utilizados frecuentemente son los relacionales. Estos operadores comprueban la verdad o falsedad de una afirmación. Por lo tanto, devuelven un valor de tipo **bool**. Veamos cómo se pueden usas estos operadores en un ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    bool q;
    int i=1, j=2;

    q = (j > i);  //Es j mayor a i? SI -> q = true

    q = (j >= i); //Es j mayor o igual a i? SI -> q = true

    q = (j <= i); //Es j menor o igual a i? NO -> q = false

    q = (j < i);  //Es j menor a i? NO -> q = false

    q = (j == i); //Es j igual a i? NO -> q = false

    q = (j != i); //Es j distinto a i? SI -> q = true

    return 0;
}
```

En este ejemplo fuimos sobrescribiendo los valores resultantes en la variable q, aunque eso no es obligatorio. De hecho estos operadores se suelen utilizar para comprobar alguna condición en un momento determinado, pero el valor de retorno no se guarda a menos que sea necesario más adelante. El error más común que puede encontrarse al utilizar estos operadores es confundirse el operador asignación (=) con el operador de igualdad (==).

3.3.3 Operadores lógicos

Los operadores lógicos están estrechamente relacionados a los operadores que vimos en la sección anterior. Básicamente constan de 3: `&&` (and), `||` (or) y `!` (not). Y como puede esperarse, su valor de retorno es de tipo **bool**. Veamos un ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    bool q, x=true, y=false;

    q = (x && y);  // q = false

    q = (x || y);  // q = true

    q = (x && !y);  // q = true

    q = (!x && y);  // q = false

    q = (x || !y);  // q = true

    q = (!x || !y);  // q = true

    q = !(x || !y);  // q = false

    return 0;
}
```

Esto debería ser fácil de comprender una vez que se familiarice con los símbolos que representan las operaciones, sino repase sus apuntes de álgebra.

3.3.4 Operadores referenciales

Estos operadores son sólo 2, y a uno ya lo introdujimos antes: `&`. Su función es operar sobre el bloque de memoria que ocupa la variable. En el caso que ya vimos de `&`, aplicado sobre una variable nos retorna la dirección de memoria donde se encuentra almacenado el valor correspondiente a la misma.

El otro operador es `*`, que utiliza el mismo símbolo que la operación de multiplicación, pero hace algo muy distinto. El operador referencial `*` aplicado sobre UNA DIRECCIÓN DE MEMORIA, devuelve el valor que esté almacenado allí. Esto puede sonar extraño al principio, pero si lo piensa, es una herramienta muy útil. Y para poder sacarle el mayor provecho, el

lenguaje nos facilita además un tipo de dato especial: el puntero. No se deje confundir por su nombre, es sólo para indicar que este tipo de variable en realidad "apunta" hacia otra variable. En realidad lo que hace es guardar la dirección de memoria de esa otra variable, por lo que tendremos acceso a ella desde el puntero también; de ahí el nombre. Pero esto quedará mucho más claro en un ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    int x=5, z=2;  //variable de tipo int

    int *y;    //puntero de tipo int

    y = &x;    //guardo la direccion de memoria donde almacene el valor 5

    cout << "Ahora puedo acceder a x como siempre: " << x << endl;

    cout << "Pero tambien desde el puntero: " << *y << endl;

    //No solo puedo leer el valor guardado en x, desde y, sino tambien
        modificarlo!

    *y = 0;

    cout << "Cual es el valor de x ahora?: " << x << endl;

    //Y ademas puedo usar el mismo puntero para guardar otra direccion,
        sobrescribiendo la anterior:

    y = &z;

    cout << "Cual es el valor al que esta apuntando y ahora?: " << *y <<
        endl;

    return 0;
}
```

El puntero es una herramienta heredada de C, y si bien es sumamente útil, también es muy peligrosa. No peligrosa en el sentido que puede dañar algo, sino que es MUY fácil cometer errores y muy difícil de encontrarlos luego! Por esa razón, sólo deben ser utilizados en casos excepcionales, donde las herramientas propias de C++ no alcancen. Igualmente, esto no significa

que no los vayamos a utilizar en la materia, lo haremos; de hecho, le recomendamos que se haga amigo de los punteros lo antes posible. Y qué mejor forma que viendo cuánto se pueden complicar, si uno así lo desea. Veamos un ejemplo similar al anterior, pero con una vuelta de tuerca más:

```
#include <iostream>
using namespace std;

int main()
{
    int x=5;   //variable de tipo int

    int *y;    //puntero de tipo int

    int **z;   //puntero de tipo int*

    y = &x;    //guardo la direccion de memoria donde almacene el valor 5

    z = &y;    /*guardo en z la direccion de memoria donde almacene la
        direccion de memoria donde almacene el valor 5. Esto puede sonar
        confuso, pero siga leyendo y ya vera que sentido tiene esto.*/

    cout << "Ahora puedo acceder a x como siempre: " << x << endl;

    cout << "Pero tambien desde el puntero: " << *y << endl;

    cout << "Y tambien desde z: " << *(*z) << endl; /*Si entendio esto a la
        primera, felicitaciones, sino no se preocupe que es normal.*/

    return 0;
}
```

Esto de guardar en una variable de tipo puntero la dirección de memoria de otra variable puede anidarse hasta el infinito, aunque no es recomendable. Pero por qué aplicarle dos asteriscos a `z` nos devuelve el valor de `x`? Vamos de a una cosa: en `z` tenemos guardada la dirección de `y`. Asi que cuando hacemos `*z`, lo que nos devuelve es el valor guardado en `y`. Es decir `*z == y`. Ese valor justamente es otra dirección de memoria, así que al aplicarle otra vez el `*`, nos devolverá el valor guardado allí, que en este caso es 5, el valor de `x`. En resumen: `*(*z) == *(y) == x`.

3.3.5 Operadores de bits

Ya vimos que los tipos de datos son representaciones binarias, es decir, un tren de unos y ceros de un cierto tamaño. Para operar directamente sobre esos valores binarios, es que existen

este tipo de operadores. Por ahora sólo introduciremos 2: << y >>. Ya a esta altura debe haberse resignado a que los símbolos son reutilizados una y otra vez para distintas operaciones. El compilador se encarga de elegir la interpretación correcta, en base al/los argumento/s utilizados.

El operador << mueve todos los valores binarios 1 lugar hacia la izquierda (metiendo un 0 por la derecha), mientras que el >> lo hace hacia la derecha (metiendo un 0 por la izquierda). Existen bastantes operadores más, en los cuales no ahondaremos por ahora. Por ahora sólo tenga presente que existen, por si alguna vez los necesita.

4 Control de flujo del programa

El flujo de un programa es, esencialmente, secuencial. El **main** toma el control al momento de la ejecución, y las sentencias contenidas allí se van ejecutando una atrás de la otra, en el orden en que están escritas. Sin embargo, este flujo se puede alterar mediante el llamado a funciones, o el uso de bloques condicionales, o de bloques repetitivos.

4.1 Condicionales

4.1.1 if-else

Este es el tipo más simple de condicional. Consta de las palabras clave **if** y **else**. Veamos cómo funcionan con un ejemplo simple:

```
#include <iostream>
using namespace std;

int main()
{
    short int x=5, y=7;

    if (x>y)
    {
        cout << "Esto no se va a imprimir en pantalla";
    }
    else
    {
        cout << "Pero esto si" << endl;
    }

    return 0;
}
```

Es muy simple, no? Esencialmente, si el argumento del **if** () es verdadero, se ejecuta el bloque que le sigue, sino se ejecuta el bloque del **else**. Veamos otro ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    bool q=true;
```

```
    short int x=5, y=7;

    if (x>y)
    {
        cout << "Esto no se va a imprimir en pantalla";
    }
    else if (q)
    {
        cout << "Pero esto si" << endl;
    }
    else
        cout << "Y esto no"

    return 0;
}
```

Ahora si no se cumple la primera condición, nos fijamos en otra condición distinta, y sólo si esa tampoco se cumple, se ejecutará el tercer bloque. En este caso, la segunda condición se cumple, así que la ultima sentencia, definida por el **else**, nunca se ejecuta. Y otra cosa que introdujimos: los bloques de código que sólo constan de una única sentencia no necesitan llaves, pueden obviarse. El siguiente ejemplo es idéntico al anterior:

```
#include <iostream>
using namespace std;

int main()
{
    bool q=true;
    short int x=5, y=7;

    if (x>y)
        cout << "Esto no se va a imprimir en pantalla";
    else if (q)
        cout << "Pero esto si" << endl;
    else
        cout << "Y esto no"

    return 0;
}
```

Veamos un ejemplo diferente, donde el flujo del programa se pueda modificar dependiendo de algún valor ingresado por el usuario:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int x=5, y;

    cout << "Ingrese un valor entero: ";

    cin >> y;

    if (y>x)
        cout <<"El valor ingresado es mayor a " << x << endl;
    else
    {
        cout << "El valor ingresado es menor o igual a " << x << endl;

        return 0;
    }

    cout << "Esto solo se mostrara si el valor ingresado fue mayor a " << x
        << endl;

    return 0;
}
```

Aquí introdujimos el comando opuesto a **cout** -> **cin**. El funcionamiento es similar, pero en dirección contraria (fijese además que el operador utilizado es >> en vez de <<). **std::cin** toma valores ingresados desde el teclado, y los guarda en las variables a la derecha del operador >>. Lo usaremos frecuentemente, así que se acostumbrará, así como ya se acostumbró al **cout**. Por otro lado, ahora nuestro código tiene 2 sentencias de **return** en distintos lugares. Si corre este programa, verá que dependiendo de qué valor y ingrese, si el programa finaliza mediante una u otra. Veamos otro ejemplo más:

```
#include <iostream>
using namespace std;

int main()
{
    int x, y;

    cout << "Ingrese dos valores enteros, separados por espacio o salto de
        linea: ";

    cin >> y >> x;
```

```
    if(y>x)
        cout << y <<" es mayor a " << x << endl;
    else
    {
        cout << y << " es menor o igual a " << x << endl;

        return 0;
    }

    cout << "Esto solo se mostrara si " << y << " es mayor a " << x << endl;

    return 0;
}
```

Aquí se ve cómo **std::cin** puede ser utilizado para ingresar tantos valores como se desee.

Una cosa a tener en cuenta, es que los bloques **else** no son obligatorios. Se puede usar el condicional **if()** solo:

```
#include <iostream>
using namespace std;

int main()
{
    int x;

    cout << "Ingrese un valor entero";

    cin >> x;

    if (x<0)
    {
        cout << x << " es menor a " << 0 << endl;
        return 0;
    }

    cout << "Esto solo se mostrara si " << x << " es mayor o igual a " << 0
        << endl;

    return 0;
}
```

4.1.2 switch-case

Existen casos donde se desea comprobar múltiples condiciones, como por ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    int x;

    cout << "Ingrese un valor entero";

    cin >> x;

    if (x==0)
        cout << x << " es igual a " << 0 << endl;
    else if (x==1)
        cout << x << " es igual a " << 1 << endl;
    else if (x==2)
        cout << x << " es igual a " << 2 << endl;
    else if (x==3)
        cout << x << " es igual a " << 3 << endl;
    else if (x==4)
        cout << x << " es igual a " << 4 << endl;
    else
        cout << "Es otro" << endl;

    return 0;
}
```

Y si bien esto no está mal, existe otra forma más prolija de hacer lo mismo:

```
#include <iostream>
using namespace std;

int main()
{
    int x;

    cout << "Ingrese un valor entero";

    cin >> x;

    switch (x)
```

```
    {
        case 0:
           cout << x << " es igual a " << 0 << endl;
           break;

        case 1:
            cout << x << " es igual a " << 1 << endl;
            break;

        case 2:
            cout << x << " es igual a " << 2 << endl;
            break;

        case 3:
            cout << x << " es igual a " << 3 << endl;
            break;

        case 4:
            cout << x << " es igual a " << 4 << endl;
            break;

        default:
            cout << "Es otro" << endl;
    }

    return 0;
}
```

Básicamente, la expresión (en este caso `x`) que se pasa al **switch** se compara con los distintos **case**, y si se cumple alguno, se ejecutan todas las sentencias que sigan. Dentro de cada **case** no es necesario utilizar llaves, incluso cuando hay más de una sentencia. Esto se debe a que todo el **switch** es un bloque de código. De hecho, es la razón por la cual debemos agregar la sentencia **break** al final de cada **case**. Este comando finaliza el bloque de código actual, y devuelve el control al que este por encima; en este caso el **main**. Si no agregamos estos **break**, una vez que se cumpla con algún **case**, todas las sentencias que sigan, incluidas las propias de los **case** que le sigan, van a ser ejecutadas! El comando **break** no es exclusivo del **switch**; se puede utilizar también dentro de cualquier estructura de repetición.

4.2 Repetidores

Existen 3 formas de repetir sentencias en C++. Todas comparten la misma lógica: repetir un bloque de código hasta que se cumpla una condición preestablecida. A continuación las

detallaremos.

4.2.1 for

Esta es probablemente la forma más frecuente de iterar un bloque de código. Empecemos con lo más simple: una iteración infinita súper básica.

```
#include <iostream>
using namespace std;

int main()
{
    for (;;)
    {
        cout << "Esto se imprimira en pantalla una y otra vez, infinitamente
            ..." << endl;
    }

    return 0;
}
```

Fíjese la estructura anterior; consta de un bloque de código, con el **for(;;)** adelante. Seguramente se estará preguntando para qué son los **;;** adentro de los paréntesis. Esto es debido a que la instrucción **for** requiere de 3 cosas, separadas por **;** y en este caso las 3 están vacías. Veamos un ejemplo donde no lo estén:

```
#include <iostream>
using namespace std;

int main()
{
    for (int i=0 ; i<10 ; i++)
    {
        cout << "Esto se imprimira en pantalla 10 veces: " << i+1 << " de "
            << 10 << endl;
    }

    return 0;
}
```

Veamos entonces para qué sirve cada parte. La primera (`int i=0`) es una sentencia que se ejecuta antes de empezar a iterar; sea lo que sea que escriban allí, se ejecutará una única vez, al comienzo. La segunda (`i<10`) define la condición que debe cumplirse para que el bloque de código se ejecute. Se chequea al comienzo de cada ciclo, y mientras no sea **false**, se ejecutará

una y otra vez el bloque; por eso cuando estaba vacía se repetía infinitamente. La última es una sentencia que se ejecuta al final de cada ciclo. En este caso, le sumamos 1 a la variable `i`. Veamos otro ejemplo, aplicando de paso la instrucción **break**:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Lo que sigue se imprimiria en pantalla una y otra vez,
        infinitamente, si no fuera por el break..." << endl << endl;

    int i=0;

    for (;;)
    {
        cout << "Esto se imprimira en pantalla 10 veces: " << i+1 << " de "
            << 10 << endl;

        i++;

        if (i>=10)
            break;
    }

    return 0;
}
```

Este ejemplo hace lo mismo que el anterior, sólo que en vez de aprovechar los lugares que el **for** tiene reservados, lo hacemos "a mano". Obviamente, esto no tiene mucho sentido, ya que requiere de más código, y es más propenso a equivocaciones. Pero aprovechemos este ejemplo para mostrar dónde se ejecuta cada parte del **for**:

```
#include <iostream>
using namespace std;

int main()
{
    int i=0;

    cout << "Esto esta antes del for" << endl;

    for (cout << "Esto esta en el primer argumento del for" << endl << endl
        ; true ; cout << "Esto esta en el tercer argumento del for" << endl)
```

```
    {
        cout << "Esto se imprimira en pantalla 10 veces: " << i+1 << " de "
            << 10 << endl;

        i++;

        if (i>=10)
        {
            cout << "Si entramos aquí, se termina el for mediante el break"
                << endl;
            break;
        }
    }
    cout << "Esto esta despues de finalizar el bloque del for" << endl <<
        endl;

    return 0;
}
```

Ejecute este programa para ver el resultado; comparar lo que se imprime en la consola con el código le ayudará a entender mejor el funcionamiento del **for**.

4.2.2 while

Este repetidor es más simple que el **for**, en el sentido que sólo requiere de un argumento, equivalente al segundo argumento del **for**: la condición que debe cumplirse para que se ejecute el bloque, y que se chequea al comienzo de cada iteración. Veamos un ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    int i=0;

    while (i<10)
    {
        cout << "Esto se imprimira en pantalla 10 veces: " << i+1 << " de "
            << 10 << endl;
        i++;
    }

    return 0;
}
```

Una diferencia importante respecto del **for**, es que el argumento del **while** no puede dejarse vacío, si lo hacen el compilador les marcará un error. Si se desea hacer un ciclo infinito, se puede hacer así:

```
#include <iostream>
using namespace std;

int main()
{
    while (true)
    {
        cout << "Esto se imprimira en pantalla una y otra vez, infinitamente
            ..." << endl;
    }

    return 0;
}
```

Y si se quiere romper un ciclo infinito, se puede usar **break**:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Lo que sigue se imprimiria en pantalla una y otra vez,
        infinitamente, si no fuera por el break..." << endl << endl;

    int i=0;

    while (true)
    {
        cout << "Esto se imprimira en pantalla 10 veces: " << i+1 << " de "
            << 10 << endl;

        i++;

        if (i>=10)
            break;
    }

    return 0;
}
```

Ya ve que la lógica es muy similar, y sólo cambian algunos detalles. De hecho uno podría hacer

lo mismo usando **while** o **for**. Y por qué existen ambos entonces? Porque dependiendo lo que se quiera hacer, uno o el otro es más adecuado. Como regla simple, si no va a utilizar al menos 2 de los 3 argumentos del **for**, mejor use **while**.

4.2.3 do-while

El comportamiento de esta estructura de control es idéntico al anterior, con una única diferencia: la condición se chequea al final, en vez de al principio del ciclo. Veamos un ejemplo que compare ambas:

```
#include <iostream>
using namespace std;

int main()
{
    while (false)
    {
        cout << "Esto jamas se imprimira en pantalla, ya que la condicion es
             falsa, y se chequea antes de llegar aca." << endl;
    }

    do
    {
        cout << "Esto se imprimira una sola vez en pantalla, ya que la
            condicion es falsa, pero se chequea al final del ciclo, por lo
            que pasamos por aca una vez." << endl;
    } while (false);

    return 0;
}
```

Fíjese que en el segundo caso es necesario poner un **;** luego del **while()**. Si no lo cumple, el compilador le marcará error.

4.3 Funciones

Las funciones son una de las bases fundamentales de la división de tareas en programación. Ayudan también a la reutilización de código, ya que una misma función puede ser de utilidad para varios programas distintos. Veamos las partes de una función, con un ejemplo:

```
#include <iostream>
using namespace std;

int sumar(int,int); //declaracion de la funcion
```

```
int main()
{
    int x=3, y=5, z;

    z = sumar(x, y); //uso la funcion con los argumentos x,y

    cout << x << " + " << y << " = " << z << endl;
    return 0;
}

int sumar(int a, int b) //definicion de la funcion
{
    return a+b;
}
```

La declaración de la función define el tipo de retorno (en este caso **int**), el nombre (en este caso "sumar") y el número y tipo de los argumentos (en este caso: 2 argumentos, ambos tipo **int**). La declaración debe hacerse antes de poder utilizar la función. Por lo tanto, antes del **main** en este caso. La definición de la función, que en este caso se puso después del **main** es donde explicitamos qué es lo que hace exactamente. En el caso en que se defina la función antes de su primer uso (en este caso sería antes del **main**), no es necesario declararla. Veamos con un ejemplo:

```
#include <iostream>
using namespace std;

int sumar(int a, int b) //definicion de la funcion
{
    return a+b;
}

int main()
{
    int x=3, y=5, z;

    z = sumar(x, y);

    cout << x << " + " << y << " = " << z << endl;
    return 0;
}
```

Si bien puede parecer más cómodo hacerlo así, no es una buena práctica. Por qué? Cuando un programa tiene muchas funciones, al declararlas así el **main** queda al final del archivo, por un

lado, lo que no es muy cómodo. Y si necesitara revisar qué tipo de retorno y/o qué argumentos requiere cada función, deberá buscar entre medio de miles de lineas de código. En cambio, si las declaraciones están separadas de las definiciones, es mucho más fácil revisar y comparar las funciones entre sí.

Algo importante que debe entender para utilizar funciones correctamente es cómo se pasan los argumentos. Qué ocurre cuando llamamos a la función `sumar(x,y)`? Básicamente, dentro de la función las variables locales a,b son creadas e inicializadas con los valores de los argumentos pasados. En este ejemplo, se copian los VALORES de `x,y` en `a,b`. Las variables locales `a` y `b` poseen su propia dirección de memoria, así que una vez que se copiaron los valores de `x` e `y` allí, pasan a ser absolutamente independientes. Al finalizar la función, todas las variables locales son destruidas. Esto agrega seguridad al código, ya que impide, por ejemplo, modificar involuntariamente los valores de las variables pasadas como argumento. Veamos en un ejemplo:

```
#include <iostream>
using namespace std;

void probar(int);

int main()
{
    int x=3;

    cout << "valor inicial de x = " << x << endl;
    cout << "direccion inicial de x = " << &x << endl <<endl;

    probar(x);

    cout << "valor final de x = " << x << endl;
    cout << "direccion final de x = " << &x << endl;

    return 0;
}

void probar(int a)
{
    cout << "valor inicial de a = " << a << endl;
    cout << "direccion inicial de a = " << &a << endl;

    a=10;

    cout << "valor final de a = " << a << endl;
    cout << "direccion final de a = " << &a << endl <<endl;
```

```
    return;
}
```

Note que el tipo de retorno de la función se puso como **void**. Este es un tipo especial; ponerle tipo de retorno **void** a una función significa que NO DEVOLVERÁ NINGÚN VALOR. Por eso el **return** no tiene argumento.

Ya vimos que el pasaje de argumentos por copia no permite modificar las variables originales. Y si queremos poder modificar el valor de x desde la función? Existe una forma simple para hacerlo, veamos cómo funciona primero en un ejemplo:

```
#include <iostream>
using namespace std;

void probar(int &); //note el cambio aqui

int main()
{
    int x=3;

    cout << "valor inicial de x = " << x << endl;
    cout << "direccion inicial de x = " << &x << endl <<endl;

    probar(x);

    cout << "valor final de x = " << x << endl;
    cout << "direccion final de x = " << &x << endl;

    return 0;
}

void probar(int &a) //y tambien aqui
{
    cout << "valor inicial de a = " << a << endl;
    cout << "direccion inicial de a = " << &a << endl;

    a=10;

    cout << "valor final de a = " << a << endl;
    cout << "direccion final de a = " << &a << endl <<endl;

    return;
}
```

Note que lo único que cambiamos en el código es que agregamos un `&` en el tipo del argumento, tanto en la declaración como en la definición de la función. Es decir, el tipo cambio de **int** a **int** `&`. Y esto por qué cambió tan radicalmente el comportamiento? Porque al definirlo así, estamos indicando que no queremos crear una nueva variable y copiar un valor allí, sino que queremos utilizar directamente la variable `x`. Y para hacer esto necesitamos acceso a la dirección de memoria de `x`! Al poner ese `&` forzamos a que la variable local `a` comparta la misma dirección de memoria que `x`, por eso cuando modificamos el valor de `a`, también se modifica el de `x`, en realidad `a` y `x` son lo mismo, sólo que con distinto nombre. Cuando el paso de argumentos utiliza esta forma, se dice que los argumentos se pasan por referencia. Es decir, `a` es una referencia a `x`.

Esto de pasar argumentos por referencia es MUY útil, debido a que tiene una gran ventaja: imagine que quiere pasar una variable de gran tamaño a una función. Si la pasamos directamente como en el primer ejemplo, estaremos copiando una gran cantidad de bytes a otro lugar en la memoria. Esto consume tiempo de procesamiento y memoria. En cambio, si pasamos el argumento por referencia, no copiamos nada! Sólo pasamos la dirección de memoria de la variable; y las direcciones de memoria poseen un tamaño fijo. Así que pasando argumentos por referencia tiene el potencial de ahorrar muchos recursos. Sin embargo, debido a lo que discutimos previamente, el pasaje de argumentos por referencia pone en riesgo de ser modificadas a las variables originales! Si uno tiene cuidado, no va a haber problemas, pero el riesgo existe. Si este riesgo no es aceptable, existe una forma de ayudar al programador a detectar si incurrió en esto sin querer. Veamos cómo debemos cambiar el código del último ejemplo para asegurar que no nos equivoquemos:

```
#include <iostream>
using namespace std;

void probar(const int &); //note el cambio aqui

int main()
{
    int x=3;

    cout << "valor inicial de x = " << x << endl;
    cout << "direccion inicial de x = " << &x << endl <<endl;

    probar(x);

    cout << "valor final de x = " << x << endl;
    cout << "direccion final de x = " << &x << endl;

    return 0;
}
```

```
void probar(const int &a) //y tambien aqui
{
    cout << "valor inicial de a = " << a << endl;
    cout << "direccion inicial de a = " << &a << endl;

    a=10;

    cout << "valor final de a = " << a << endl;
    cout << "direccion final de a = " << &a << endl <<endl;

    return;
}
```

Fíjese que sólo agregamos el comando **const** frente al tipo del argumento, tanto en la declaración como en la definición de la función. Esto le dice al compilador que chequee si intentamos modificar en algún momento la variable. Si esto ocurre, nos indicará un error de compilación por intentar modificar una referencia de sólo lectura. Intente compilar este último ejemplo y lo verá. Si en ningún momento se intenta modificar la variable, el programa compilado resultante será indiferente a si pusimos o no esos **const**. Este comando es uno de tantos que sirven para ayudar al programador a no cometer errores. Puede parecer ridículo ponerse este tipo de limitaciones para no equivocarse; pero en proyectos grandes, especialmente si en el mismo participan varias personas, este tipo de precauciones pueden ser la diferencia entre el éxito y el fracaso.

5 Notas finales

Si llegó hasta aquí leyendo todo, es un buen comienzo. PERO, si sólo leyó el material y no se tomó el trabajo de probar cada ejemplo (aunque haya entendido todo), se lo olvidará en menos de 2 semanas. Es más, en un par de días no recordará ni cómo escribir el programa de "Hola mundo". La mente funciona así; la teoría es necesaria, pero si uno no practica, no aprende. Por algo el dicho dice *la práctica hace al maestro*, y no la lectura.

En la realidad actual, donde la metodología virtual está tomando cada vez mayor relevancia, serán su voluntad y disciplina, y no tanto su capacidad, las que determinarán sus resultados. Si ingresó al IB ya demostró que posee capacidad intelectual suficiente. No deje que la falta de disciplina limite su desarrollo académico/profesional.

Contacto

Si tiene problemas para instalar el IDE o simplemente quiere consultar algo, le dejamos nuestras direcciones de e-mail. No dude en escribirnos!

- horacio.fontanini@ib.edu.ar
- eduardo.tapia@ib.edu.ar
- fabian.lema@ib.edu.ar
- sergio.encina@ib.edu.ar
- paola.cordoba@ib.edu.ar
- sebastian.anguiano@ib.edu.ar
- claudio.sigvard@ib.edu.ar