Aprendiendo a programar (en C++)

Carolina Lucía González
Román Agustín Castellarin

5 de julio de 2015

Índice

1. Introducción

Programar es decirle a la computadora qué tiene que hacer, es darle instrucciones en algún lenguaje que pueda comprender.
Existen muchos lenguajes de programación. En este apunte sólo veremos uno: C++.

¿Cómo está estructurado C++?

- **Variables tipadas:** Muchas veces necesitamos que la computadora guarde algunos valores porque los debe usar luego. Estos lugares donde guarda esos valores se llaman *variables*. ¿Qué es eso de *tipadas*? Supongamos que las variables son cajas. No todas las cajas tienen el mismo tamaño, no podemos guardar cualquier cosa en cualquier caja. Podemos decir que a cada caja le corresponde un *tipo* de elementos que pueden ser almacenados en ella. Más adelante se explicará con más detalle.

- **Bucles:** Con ellos le podemos pedir a la computadora que repita algo muchas veces, hasta que se cumpla cierta condición. Por ejemplo: un profesor de educación física le puede pedir a sus alumnos que corran por el gimnasio hasta que suene el silbato.

- **Condicionales:** Con ellos planteamos dos alternativas según cierta condición que se puede cumplir o no. Por ejemplo: si está lloviendo entonces nos quedamos en casa, si no llueve entonces vamos al parque.

- **Funciones:** Son como pequeñas máquinas que cumplen una tarea determinada. Por ejemplo: si sabemos que la calculadora puede resolver productos entre dos números y tenemos que hacer muchos de estos, no tiene sentido que los hagamos a mano, usemos la calculadora.

- **Interacción con el usuario:** El usuario es quien usa el programa, no el programador (aunque a veces puede coincidir). Necesitamos tener una forma de mostrar mensajes en pantalla y también de pedir datos.

- **y más...**

2. Nuestro primer programa

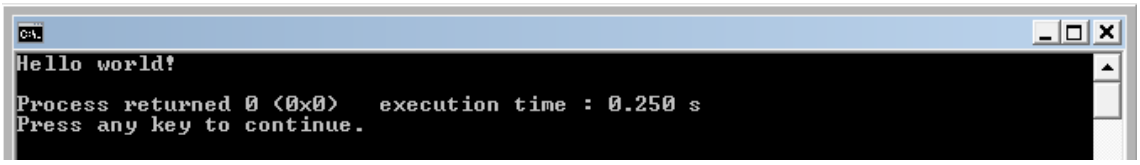
El siguiente es un código C++ que muestra un mensaje:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Muy lindo pero ¿qué se hace con esto?
En principio vamos a ignorar todo, confiando en que el apunte sabe hacer buenos programas.

De a poco vamos a ir aprendiendo qué significa cada una de las líneas del mismo.

Si abrimos este programa con algún IDE[1] y apretamos el botón de *compilar y ejecutar*, deberá aparecer una pantallita negra[2] con el mensaje "Hello world!". Esta pantallita se llama *consola*, y es similar a la siguiente:



2.1. Compilación y ejecución

Compilar no es lo mismo que ejecutar, y tampoco se hace al mismo tiempo. Primero se compila y luego se ejecuta.

Compilar es traducir el código a algo que la máquina entienda mejor (lenguaje binario). Si el código tiene errores de gramática, no va a poder compilarse y, por ende, no se podrá ejecutar. Es por ello que si no compila debemos inmediatamente revisar si nos olvidamos algún punto y coma o escribimos algo mal.

Ejecutar es hacer que la computadora efectúe las instrucciones que le dimos, es correr el programa. Si el programa corre pero no hace lo que queremos, lo más probable es que la lógica de lo que hayamos escrito no sea la correcta.

Para programas chicos, compilar no requiere demasiado tiempo y por lo general uno hace ambas cosas siempre (aunque sólo quiera ejecutar).

2.2. Mostrar mensajes en la pantalla

`cout` es una herramienta que permite mostrar mensajes.
Para entender un poco su funcionamiento podemos cambiar el texto "Hello world!" por otro, volver a compilar y ejecutar, y ver qué pasa. Efectivamente el mensaje mostrado cambia.
A tener en cuenta: el texto debe estar entre comillas.

Varios mensajes pueden ser encadenados en un único `cout`:

```
cout << "Hola" << " mun" << "do!" << endl;
```

`endl` es un *fin de línea*, un *enter*.
Si ponemos

```
cout << "Hola";
cout << " mundo!" << endl;
```

[1] Integrated Development Environment (ambiente de desarrollo interactivo o entorno de desarrollo integrado). Por ejemplo: Code::Blocks, Geany. Aquí pueden encontrar un pequeño instructivo para instalar Code::Blocks.
[2] A veces es de otro color.

la salida será “Hola mundo!”, porque en el primer `cout` no pusimos el fin de línea.

Hay algunos caracteres que deben escribirse de otra forma:

Símbolo	Significado
\n	enter
\t	tabulación
\\	\
\”	”
\’	’
\a	señal audible

De este modo podemos ver que

```
cout << "Hola mundo!" << endl;
cout << "Hola mundo!\n";
```

hace dos veces lo mismo.

Con `cout` también podemos mostrar valores:

```
cout << 2 << endl;
cout << 2*3 << endl;
cout << "1+1 = " << 1+1 << endl;
```

Al poner una cuenta, se muestra el resultado.

2.3. Comentarios

Son porciones de código ignoradas por la computadora. Sirven para hacer anotaciones.
Hay varias formas de hacer comentarios:

```
// Comentario hasta el final de la linea
/* comentario
de varias
lineas */
/** comentario de varias lineas
con otro color (en algunos editores, no en todos) */
/* este comentario puede inscrutarse en el medio del codigo */
```

Veamos un ejemplo concreto:

```
/* Programa de ejemplo
para el apunte
*/
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl; // Muestra el mensaje "Hola
mundo!"
    return /*esto no afecta al codigo*/0;
}
```

Si lo ejecutan van a ver que hace exactamente lo mismo que el primer código.

2.4. Ejercicios

1. Mostrar la siguiente tabla:

Nombre	Nota 1	Nota 2
Ana	7	8
Daniel	5	3
Ovidio	10	10

2. Hacer esa misma tabla pero con líneas divisorias. Sugerencia: usar guiones y contar (a mano, por ahora) cuántos deberían ir.

3. Variables

3.1. ¿Qué es una variable?

Una variable está formada por un *identificador* (también le podemos decir "nombre") y un *espacio de la memoria*. Por ejemplo: podemos tener una variable llamada x que tenga asignados 32 bits de memoria.

3.2. Tipos

Como las variables tienen un espacio asignado, no podemos guardar cualquier dato en ellas. Lo mínimo que hay que pedir es que ese dato quepa en ese espacio. Pero además tenemos que saber cómo guardar ese dato, así después lo podemos recuperar correctamente.
Algunos tipos que vamos a usar son: `int` (números enteros), `float` (números no enteros, "con coma"), `char` (caracteres).

Por una cuestión de simplicidad, a lo largo de esta sección trabajaremos con variables de tipo `int`.

3.3. Declaración y asignación de valores

Veamos un ejemplo del uso de variables:

```
#include <iostream>

using namespace std;

int main()
{
    int x;
    x = 5;

    return 0;
}
```

Primero tenemos que decirle a la computadora que queremos utilizar una variable. Esto lo hacemos con la línea `int x;`.
Luego le asignamos un valor, en este caso es 5. `x = 5;`

Se pueden declarar varias variables:

```
#include <iostream>

using namespace std;

int main()
{
    int a,b;
    a = 8;
    b = 3;

    return 0;
}
```

Y podemos hacer cosas más complicadas, como `x = 10 + 1;` (que le asignará el valor 11 a `x`), o `b = a*3;` (que le asignará a `b` el triple del valor de `a`).
Una opción más complicada es `x = x+1`, veamos un ejemplo:

```
int x;
x = 2;
x = x+1;
```

En la tercer línea le estamos asignando el valor 3 a `x`, ya que primero hace la suma `x+1` que le da 3 pues `x` tiene guardado el 2 en ese momento.

3.4. Identificadores

El identificador (o nombre) de una variable es una secuencia de letras y dígitos (el guión bajo _ cuenta como letra).
El primer carácter debe ser una letra: nombres como `1x` no son válidos.
Las letras minúsculas y mayúsculas son diferentes: `Hola` y `hola` son nombres de variables distintos.
Se puede usar cualquier palabra para el nombre de una variable, excepto por las *palabras reservadas* de C++, que son las que el editor resalta en color (por ejemplo, main, int, namespace).

3.5. Mostrar sus valores en pantalla

Cuando queremos ver los valores que guardamos en ciertas variables, lo podemos hacer con `cout`.

```
#include <iostream>

using namespace std;

int main()
{
    int x;
```

```
    x = 5;

    cout << "x = " << x << endl;

    return 0;
}
```

Esto debería mostrar

```
x = 5
```

3.6. Ingresar sus valores por pantalla

Probemos el siguiente código:

```
#include <iostream>

using namespace std;

int main()
{
    int x;

    cout << "Ingrese un valor: ";
    cin >> x;
    cout << "El valor ingresado es: " << x << endl;

    return 0;
}
```

Cuando nos muestre "Ingrese un valor", ingresemos un número entero, por ejemplo, el 3. Ahora el programa nos dice "El valor ingresado es: 3".

`cin` permite ingresar valores *en tiempo de ejecución*. Esto sirve para que nuestro programa sea reutilizable con nuevos valores, sin necesidad de volver a compilar.

3.7. Ejercicios

1. Hacer un programa que permita el ingreso de dos números enteros y luego muestre la suma de ambos.

2. Extender el programa anterior para que también muestre producto, cociente y resta. ¿Qué pasa cuando intentamos dividir por 0?

4. Condicionales

A la hora de hacer un programa muchas veces se nos presentan situaciones en las que tenemos que hacer distintas cosas según si pasa algo o no.
`if` sirve justamente para resolver esto. Se pregunta qué pasa en una determinada situación. Por ejemplo:

"Estoy organizando una fiesta. Si llueve, la hago bajo techo. Si no, la hago en el patio."

Se usa así:

```
if(condicion){
    lo que queremos hacer si la condicion se cumple
}else{
    lo que queremos hacer si la condicion NO se cumple
}
```

El siguiente ejemplo se fija si un número es mayor a otro:

```
if(a > b){
    cout << "A es mayor a B" << endl;
}else{
    cout << "A es menor o igual a B" << endl;
}
```

Se puede usar `else` como no. Generalmente se usa porque tenemos que hacer cosas, pero si no es necesario hacer nada en el caso de que no se cumpla la condición, no se usa.

Para algunas cosas, como comparar si es mayor, menor, mayor o igual, o menor o igual, no hay problema con los signos, es decir, se usan como siempre ($>$, $<$, $>=$, $<=$). El problema surge cuando queremos ver si se cumple una igualdad:

```
if(a == b){
    cout << "A es igual a B" << endl;
}
```

Como vemos, se tiene que usar un doble signo (==).
Para *distinto*, a diferencia de otros lenguajes, se usa != .

4.1. Varias condiciones

¿Qué pasa si hay una sola respuesta a más de una posibilidad?
En castellano uno diría, por ejemplo:
"Si llueve *o* hay mucho viento, me quedo en casa."
"Si hace calor *y* hay sol, voy a la playa."

En C++ esto se pone con `||` (representa *o*) y && (representa *y*).
Podríamos tener:

```
if(a<b || a==0){
    cout << "A es menor a B o A vale 0." << endl;
}
```

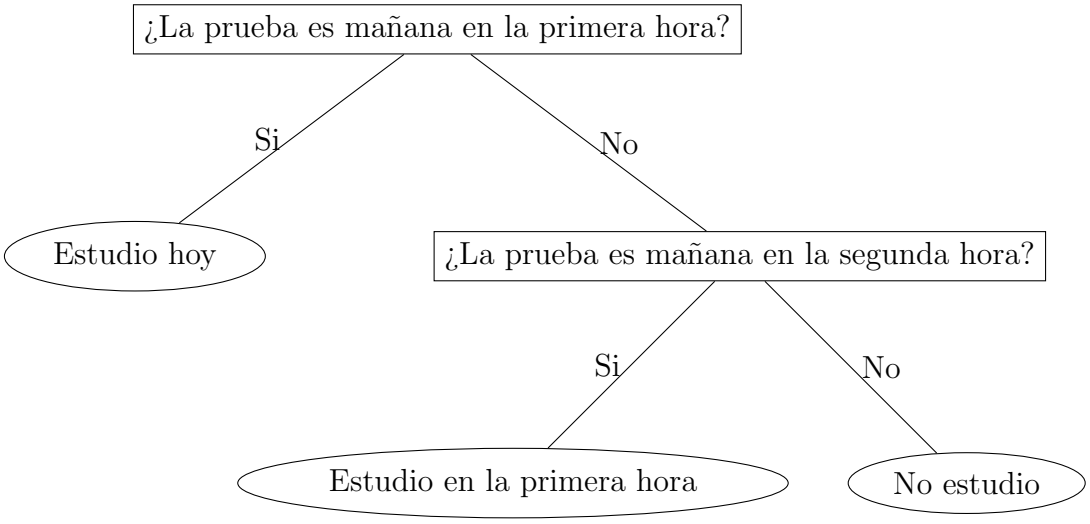
O también:

```
if(a<b && b<c){
    cout << "A es menor a B y B es menor a C." << endl;
}
```

Ahora consideremos un caso más complejo. Por ejemplo:

"Si la prueba es mañana en la primera hora, estudio hoy. Si no: si es mañana pero en la segunda hora, estudio durante la primera hora. Si no, no estudio."

Gráficamente:



En C++:

```
if( la prueba es manana en la primera hora ){
    estudio hoy
}else{
    if( la prueba es manana en la segunda hora ){
        estudio en la primera hora
    }else{
        no estudio
    }
}
```

4.2. El operador ? :

Se trata de una expresión condicional que devuelve un valor. Se utiliza así:

```
condicion ? expV : expF
```

Se evalúa la condición, si es verdadera, se evaluará expV y expF será ignorada; si es falsa, se evaluará expF y expV será ignorada. El resultado (el valor devuelto) será el de la expresión evaluada.

En principio vamos a pedirle a expV y expF que sean del mismo tipo, aunque esto no sea necesariamente así.

Ejemplos:

```
int s = x < 0 ? -1 : 1;

float f = x == 0 ? 3.14 : 2.72;

int maxAB = a > b ? a : b;
```

```
cout << "Hay " << n << (n != 1 ? " elementos." : " elemento.") <<
endl;
```

Hacer

```
variable = condicion ? expV : expF
```

es lo mismo que hacer

```
if(condicion){
    variable = expV;
}else{
    variable = expF;
}
```

Y podemos ver claramente que el operador `?:` simplemente nos ahorra escritura.

4.3. Ejercicios

1. A la *calculadora* de la sección anterior, agregarle cosas: que muestre un mensaje cuando queremos dividir por 0, que nos pregunte qué operación queremos hacer.

2. Hacer un programa que pida dos números, a y b, y muestre un cartel que diga "a es múltiplo de b" o "a no es múltiplo de b", según corresponda.

3. Dados tres números, determinar el mayor de ellos.

4. Hacer un programa que pida las coordenadas de 3 puntos (6 enteros) y diga si el tercer punto esta dentro del rectángulo determinado por los otros dos. *Usar solamente un if.* Nota: el rectángulo queda determinado por dos vértices opuestos.

5. Tipos y operadores

5.1. Tipos y tamaños de datos

Estos son los tipos de datos básicos en C++:

- Tipos de caracteres: representan un solo carácter, como 'A' o '$'. El único de estos tipos es `char`, de un solo byte.

- Tipos de enteros: almacenan un número entero, como 7 o 1024. Los hay de varios tamaños, y pueden ser con signo (aceptan negativos) o sin signo (`unsigned`, sólo positivos y cero). Los más usados de este grupo son `int` y `long long int`, siendo este último capaz de almacenar números más grandes.

- Tipos de números *de punto flotante*: representan valores reales, o sea, "números con coma", como 3,14 o 0,503. Hay distintos niveles de precisión, dependiendo del tamaño que se le asigne. En este grupo nos encontramos con `float` y también con `double`, que tiene mayor precisión que `float`.

- Tipos *booleanos*: de estos hay un único tipo: `bool`. Representa dos valores: verdadero o falso.

5.1.1. char

Seguramente nos parecerá raro e inclusive ilógico hacer cuentas del estilo `'D' + 3 - 'A'`, pero esto es totalmente válido en C++, y su resultado es 6.
La razón por la cual esto no es un disparate es que existe una equivalencia entre los caracteres (`char`) y los números enteros. Esta equivalencia se puede ver en la *tabla ASCII*, donde a cada carácter se le asigna un único número, entre 0 y 255, distinto al de los demás caracteres.
Así, escribir

```
char c = 'A';
```

es exactamente lo mismo que

```
char c = 65;
```

Los números están asignados siguiendo cierto criterio. De esta forma, 66 corresponde al carácter 'B', 67 a la 'C' y así consecutivamente, por lo que la cuenta anterior se reduce a $68+3-65=6$.
'A' no es lo mismo que 'a': 'A' es 65 y 'a' es 97.

Para ver el valor numérico de un carácter, podemos hacer:

```
char c;
cin >> c;
int valor = c;
cout << valor << endl;
```

5.1.2. bool

Como mencionamos antes, las variables de tipo `bool` sólo tienen dos valores posibles: *verdadero* (`true`) o *falso* (`false`).

Podemos tener

```
bool a = true;
bool b = false;
```

Pero también

```
bool c = 1 < 2;
bool d = true && false;
```

y `c` será igual a `true`, y `d` será igual a `false`.

5.2. Operadores aritméticos

$+$	suma
$-$	resta
$*$	producto
$/$	división
$\%$	módulo

$a\,\%m$ devuelve el resto de la división de a por m. Por ejemplo: $6\,\%3 = 0$, $15\,\%4 = 3$, $10\,\%13 = 10$.

La suma, la resta y la multiplicación funcionan como es esperado en las variables numéricas. Sin embargo, la división y el módulo pueden resultar un poco más complicadas.

Al dividir dos números enteros (`int`), el resultado es un entero, *redondeado hacia abajo*:
$17/2 = 8$
$15/4 = 3$
También, y sólo entre números enteros (`int`), se puede calcular el resto de dicha división:
$17\,\%2 = 1$
$15\,\%4 = 3$

Por suerte, al trabajar entre números que permiten decimales (`float`), la división se realiza sin sorpresas adicionales:
$17{,}0/2{,}0 = 8{,}5$
$15{,}0/4{,}0 = 3{,}75$

5.2.1. Operadores de incremento y decremento

Ahora vamos a ver dos operadores *unarios* que son ampliamente usados en casi todos los programas.
Estos operadores son los de incremento (`++`) y decremento (`--`), los cuales actúan sumándole y restándole 1 al operando, respectivamente. Son operadores que actúan únicamente sobre variables, por lo que expresiones como `(a+b)++` o `--5` no tienen sentido y dan error al compilar.
Existen dos versiones de estos operadores: *prefija* y *postfija*.

Versión prefija
Se modifica el valor de la variable y ese es el resultado de la operación. Ejemplo:

```
int a = 5;
int b = ++a;
```

Luego de ese fragmento de código tenemos `a` = 6 y `b` = 6, ya que en la asignación de `b` primero se incrementa el valor de `a` en 1, y luego se lo consulta. Esto equivaldría a:

```
a = 5;
a = a + 1;
b = a;
```

Versión postfija
Se modifica el valor de la variable, pero el resultado de la expresión es el valor original de la variable (antes del incremento o decremento). Ejemplo:

```
int a = 5;
int b = a++;
```

Luego de ese fragmento de código tenemos `a` = 6 y `b` = 5, ya que en la asignación de `b` primero se consulta el valor de `a`, y luego se le incrementa en 1. Esto equivaldría a:

```
a = 5;
b = a;
a = a + 1;
```

5.2.2. Operadores de asignación

Las expresiones tales como

```
n = n+2;
```

en las que la variable del lado izquierdo se repite inmediatamente en el lado derecho, pueden ser escritas de forma compacta:

```
n += 2;
```

El operador `+=` se llama *operador de asignación*.

La mayoría de los operadores binarios (como `+`, `-`, `*` y `/`) tienen un correspondiente operador de asignación. De esta forma tenemos que:

`a += b`	equivale a	`a = a + (b)`
`a -= b`	equivale a	`a = a - (b)`
`a *= b`	equivale a	`a = a * (b)`
`a /= b`	equivale a	`a = a / (b)`

Los paréntesis en `b` están por situaciones como la siguiente: `x *= y+1`, que significa `x = x * (y+1)` y no `x = x*y + 1` (notemos que son expresiones distintas).

5.3. Conversiones de tipo

Cuando se realizan operaciones entre diferentes tipos numéricos, todos los operandos se convierten al tipo de mayor precisión.
El orden de precisión es: `char` < `int` < `float`.

Por lo tanto, las siguientes igualdades valen en C++:
65.0/5.0 = 65.0/5 = 65/5.0 = 'A'/5.0 = 13.0

Hay que tener cuidado de no cometer errores del siguiente tipo:

```
float v = 5 / 2;   // v = 2.0
cout << 'A' + 2 << endl; // Muestra 67, no 'C'
```

Para evitar estos problemas, que a veces pueden estar más encubiertos, se puede hacer lo siguiente: colocar `(<tipo>)` antes de la expresión que deseamos transformar a ese tipo.
Por ejemplo:

```
float v = (float)5 / 2;
cout << (char)('A'+ 2) << endl;
```

De esta forma estamos diciendo, que 5 es un `float` y que `'A'+ 2` es un `char`.
Notemos que si ponemos `(float)(5 / 2)` estamos transformando a `float` el resultado de 5/2, que ya vimos que es 2.
En el otro caso queremos que el resultado (67) sea tratado como `char`. La suma de chars siempre se considera un número entero, por lo tanto `'A' + (char)2` también hubiese mostrado 67.

5.4. Precedencia y orden de evaluación

Al igual que en matemática, algunas operaciones se resuelven antes que otras: el producto y la división se hacen antes que la suma y la resta, además se resuelve de izquierda a derecha.
En la siguiente tabla se presentan la precedencia y la asociatividad de los operadores de C:

Operadores	Asociatividad
`++ -- (tipo)`	derecha a izquierda
`* / %`	
`+ -`	
`< <= > >=`	
`== !=`	
`&&`	
`||`	
`?:`	derecha a izquierda
`= += -= *= /= %=`	derecha a izquierda

Nota: todas las asociatividades son "izquierda a derecha" salvo que se indique lo contrario. Los operadores que están en el mismo renglón tienen la misma precedencia. Los renglones están en orden de precedencia descendente.

5.5. Ejercicios

1. Dadas las siguientes variables:

```
int a = 20, b = 11;
float c = 2.5;
```

   ¿cuáles serán los valores de las siguientes variables?

```
int m = a / b + a;
float n = c * ( a / b );
float o = ( c * a ) / b;
int p = c;
```

2. Ingresar una letra minúscula y mostrarla en mayúscula.

3. Ingresar un carácter. Si es una letra minúscula, mostrarla en mayúscula. Si es una letra mayúscula, mostrarla en minúscula. Si no es una letra, mostrarlo sin cambios.

4. ¿Qué valores tendrán las siguientes variables al finalizar el fragmento?

```
char a = 'd';
a -= 3;
int b = 2;
int c = b++ * 7;
int d = --b - ++c;
int e = b+1;
```

5. (Hacer luego de ver la sección 6.) ¿Existe alguna diferencia de comportamiento entre

```
for(int i = 0; i < 5; ++i )
    cout << i << endl;
```

   y

```
for(int i = 0; i < 5; i++ )
    cout << i << endl;
```

   ?

6. Bucles

Sirven para repetir instrucciones. Hay varias formas de hacer esto.

Nota (que se entenderá mejor luego de mirar las siguientes subsecciones): Entre las instrucciones a ejecutar debe haber alguna que cambie la condición, de lo contrario se tendría un *bucle infinito* (salvo que esto sea lo que se busca). Un bucle infinito es algo que nunca termina, son instrucciones que se repiten infinitas veces.

6.1. while

```
while( condicion ){
    instrucciones a ejecutar mientras se cumpla la condicion
}
```

Veamos un ejemplo concreto:

```
#include <iostream>

using namespace std;

int main()
{
    int n;
    n = 1;
    while(n < 3){
        cout << n << endl;
        n = n + 1;
    }

    return 0;
}
```

La salida de este programa es

```
1
2
```

Hagamos de cuenta que somos la computadora y ejecutemos:

- $n = 1$
  ¿$1 < 3$? Sí, entonces
  - mostrar 1
  - $n = 1 + 1$
- $n = 2$
  ¿$2 < 3$? Sí, entonces
  - mostrar 2
  - $n = 2 + 1$
- $n = 3$
  ¿$3 < 3$? No, entonces terminamos acá.

6.2. for

Tiene 4 partes.

```
for( parte1 ; parte2 ; parte3 ){
    parte4
}
```

Es equivalente a:

```
parte1
while(parte2){
    parte4
    parte3
}
```

¿Por qué existe entonces? Ahorra escritura y (a veces) deja los programas más claros y entendibles.

Su uso más común es:

```
for(i=a;i<b;i++){
    instrucciones
}
```

Lo que hace es ejecutar ciertas instrucciones para i desde a hasta b.

Veamos el mismo programa del `while` pero con con `for`:

```
#include <iostream>

using namespace std;

int main()
{
    int n;
    for(n=1;n<3;n++){
        cout << n << endl;
    }

    return 0;
}
```

6.3. do-while

```
do{
    cuerpo
}while(condicion);
```

¿Qué tiene de distinto al while?

```
while(condicion){
    cuerpo
}
```

En `do-while`, el cuerpo se ejecuta primero y luego se comprueba la condición. En `while` primero se verifica la condición y después se ejecuta el cuerpo.
Veamos un ejemplo en cual son distintos:

```
#include <iostream>

using namespace std;

int main()
{
    int i;

    /** while **/
    i = 0;
    while(i > 0){
        cout << "while: " << i << endl;
        i = i-1;
    }
    cout << "Despues del while: i = " << i << endl << endl;

    /** do-while **/
    i = 0;
    do{
        cout << "do-while: " << i << endl;
        i = i-1;
    }while(i > 0);
    cout << "Despues del do-while: i = " << i << endl;

    return 0;
}
```

¿Qué les parece que debería mostrarse en pantalla luego de ejecutarlo? Ejecútenlo y comprueben si es así.

6.4. Ejercicios

1. Mostrar los números de 1 a n, donde n es un número ingresado al ejecutar el programa.
2. Ingresar dos números, m y n, y dibujar un rectángulo de $m \times n$ con caracteres ASCII, similar al siguiente:

```
+-------+
|       |
|       |
+-------+
```

3. Calcular el promedio de 10 números (`float`) ingresados por el usuario.
4. Ingresar b y e, siendo b `float` y e `int`, y calcular b^e.
5. Mostrar los divisores de un número positivo n ingresado por el usuario. Si el número ingresado no es positivo, mostrar un mensaje de error.
6. (Difícil) Determinar si el número n ingresado es *primo*. Nota: un número es primo si tiene solamente 2 divisores (1 y sí mismo). El 1 no es primo.

7. Arrays (o arreglos, o matrices)

Supongamos que, por alguna razón, necesitamos 100 variables. ¿Qué hacemos? ¿Debemos declararlas a todas a mano?
En matemática la hacen fácil: uno diría que tiene las variables $x_0, x_1, \ldots, x_{99}$. En informática tomamos esta idea de la siguiente manera:

```
<tipo> x[100];
```

Con esta línea se declaran 100 variables (de tipo <tipo>) que se llaman `x[0]`, `x[1]`, ..., `x[99]`.

Los *arrays* nos permiten declarar muchas variables de una sola vez. Todos los elementos de un *array* son del mismo tipo, y para acceder a cada elemento se usan *índices*. A la cantidad de índices que hacen falta para identificar un elemento en un array se la llama *dimensión*.

La sintaxis de declaración es la siguiente:

```
<tipo> <nombre>[<tamaño1>]([<tamaño2>]...) = {<inicialización>};
```

Ejemplos de arrays:

```
char Palabra[10];    // Array de una dimensión, de 10 variables de
tipo char
int Grilla[243][7]; // Array de dos dimensiones, de 243*7 = 1701
variables de tipo int
```

Como los arrays declaran muchas variables, se necesita usar un índice para identificar una variable en particular. Los índices se numeran a partir del CERO, como se ve en el ejemplo:

```
int ListadoDeCalificaciones[35];

ListadoDeCalificaciones[0] = 8;  // Primera variable
ListadoDeCalificaciones[1] = 10; // Segunda variable
...
ListadoDeCalificaciones[34] = 6  // Ultima variable.
```

Los arrays pueden ser inicializados sólo cuando son declarados, utilizado *llaves* o, en el caso de arrays de chars, utilizando comillas. Si no se menciona el *tamaño* de un array, el compilador lo deduce a partir de la inicialización por llaves. Si el tamaño indicado es mayor a la cantidad de elementos provistos, el resto se completa con el valor nulo por defecto (`false`, `0`, `0.0` ó '\0').
Ejemplos:

```
char p[] = "Hola"; // Es lo mismo que char p[5] = {'H', 'o', 'l',
'a', '\0'};

int lista[10] = { 2, -5, 3 };  // El resto se completa con ceros.
```

Se puede recorrer fácilmente un array utilizando *bucles*:

```
int M[6] = {1, 24, -5, 3}; // M vale {1, 24, -5, 3, 0, 0}
for(i = 0; i < 6; i++){
    cout << "El elemento " << i+1 << " es: " << M[i] << endl;
}
```

Muestra en pantalla:

```
El elemento 1 es: 1
El elemento 2 es: 24
El elemento 3 es: -5
El elemento 4 es: 3
El elemento 5 es: 0
El elemento 6 es: 0
```

Detalle:

```
char p[] = "Hola"; // Es lo mismo que char p[5] = {'H', 'o', 'l',
'a', '\0'};
```

El *terminador* ('\0') se agrega automáticamente al inicializar el array de esta forma.

7.1. Operaciones prohibidas

- Los arrays NO pueden ser copiados, ni sumados, ni restados, *ni nada*. Para lograr cualquiera de estas cosas se debe proceder elemento por elemento.

```
float A[10], B[10];
A = B;  // NO
```

- Los arrays NO pueden ser mostrados ni ingresados directamente sin recorrerlos (hay que mostrar elemento por elemento)[3].

```
cout << A << endl;  // NO
cin >> A;           // NO
```

- Los arrays NO pueden ser devueltos por una función.

```
int[5] f(){  // NO, NO, NO, NO, NO !
    int A[5];
    return A;
}
```

7.2. Cadenas de caracteres

¿Qué ocurre cuando hacemos un array de enteros? Simple, obtenemos una lista de enteros. Pero ¿qué sucede al declarar una lista de caracteres? Obtenemos una lista de caracteres, obvio, que fuera de la informática le solemos llamar palabras y frases.

Los *arrays de char* se pueden recorrer mientras no se alcance el *terminador* ('\0').

[3] Excepto los arreglos de caracteres, que veremos en la siguiente subsección.

```
char M[10] = "Pala";
M[3] = 'o';  // Ahora M dice "Palo";
for(i = 0; M[i] != '\0'; i++)
    cout<< "El elemento " << i+1 <<" es: " << M[i] << endl;
```

Esto muestra en pantalla:

```
El elemento 1 es: P
El elemento 2 es: a
El elemento 3 es: l
El elemento 4 es: o
```

Los arrays de char SÍ se pueden mostrar directamente sin tener que recorrer todos los elementos:

```
cout << "La palabra es: " << M << "." << endl;
```

La salida será:

```
La palabra es: Palo.
```

También pueden ser leídos directamente:

```
cin >> M;  // Lee hasta un blanco (espacio, tabulación o enter)
cin.getline(M,10);  // Lee hasta un enter o hasta que se hayan
ingresado 9 caracteres.
```

Para manipular cadenas de caracteres se usan las funciones de la biblioteca `cstring`, la cual debe agregarse como `#include<cstring>`. Las más usadas son:

```
strcpy(A,B); // Copia la palabra B en A (A <- B)
strcat(A,B); // Copia la palabra B al final de A (A <- A++B). Es
decir, concatena las palabras.
strcmp(A,B); // Compara A y B. Devuelve 0 si son iguales.
strlen(A);   // Devuelve la cantidad de caracteres de A.
```

7.3. Ejercicios

1. Ingresar n números enteros y calcular el mínimo, el máximo y el promedio de ellos. Si el número n (ingresado por el usuario) no es positivo o es mayor a 10, se deberá volver a pedir n (esto se repetirá tantas veces como haga falta hasta que n cumpla con lo pedido).

2. Ingresar 10 números y luego mostrarlos en el orden inverso. Ejemplo: si se ingresaron los números 1, 2, 3, ..., 10, hay que mostrar 10, ..., 3, 2, 1.

3. Ingresar su nombre y apellido, luego mostrarlos de la forma “Apellido, Nombre”.

4. Repetir el ejercicio anterior pero con 10 personas (los nombres deben ser mostrados después de haber ingresado todos). Se puede asumir que la longitud máxima de un nombre o apellido es de 100 letras.

5. Pedir una *frase* y decir cuántos caracteres tiene. Se puede asumir que la frase ingresada tendrá menos de 100 caracteres.

6. (Difícil) Pedir una *palabra* y decir si es un *palíndromo* (o sea, si se lee igual de izquierda a derecha que de derecha a izquierda).
Algunos ejemplos para probar si funciona: allá, Ana, arenera, ananá, Neuquén, oso, radar, reconocer, somos, sometemos. Nota: no ingresar eñes ni tildes.

7. (Más difícil) Pedir una *frase* y decir si es un *palíndromo*. No deben tenerse en cuenta los caracteres que no son letras.
Algunos ejemplos: "La ruta natural", "Se van sus naves", "No di mi decoro, cedí mi don", "Amo a la OMA", "Oirás orar a Rosario". Nota: no ingresar eñes, tildes, "¿" ni "¡".

8. Funciones

Como se comentó en la introducción, una función es una maquinita que cumple una tarea determinada.
Una función que suma dos números enteros puede ser:

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

En la primera línea estamos diciendo que la función `suma` devuelve un valor de tipo `int` y toma dos valores, también de tipo `int`. Lo que está entre llaves es el código de la función, las instrucciones que va a ejecutar cada vez que la invoquemos.
`return` es la instrucción que nos sirve para decir qué devuelve la función. En el caso de `suma`, nos devuelve el valor de la variable `r`.
Cuando hacemos `suma(1,2)`, se *llama* a la función `suma` y se ejecuta su código.

Aquí podemos ver un programa completo que hace uso de esta función suma:

```
#include <iostream>

using namespace std;

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

int main()
{
    cout << suma(1,2) << endl;

    return 0;
}
```

En general: una función se declara y define de la siguiente forma:

```
<tipo retorno> <nombre>(<tipo1> <parámetro1>, ...){
        <código>
}
```

Toda función debe ser declarada antes de las funciones que la invoquen, pero puede ser definida luego. Por ejemplo:

```
#include <iostream>

using namespace std;

int suma(int,int);

int main()
{
    cout << suma(1,2) << endl;

    return 0;
}

int noHagoNada(int a){
    return a;
}

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

Notemos que main es una función que no toma ningún parámetro y devuelve un entero.

Las variables que declaramos dentro de una función no tienen validez fuera de ella, no existen o no son las mismas.
Miremos el siguiente código:

```
#include <iostream>

using namespace std;

int suma(int a, int b){
    int r;
    r = a + b;
    cout << "dentro de suma, r = " << r << endl;
    return r;
}
```

```
int main()
{
    int r = 0;

    cout << "fuera de suma, r = " << r << endl;
    suma(1,2);
    cout << "fuera de suma, r = " << r << endl;

    return 0;
}
```

La salida es

```
fuera de suma, r = 0
dentro de suma, r = 3
fuera de suma, r = 0
```

justamente porque la `r` en `suma` es una nueva variable, que no tiene nada que ver con la `r` en `main`. Lo mismo hubiese ocurrido con las variables `a` y `b` que están declaradas como parámetros de `suma`.
Esto nos sirve para no tener que preocuparnos por los nombres de variables que usamos dentro y fuera de las funciones.

Otra cosa que podemos ver en el código anterior es que llamamos a la función pero en ningún momento estamos usando el valor de retorno. Esto obviamente no es un problema, de hecho lo hacemos mucho más seguido de lo que pensamos.

8.1. Funciones que no devuelven valores

Un caso particular de funciones es el de aquellas que no devuelven ningún valor. Por ejemplo, podríamos tener una función que sólo muestre mensajes:

```
void f(int n){
    cout << n << endl;
}
```

`void` significa *vacío*, y, en este caso, sirve para indicar que la función no retorna nada.

8.2. Ejercicios

1. Hacer una función que diga si un número es par. Poner un ejemplo simple dentro de main para verificar si funciona.

2. Hacer una función `max2` que tome dos números enteros y devuelva el máximo de ellos.

3. Hacer una función `max3` que tome tres números enteros y devuelva el máximo de ellos. Usar `max2` en alguna parte.

4. Renombrar a `max2` y a `max3` como `maximo`. Verificar que funciona.
   A veces es muy útil tener funciones que se llamen igual (porque de alguna forma, hacen lo mismo) pero que puedan recibir distinta cantidad de parámetros.

5. Hacer una función que diga si una frase es palíndromo.

9. Variables globales

Una variable se llama *global* si no está declarada dentro de una función.
Esta variable será *visible* para todas las funciones del programa[4] que hayan sido definidas después[5] que ella.

```
#include <iostream>

using namespace std;

int g; // Variable global

int sumarG(int n){
    return n+g;
}

void cambiarG(int v){
    g = v;
}


int main()
{
    cout << sumarG(10) << endl;

    g = 5;
    cout << sumarG(10) << endl;

    cambiarG(2);
    cout << sumarG(10) << endl;

    return 0;
}
```

La salida de este programa será:

```
10
15
12
```

porque el valor de g fue cambiando a lo largo del programa.

Cuando declaramos, y no inicializamos, una variable dentro de una función, ésta se llena de "basura", es decir, un valor aleatorio que no conocemos. Pero cuando la declaramos global, su valor inicial será nulo (`false`, `0`, `0.0` ó '\0').

[4] Asumiendo que el programa consiste en un solo archivo, no nos metamos por ahora en cosas más complicadas.
[5] "Después" quiere decir "más abajo".

10. Estructuras

Las estructuras son una herramienta que nos permite agrupar una serie de datos relacionados en un mismo paquete, pudiendo manejar el paquete como una sola variable.
Al declarar una estructura, estamos en realidad definiendo un nuevo tipo de datos que nos va a servir luego para declarar variables o arrays de paquetes.

La sintaxis es la siguiente:

```
struct <nombre_tipo>{
        <tipo1> <nombre_atributo1>;
        <tipo2> <nombre_atributo2>;
        ...
};
```

IMPORTANTE: la definición de un tipo estructura siempre termina en punto y coma.

10.1. Inicialización

Al igual que los arrays, las estructuras pueden ser inicializadas sólo cuando son declaradas, utilizando notación de llaves.

10.2. Acceso (operador “.”)

Para acceder a un miembro de la estructura se usa el operador punto “.”.

10.3. Asignación (operador “=”)

Las variables de tipos estructuras pueden ser copiadas común y corrientemente, incluso si tienen arrays (¡estos se copiarán correctamente si tener que recorrerlos elemento por elemento!).

Ejemplo práctico:

```
#include <iostream>

using namespace std;

struct persona{
    char nombre[32];
    int edad, DNI;
};

int main()
{
    persona A, B = { "Roman", 18, 39654321 };
    A = B;
    cout << "Soy " << A.nombre << " y mi DNI es " << A.DNI << endl;

    return 0;
}
```

Si bien podemos declarar estructuras donde queramos, conviene hacerlo globalmente para que el nuevo tipo definido sea visible en varias funciones.

10.4. Ejercicios

1. Definir una estructura llamada *punto* donde se almacenen las coordenadas de un punto en el plano.
2. Ingresar un punto por el teclado. Luego, realizar dos copias de ese punto.
3. Definir una estructura del tipo *rectángulo* a partir de los vértices inferior izquierdo y superior derecho. Usar la estructura *punto* para el tipo de los vértices.
4. Escribir una función que devuelva el área de un rectángulo.
5. Escribir una función que diga si dos rectángulos se superponen.
6. Declarar un array de 10 puntos. Ingresarlos y mostrarlos. Encontrar un rectángulo que los cubra a todos y mostrar las coordenadas de sus vértices.

11. Lectura y escritura de archivos

Hay varias formas de hacer esto, pero presentaremos sólo una, la que consideramos más adecuada para el nivel de apunte.
Miremos el siguiente código:

```
#include <iostream>
#include <cstdio>

using namespace std;

int main()
{
    freopen("entrada.in","r",stdin);  // Con esta linea cin va a
leer desde "entrada.in" en lugar del teclado
    freopen("salida.out","w",stdout); // y con esta cout va a
imprimir al archivo "salida.out" en lugar de la pantalla

    int a,b,c;
    cin >> a >> b >> c;
    cout << a+b+c << endl;

    return 0;
}
```

Es necesario crear `entrada.in` en la misma carpeta donde está el programa, y completarlo con los datos que exija el mismo. `salida.out` se creará automáticamente al ejecutar el programa.

Vamos a usar `freopen` de la siguiente forma:

```
freopen(<nombre archivo>,"r",stdin);  // para el archivo de entrada.
freopen(<nombre archivo>,"w",stdout); // para el archivo de salida.
```

En el caso de tener que leer la entrada de un archivo pero que la salida deba ser por la consola, simplemente no ponemos la línea que abre el archivo de salida. Lo mismo ocurre si no tenemos que leer desde un archivo.

11.1. Ejercicios

1. Rehacer el problema 4 de la sección 7, pero esta vez leyendo los nombres desde un archivo y escribiéndolos en otro.

2. Rehacer el problema de la frase palíndromo, pero esta vez leyéndola de un archivo. La salida debe seguir siendo la consola. Extender el tamaño de la frase a 500 caracteres.

3. Hacer dos programas: uno que escriba 10.000 números cualquiera en un archivo, y otro que calcule el promedio de todos ellos. ¿Da bien? Tener cuidado con el *overflow*[6].

4. Rehacer el problema 6 de la sección 10. Los puntos se leen del archivo `puntos.in`. La primera línea de ese archivo contiene un número n que indica la cantidad de puntos que hay. Luego hay n líneas, cada una describiendo un punto (con dos números x, y separados por un espacio). La salida, en el archivo `puntos.out`, deben ser dos líneas: la primera con las coordenadas del vértice inferior izquierdo del rectángulo, y la segunda con las del superior derecho.

12. ¿Cómo seguir aprendiendo?

Si leyeron todo el apunte, hicieron todos los ejercicios y, fundamentalmente, *lo entendieron*, entonces ¡felicitaciones! ¡ya están en condiciones de avanzar por su cuenta!
No obstante, recomendamos seguir profundizando estos conceptos. Con todo lo visto en este apunte (y con un poco de ingenio) pueden resolver la mayoría de los problemas de programación que se les van a plantear.

Pueden seguir mirando en `c.conclase.net/curso` o en `www.cplusplus.com` (este último para los que saben inglés).

Para empezar a resolver problemas del estilo de las olimpíadas de informática, les recomendamos crear una cuenta en `train.usaco.org/usacoregister` e ir siguiendo el curso (sólo la parte para registrarse está en inglés, después se puede poner en español).
También en `bit.do/oiapoli` pueden encontrar más problemas, pero estén atentos al nivel de dificultad indicado porque no todos son adecuados para quienes recién empiezan.

Referencias

[1] Brian W. Kernighan, Dennis M. Ritchie, *El lenguaje de programación C*, Prentice-Hall, 1991.

[2] `c.conclase.net/curso`

[6] Desborde. Ocurre cuando se quiere almacenar un valor muy grande en una variable. Podemos pensarlo como servir demasiada agua en un vaso, cuando supera su capacidad se desborda.