



# El lenguaje C++

# Conceptos básicos

## Programa

Conjunto de símbolos que especifican instrucciones, generalmente para ser ejecutadas por una computadora.

## Lenguaje de programación

Es un mecanismo de abstracción compuesto por un conjunto de reglas gramaticales que permite especificar las instrucciones que se le da a una computadora.

## Compilación

Proceso por el cual se transforma un programa (en algún lenguaje especificado) y lo transforma a una representación binaria (lenguaje de máquina) ya que las computadoras solo entienden ceros y unos (bits).



# Un poco de historia...

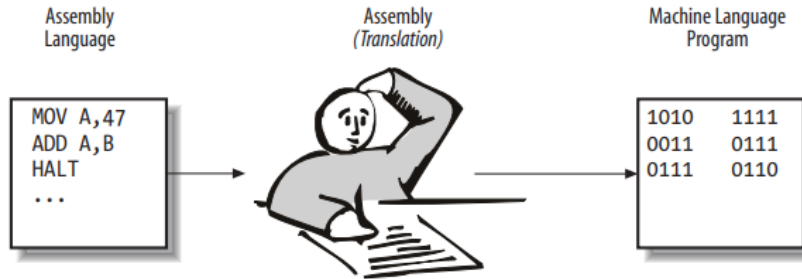
## ¿Cómo era programar en los tiempos antiguos?

Las computadoras solo entienden lenguaje binario, es decir instrucciones que son 0s y 1s, esto es el lenguaje de máquina.

Un típico programa podría lucir como el cuadro de la derecha

1000	1001
0010	1010
0101	1011
...	...

Programar de esta forma era tedioso por lo que se creó el lenguaje ensamblador (*assembly language*), un lenguaje de bajo nivel donde las instrucciones se representaban por palabras claves que podían trasladarse a lenguaje de máquina. Sin embargo, esta traducción inicialmente se hacía manualmente.



Fuente: Practical C++ Programming [1]

Posteriormente se hizo un programa que se encargara de hacer esta traducción automáticamente, un programa *ensamblador*.

Finalmente los lenguajes fueron evolucionando, pareciéndose más al lenguaje humano. Esto son los lenguajes de alto nivel como C++, Python, Java, etc.



Es un lenguaje de programación de alto nivel creado en 1980 como la continuación del lenguaje C.

Es el lenguaje preferido en programación competitiva debido a su rapidez.

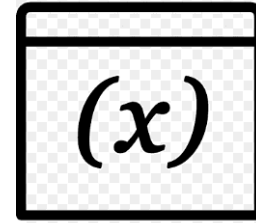
Existen distintas versiones de C++: C++98, C++11, C++14, C++17, C++20.



# Elementos básicos de los lenguajes de programación

## Variable

Es un símbolo abstracto con nombre que sirve para almacenar (en memoria) un valor que puede cambiar en el tiempo.



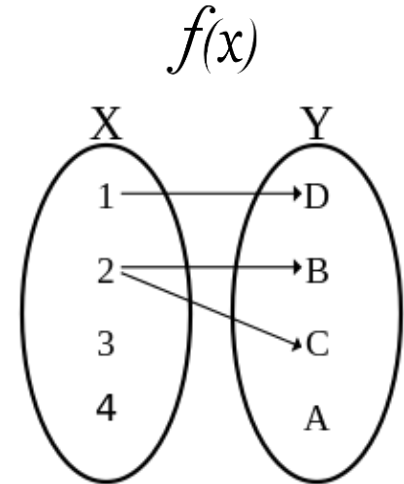
## Sentencia

Es una instrucción que realiza una acción

## Función

Es una generalización de las funciones matemáticas que mapean un conjunto de entrada a otro de salida. Así como las variables, también poseen nombre.

La distinción es que en programación, las funciones pueden no tener ningún parámetro de entrada o ningún parámetro de salida. Además de también tener un **cuerpo** conformado por una serie de sentencias que realizan una acción.



# Elementos básicos de los lenguajes de programación

## Operador

Similar a las funciones pero se representan por medio de un símbolo aritmético en vez de un nombre.



## Tipos de dato

Es aquello que categoriza a una variable y permite especificar su dominio (conjunto de valores permitidos) y conjunto de operaciones. Entre los principales, tenemos a los enteros, los reales, los caracteres y los textos.

## Librería

Colección de sentencias, funciones, etc. previamente escritas que permiten realizar operaciones útiles.



# Ejemplo de un programa simple en C++



# Ejemplo de un programa simple en C++



```
1  #include <algorithm>
2  #include <cmath>
3  #include <iostream>
4
5  int main() {
6      std::cout << "hello, world" << std::endl;
7      return 0;
8  }
```



```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      cout << "hello, world" << endl;
6      return 0;
7  }
```

Librería que incluye la gran mayoría de librerías estándar.

Evita tener que poner std::delante de cada función estándar.



# Comentarios

Anotaciones en los códigos que no tienen **ningún** efecto en la ejecución del programa. Sirven como una forma de descripción extra para la persona que lee el código.

## Tipos:

- ☐ Comentarios simples

```
// Este es un comentario simple  
int x = 5; // También se puede poner al final de una expresión
```

- ☐ Comentarios multi-línea

```
/*  
Este es un comentario multi línea  
que se expande en varias líneas.  
*/
```

# Tipos de Datos - Primitivos

## Enteros

Nombre	# bits	# bytes	Rango
<i>short</i>	16	2	$[-2^{15}, 2^{15} - 1] \approx [-3 \times 10^5, 3 \times 10^5]$
<i>int</i>	32	4	$[-2^{31}, 2^{31} - 1] \approx [-2 \times 10^9, 2 \times 10^9]$
<i>long long</i>	64	8	$[-2^{63}, 2^{63} - 1] \approx [-9 \times 10^{18}, 9 \times 10^{18}]$

### Warning!

Si sobrepasas los límites del rango, ocurre un *overflow*, y los valores suelen empezar a “rotar”. (Si sobrepasas el límite superior vas al inferior y viceversa)

## Enteros sin signo

Nombre	# bits	# bytes	Rango
<b><i>unsigned short</i></b>	16	2	$[0, 2^{16} - 1] \approx [0, 6 \times 10^5]$
<b><i>unsigned int</i></b>	32	4	$[0, 2^{32} - 1] \approx [0, 4 \times 10^9]$
<b><i>unsigned long long</i></b>	64	8	$[0, 2^{64} - 1] \approx [0, 10^{19}]$

```
short x = 5;
int x = 5;
long long x = 5;

unsigned short x = 5;
unsigned int x = 5;
unsigned long long x = 5;
```

# Ejemplo overflow

```
overflow.cpp > main()
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      short x = 32767;
6      cout << x << endl;
7
8      short y = 32768;
9      cout << y << endl;
10
11     return 0;
12 }
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 overflow.cpp -o overflow && ./overflow
32767
-32768
```



# Tipos de Datos - Primitivos

## Decimales


Nombre	# bits	# bytes	Rango aproximado	Precisión
<i>float</i>	32	4	$\pm 1.18 \times 10^{-38}$ a $\pm 3.4 \times 10^{38}$	6-9 decimales
<i>double</i>	64	8	$\pm 2.23 \times 10^{-308}$ a $\pm 1.8 \times 10^{308}$	15-17 decimales
<i>long double</i>	80-96	12	$\pm 3.36 \times 10^{-4932}$ a $\pm 1.19 \times 10^{4932}$	18-21 decimales
	128	16	$\pm 3.36 \times 10^{-4932}$ a $\pm 1.19 \times 10^{4932}$	33-36 decimales

Los números reales tienen una representación particular similar a la fórmula  $\pm \text{mantisa} \times 2^{\text{exponente}}$ .

La mayoría de problemas en programación competitiva que trabajen con números reales te pedirán un error (relativo o absoluto) máximo de  $10^{-6}$ .

En C++ se puede usar notación exponencial para definir un número decimal.

```
float x = 1.6;  
double x = 1.6;  
long double x = 1.6;  
long double x = 1e10;
```



# Tipos de Datos - Primitivos

## Booleanos

Nombre	# bits	# bytes	Valores
<i>bool</i>	8	1	Verdadero ( <i>true</i> ) y falso ( <i>false</i> )

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

## Caracteres

Cada carácter se expresa entre comillas simples en C++ o también pueden expresarse mediante su código entero en ASCII

Nombre	# bits	# bytes	Valores
<i>char</i>	8	1	256 caracteres ASCII.

```
char c = 'a';  
char c = 97; // same as 'a'  
char c = '\n'; // end line character
```

# Tipos de Datos - Compuestos

## Cadenas

Los *strings* son Conjunto de caracteres. Se representan con comillas dobles. Vienen implementados en la librería *string*.

```
string message = "hello, world";
```

## Pairs

Pares que agrupan 2 tipos de datos cualquiera. Para acceder al primer elemento se usa *.first* y para el segundo se usa *.second*

```
pair<int, string> p = {5, "test"};  
cout << p.first << " " << p.second;
```

## Auto


Detecta automáticamente el tipo de dato, siempre que sea posible.

```
int a = 5;  
int b = 6;  
auto x = a + b;
```

# Constantes

En vez de definir variables, también puedes definir constantes poniendo la palabra **const** antes del tipo de dato. Si intentas cambiar una constante, tendrás un error de compilación.

```
const double PI = 3.1415926536;  
PI = 3;
```



Error

# Casting

Es el proceso por el cual se transforma el tipo de dato de una variable a otro.

**Ejemplo:** Algunas conversiones válidas de int a double

```
int a = 5;  
double b = (double)a + 7.2;  
double b = double(a) + 7.2;  
double b = 1.0 * a + 7.2;
```

# Lectura y Escritura

La librería *iostream* nos provee de la función *cin* para la lectura de datos y *cout* para la escritura.

La función *cin* utiliza los caracteres `>>` para separar las variables que se leen mientras que *cout* utiliza `<<` para las variables que se imprimen. Adicionalmente al imprimir las variables se puede agregar un *endl* que significa un fin de línea o un enter.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int a, b;
6      cin >> a >> b;
7      cout << a << " , " << b << endl;
8      return 0;
9  }
```



# Lectura y Escritura rápida

La lectura y escritura estándar de C++ suele ser muy lenta para programación competitiva. Pero existen algunas sugerencias para hacerla más rápida:

## ❑ Desactivar sincronización con funciones de C

En el lenguaje C el input y output eran manejados con las funciones *scanf* y *printf*. C++ quiso mantener esta compatibilidad y realiza una sincronización entre las funciones de lectura y escritura de C con las de C++, pero esto hace más lento el proceso.

→ Utiliza la sentencia ***ios\_base::sync\_with\_stdio(false);*** para desactivar esta sincronización

```
test.cpp > main()
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios_base::sync_with_stdio(false);
6
7      int x;
8      cin >> x;
9      cout << x << endl;
10 }
```

# Lectura y Escritura rápida

## ❑ Desactivar sincronización entre cin y cout

Por default, *cin* esta atado a *cout*. Esto es útil para programas que interactúan con el usuario, pero en programación competitiva, la mayoría de veces esto no es necesario ya tenemos el input de antemano por lo que podemos desactivar esto para agilizar el programa.

→ Utiliza la sentencia ***cin.tie(0);*** para desconectar *cin* de *cout*.

Ten cuidado de los efectos secundarios si es que haces un programa que interactúe con un usuario.

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main() {
5     cout << "Ingresa tu edad: ";
6     int age;
7     cin >> age;
8     cout << "Tu edad es: " << age << endl;
9 }
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 test.cpp -o test && ./test
Ingresa tu edad: 20
Tu edad es: 20
```

```
test.cpp > main()
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main() {
5     ios_base::sync_with_stdio(false);
6     cin.tie(0);
7
8     cout << "Ingresa tu edad: ";
9     int age;
10    cin >> age;
11    cout << "Tu edad es: " << age << endl;
12 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 test.cpp -o test && ./test
20
Ingresa tu edad: Tu edad es: 20
```

# Lectura y Escritura rápida

## ❑ Reemplazar *endl* por el caracter de fin de línea

La sentencia *endl* suele ser más lenta debido a que hace una operación extra llamada *flush* que imprime en la pantalla todo lo que esté en el buffer (donde se guarda el output). El fin de línea no hace esto por lo que es más rápido.

```
test.cpp > main()
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int x;
6      cin >> x;
7      cout << "El valor de x es " << x << endl;
8      int y;
9      cin >> y;
10     cout << "El valor de y es " << y << endl;
11 }
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 test.cpp -o test && ./test
1
2
El valor de x es 1
El valor de y es 2
```

```
test.cpp > main()
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios_base::sync_with_stdio(false);
6      cin.tie(0);
7
8      int x;
9      cin >> x;
10     cout << "El valor de x es " << x << "\n";
11     int y;
12     cin >> y;
13     cout << "El valor de y es " << y << "\n";
14 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 test.cpp -o test && ./test
1
2
El valor de x es 1
El valor de y es 2
```

# Lectura y Escritura rápida



```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      cout << "Hola mundo" << endl;
6      return 0;
7  }
-
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios_base::sync_with_stdio(false);
6      cin.tie(0);
7
8      cout << "Hola mundo\n";
9      return 0;
10 }
```

# Lectura y Escritura — Precisión de decimales

Para poder imprimir números reales (*float*, *double*, *long double*) algunos problemas te pedirán un primer un número exacto de decimales, otros te pedirán que el error relativo y absoluto no exceda cierto valor. Por ejemplo, suele ser muy común un mensaje como este:

Your answer is considered correct if its absolute or relative error doesn't exceed  $10^{-6}$ . Namely, if your answer is  $a$ , and the jury's answer is  $b$ , then your answer is accepted, if  $\frac{|a-b|}{\max(1,|b|)} \leq 10^{-6}$ .

La librería *iomanip* te permite especificar cuántos decimales quieres imprimir, utilizando la función *setprecision(x)*. Usualmente también debe ser acompañado de *fixed* para que la parte entera no se vea afectada.

**Nota:** La precisión especificada solo sirve para los tipos de datos decimales, los enteros seguirán siendo mostrados sin decimales.

```
test.cpp > main()
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios_base::sync_with_stdio(false);
6      cin.tie(0);
7      cout << fixed << setprecision(6);
8
9      cout << 1.0 / 3.0 << "\n";
10     double a = 1;
11     int b = 1;
12     cout << a << " " << b << "\n";
13
14     return 0;
15 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 test.cpp -o test && ./test
0.333333
1.000000 1
```

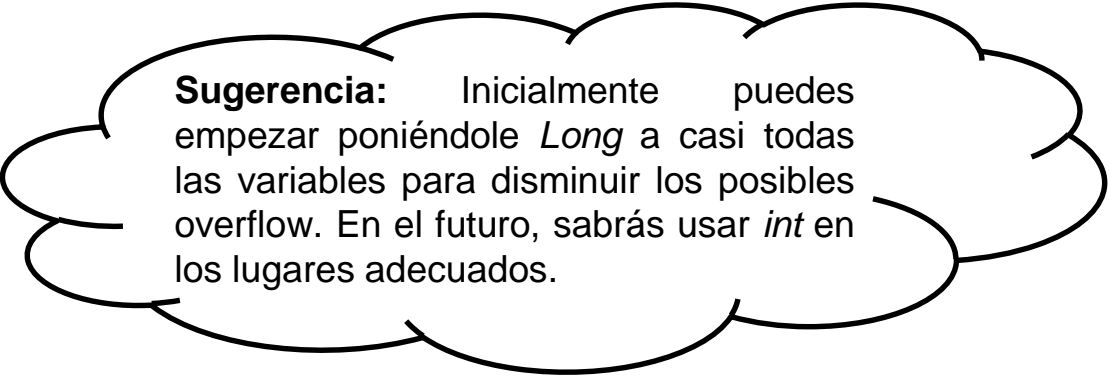
# Alias para tipos de datos

Con **using** podemos ponerle un “alias” a los tipos de datos, lo cual es útil para poder acortarlos.

El formato es el siguiente

**using** [new\_name] = [original\_name]

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  using Long = long long;
5  using Double = long double;
6
7  int main() {
8      ios_base::sync_with_stdio(false);
9      cin.tie(0);
10
11     Long x = 5;
12     Double y = 6.1;
13     return 0;
14 }
```



**Sugerencia:** Inicialmente puedes empezar poniéndole *Long* a casi todas las variables para disminuir los posibles overflow. En el futuro, sabrás usar *int* en los lugares adecuados.

# Operadores

Operador	Descripción	Ejemplo
+	Suma	<code>int x = a + b;</code>
-	Resta	<code>int x = a - b;</code>
*	Multiplicación	<code>int x = a * b;</code>
/	División entera cuando ambos operandos son enteros.	<code>int a, b; int x = a / b;</code>
	División real si alguno de los operandos es real	<code>double a, b; double x = a / b;</code>
%	Módulo (residuo de la división). Solo definido para enteros.	<code>int x = a % b;</code>

**División entera:**  
Se elimina la parte decimal

$$5 / 2 = 2$$

$$-5 / 2 = -2$$

$$5 / -2 = -2$$

**División real:**

$$5.0 / 2 = 2.5$$

$$-5.0 / 2 = -2.5$$

$$a = b * (a / b) + a \% b$$

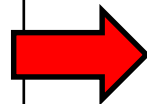
$$5 \% 2 = 1$$

$$-5 \% 2 = -1$$

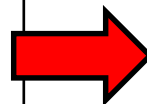
$$5 \% -2 = 1$$

# Operadores

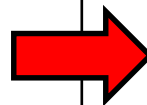
Operador	Descripción	Ejemplo
=	Asignación	<pre>int x = 5;</pre>
++	Operador unario que suma 1 a la variable junto a él. Puede ir a la izquierda o derecha.	<pre>int x = 5; x++;</pre> <pre>int x = 5; ++x;</pre>
--	Operador unario que resta 1 a la variable junto a él. Puede ir a la izquierda o derecha.	<pre>int x = 5; --x;</pre> <pre>int x = 5; x--;</pre>
<pre>+=</pre> <pre>-=</pre> <pre>*=</pre> <pre>/=</pre> <pre>%=</pre>	Suma un valor a la variable Resta un valor a la variable Multiplica por un valor a la variable Divide por un valor a la variable Le saca módulo por un valor a la variable	<pre>int x = 5; x += 3;</pre>



$x$  se vuelve 6



$x$  se vuelve 4

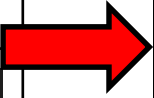


$x$  se vuelve 8



# Operadores

Operador	Descripción	Ejemplo
<code>==</code>	Igual a	<pre>int a = 5; int b = 4; int x = (a == b);</pre>
<code>!=</code>	Diferente a	<pre>int a = 5; int b = 4; int x = (a != b);</pre>
<code>!</code>	Negación	<pre>int a = 5; int b = 4; int x = !(a == b);</pre>
<code>&amp;&amp;</code>	And	<pre>int x = (a == 5) &amp;&amp; (b == 4);</pre>
<code>//</code>	Or	<pre>int x = (a == 5)    (b == 4);</pre>
<code>^</code>	Xor (or exclusivo)	<pre>int x = (a == 5) ^ (b == 4);</pre>

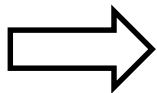


Ambos operadores son cortocircuitos.  
Si con el primer operando pueden determinar el resultado, ya no evalúan el 2do

# Precedencia en expresiones

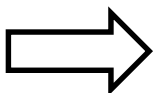
Tenga cuidado, algunas operaciones tienen precedencia sobre otras. Estamos acostumbrados a algunas de ellas (por ejemplo, que la multiplicación tiene precedencia sobre la adición); pero sobre otras, no. Si tiene dudas, coloque paréntesis.

$$a / b * c$$



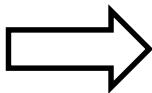
Tienen igual jerarquía, por lo que primero se ejecuta la división y luego la multiplicación

$$a == b \&\& c$$



`==` tiene precedencia sobre `&&`, por lo que se haría en este orden  $(a == b) \&\& c$ . Esto generalmente no es lo que uno quiere hacer, por lo que es mejor definir los paréntesis  $a == (b \&\& c)$

$$a == b + c$$



`+` tiene precedencia sobre `==`, por lo que se haría en este orden  $a == (b + c)$ . Esto es generalmente lo que uno quiere.

# Precedencia en expresiones

**TABLE 1-4** Operators available in C++

Operators organized into precedence groups	Associativity
() [] -> .	<i>left</i>
<i>unary operators:</i> - ++ -- ! & * ~ (type) sizeof	<i>right</i>
* / %	<i>left</i>
+ -	<i>left</i>
<< >>	<i>left</i>
< <= > >=	<i>left</i>
== !=	<i>left</i>
&	<i>left</i>
^	<i>left</i>
	<i>left</i>
&&	<i>left</i>
	<i>left</i>
? :	<i>right</i>
= <i>op=</i>	<i>right</i>

Fuente: *Programming Abstractions in C++* [2]

# Condicionales — If Else

Se utilizan cuando se requiere evaluar una expresión booleana (que puede ser verdadera o falsa) y que dependiendo de eso se realice alguna acción.

```
if (condition) {  
    // do something  
} else {  
    // do something else  
}
```

## ❑ Simple *if*

```
int grade;  
cin >> grade;  
if (grade < 10) {  
    cout << "Susti con fe\n";  
}
```

## ❑ *If – else*

```
int grade;  
cin >> grade;  
if (grade < 10) {  
    cout << "Susti con fe\n";  
} else {  
    cout << "Bien\n";  
}
```

## ❑ Multiple *if - else*

```
int grade;  
cin >> grade;  
if (grade < 10) {  
    cout << "Susti con fe\n";  
} else if (grade == 20) {  
    cout << "Perfect\n";  
} else {  
    cout << "Bien\n";  
}
```

# Bucles — For loop

Se utilizan cuando se requiere iterar y realizar alguna acción un número de veces. Posee 3 partes (posibles de omitir), que son la inicialización del bucle, la condición con la cual se definirá cuando se termina el bucle, y las actualizaciones que se harán luego de cada iteración.

```
for (initialization; condition; update) {  
    // do something  
}
```

## Ejemplo clásico

```
int n;  
cin >> n;  
for (int i = 0; i < n; i++) {  
    cout << i << "\n";  
}
```

El ejemplo indica que se iniciará la variable  $i$  en 0 y se iterará mientras  $i$  sea menor que  $n$ . En cada iteración se imprimirá el valor de  $i$ , y al terminar cada iteración, el valor de  $i$  aumentará en 1.

Por ejemplo, si  $n = 3$ , se imprimirán los valores 0 1 y 2

## Bucle infinito

```
for (;;) {  
    cout << "Al infinito y más allá\n";  
}
```

En este segundo ejemplo, no hay ningún elemento, por lo que el bucle iterará infinitamente imprimiendo un mensaje en cada iteración.

# Bucles — While loop

Es equivalente al bucle for, solo que tiene otro formato. Se iterará mientras la condición especificada siga siendo verdadera.

```
while (condition) {  
    // do something  
}
```

## Ejemplo equivalente

```
int n;  
cin >> n;  
int i = 0;  
while (i < n) {  
    cout << i << "\n";  
    i++;  
}
```

## Otro ejemplo

```
int n;  
cin >> n;  
while (n > 0) {  
    cout << n % 10 << "\n";  
    n /= 10;  
}
```

## Bucle infinito

```
while (true) {  
    cout << "Al infinito y más allá\n";  
}
```

Todo lo que se puede hacer con *for*, también se puede hacer con *while*. Pero a veces con una opción queda más sencillo que con la otra.

Muchos programadores prefieren usar *for* cuando el número de iteraciones máximas se conoce de antemano, caso contrario usan *while*, pero es una cuestión de estilos

# Alcance de una variable

Las variables poseen un alcance (*scope*), el cual es el área del programa en donde esta variable es válida. Bajo este concepto existen dos tipos de variables.

- ❑ **Variables globales:** Su alcance es en todo el programa.
- ❑ **Variables locales:** Son definidas en un bloque de código y solo tienen alcance en ese bloque. Un bloque de código es una sección encerrada por llaves ({ }). Pueden haber bloques adentro de otros.

Es posible también que dos variables de dos *scopes* distintos se llamen igual. Y en caso el *scope* de una de ella esté dentro del *scope* de otra, la variable más “profunda” será la predominante temporalmente hasta que termine su bloque. Este proceso se llama **shadow** y es mejor tratar de evitarlo

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int global = 5; // global variable
5
6  int main() {
7      int local = 0; // local variable in the main block
8      global++;      // global can be used here
9
10     if (local == 0) {
11         string localIf = "test"; // local variable inside the if block
12
13         for (int i = 0; i < 7; i++) { // i is local variable inside the for block
14             int global = 6; // local variable inside the for block that shadows global
15         }
16         // i cannot be used here
17     }
18     // localIf cannot be used here
19     return 0;
20 }
```

# Alcance de una variable

- Si una variable global no es inicializada explícitamente, obtiene su valor por defecto (por ejemplo, 0 para números, *false* para booleanos)
- Si una variable local no es inicializada explícitamente, puede tener **cualquier** valor en ese momento. Lo cual es muy peligroso si es que se intenta usar la variable sin valor alguno. Tenga en cuenta esto para evitar posibles errores.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int global;
5
6  int main() {
7      int local;
8
9      return 0;
10 }
11
```

→ Inicializa en 0 por default

→ Puede tener cualquier valor



# Static Arrays

Un arreglo sirve para representar un conjunto de elementos con un tamaño fijo. Se pueden crear arreglos de cualquier tipo de dato. El formato es el siguiente:

*type name[size];*

```
const int SIZE = 7;  
int a[SIZE];  
  
double b[100];
```

Los elementos de un arreglo están numerados desde 0. Así el primer elemento es  $a[0]$  el segundo  $a[1]$  y así hasta llegar al elemento  $a[SIZE - 1]$ , siendo  $SIZE$  el tamaño del arreglo.

```
for (int i = 0; i < SIZE; i++) {  
    cout << a[i] << "\n";  
}
```

También se pueden declarar arreglos de más dimensiones, como matrices.

```
int c[10][7];
```

**Importante:** Los arreglos **locales** son más restringidos: almacenan una menor cantidad de elementos (aunque esto depende de la memoria stack asignada al programa) y además no tienen valor por default, mientras que los arreglos **globales** no tienen estos problemas.

# Vectors

Vienen implementados en la librería *vector* y son una versión más poderosa de los arreglos. Los vectores, a diferencia de los arreglos, pueden tener tamaños variables (se les puede agregar nuevos elementos o quitarles). Además los vectores locales no tienen la misma restricción que los arreglos locales.

Su formato es

***vector***<type> name;

```
int main() {
    vector<int> a;
    vector<int> b = {5, 6, 3}; // vector with initialization

    int size = 5;
    vector<int> c(size); // vector of size 5 with all elements initialized in default 0
    vector<int> d(9, -1); // vector of size 9 with all elements initialized in -1

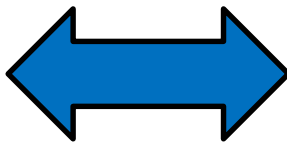
    a.push_back(7); // inserting an element at the end
    a.pop_back();   // deleting the last element

    for (int i = 0; i < c.size(); i++) { // traversing the elements of c
        cin >> c[i];
    }
    return 0;
}
```

# For basado en rango

Es un bucle for especial que nos permite iterar sobre los elementos de un vector de forma más natural sin tener que usar un iterador.

```
vector<int> v;  
for (int i = 0; i < v.size(); i++) {  
    cout << v[i] << "\n";  
}
```



```
vector<int> v;  
for (int x : v) {  
    cout << x << "\n";  
}
```

# Funciones

Las funciones son bloques de código que reciben parámetros, realizan ciertas acciones y luego retornan una respuesta. Permiten reusar código fácilmente. Ejemplos:

```
int f(int x, int y) {  
    return x * x + y * y;  
}
```

```
using Long = long long;  
  
Long factorial(int n) {  
    Long answer = 1;  
    for (int i = 1; i <= n; i++) {  
        answer *= i;  
    }  
    return answer;  
}
```

```
const double PI = 3.1416;  
  
double circleArea(double r) {  
    return PI * r * r;  
}
```

Las funciones también pueden no retornar nada. En este caso se les pone un tipo de dato **void**.

```
void printArray(vector<int> v) {  
    for (int i = 0; i < v.size(); i++) {  
        cout << v[i] << "\n";  
    }  
}
```

# Pasar parámetros por valor y por referencia

- ❑ **Por valor:** La variable es una **copia** de la original, por lo que si dentro de la función haces un cambio a esta variable, esto no afectará a la original. Es el comportamiento por default.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  void update(int x) {
5      x++;
6  }
7
8  int main() {
9      int x = 5;
10     update(x);
11     cout << x << "\n";
12
13     return 0;
14 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 test.cpp -o test && ./test
5
```

- ❑ **Por referencia:** Es la variable original. Cualquier cambio adentro de la función le afecta. Se logra poniendo un & a la izquierda de la variable

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  void update(int &x) {
5      x++;
6  }
7
8  int main() {
9      int x = 5;
10     update(x);
11     cout << x << "\n";
12
13     return 0;
14 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 test.cpp -o test && ./test
6
```

# Pasar parámetros por valor y por referencia

**Recomendación:** Por temas de eficiencia, trata de pasar tipos de datos complejos como **vector** o **string** por referencia. Imagina que tienes un vector de 1 millón de elementos y cada vez que usas una función ¡tuvieras que crear una copia del vector y de todos sus elementos! Sería muy lento.

```
int getSum(vector<int> &v) {  
    int sum = 0;  
    for (int x : v) {  
        sum += x;  
    }  
    return sum;  
}
```

# Funciones clásicas

Función	Descripción	Ejemplo
<i>sqrt(x)</i>	Calcula la raíz cuadrada	<pre>double a = sqrt(x);</pre>
<i>abs(x)</i>	Valor absoluto	<pre>int x = abs(-9);</pre>
<i>cos(x)</i> <i>sin(x)</i>	Coseno, seno de un ángulo (en radianes)	<pre>double angle = 0; double x = sin(angle);</pre>
<i>asin(x)</i> <i>acos(x)</i>	Arcoseno, arcocoseno. Retorna el ángulo en radianes.	<pre>const Double PI = acos(-1);</pre>
<i>gcd(a, b)</i> <i>lcm(a, b)</i>	Máximo común divisor y mínimo común múltiplo	<pre>int x = gcd(6, 8);</pre>
<i>swap(a,b)</i>	Intercambia los valores de dos variables	<pre>int a = 5; int b = 7; swap(a, b);</pre>

# Clases

Las clases son un tipo de dato personalizado con **atributos** y **métodos**, así como la posibilidad de agregar operadores. Además también podemos definirles **constructores**.

La estructura de una clase en C++ es de esta forma:

```
struct ClassName {  
    ...  
};
```

Una de las clases clásicas que puedes necesitar puede ser la del **Punto**.

```
struct Point {  
    int x, y;  
  
    Point() {} // default constructor  
    Point(int x, int y) { // constructor  
        this->x = x;  
        this->y = y;  
    }  
  
    double abs() {  
        return sqrt(x * x + y * y);  
    }  
  
    Point operator+(const Point &other) const {  
        return Point(x + other.x, y + other.y);  
    }  
};  
  
int main() {  
    Point A(3, 9);  
    Point B(0, 7);  
  
    cout << A.abs() << "\n";  
  
    Point C = A + B;  
    return 0;  
}
```



# Truquitos extras

- ❑ Si *long long* no es suficiente, puedes usar un tipo de dato más grande llamado `__int128`. Le puedes poner un alias para que sea más fácil utilizarlo. **Desventaja:** A priori, solo puedes usarlo para cálculos intermedios. No puedes leer una variable de este tipo o imprimirla (a menos que le agregues un método de lectura y escritura).
- ❑ En C++ existen los macros con la sentencia `#define` que permiten reemplazar una expresión por otra. Tienen el formato `#define <identificador_nuevo> <identificador_antiguo>`. Por buenas prácticas, no suele ser tan recomendable utilizarlo, pero hay un macro en particular que puede ser útil para debugging (proceso por el cual encontramos los errores – bugs).

```
using Big = __int128;  
Big x = 5;
```

```
1  #include <bits/stdc++.h>  
2  #define debug(x) cout << #x << " = " << x << endl;  
3  
4  using namespace std;  
5  
6  int main() {  
7      int x = 5;  
8      debug(x);  
9      return 0;  
10 }
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
$ g++ -Wl,--stack=268435456 -O2 -std=c++17 test.cpp -o test && ./test  
x = 5
```

# Truquitos extras

- ❑ Nunca uses los tipos de datos *short* o *float*. Probablemente nunca los necesites. También puedes ignorar los *unsigned*.
- ❑ Mucho cuidado con los números decimales. Si la respuesta no es decimal, sino un entero, es POSIBLE que puedas tratar de evitar los cálculos con números reales y trabajar en enteros. Siempre intenta evitar los números decimales siempre que puedas, a menos que requiera demasiado esfuerzo.

**Ejemplo:** Comparar fracciones



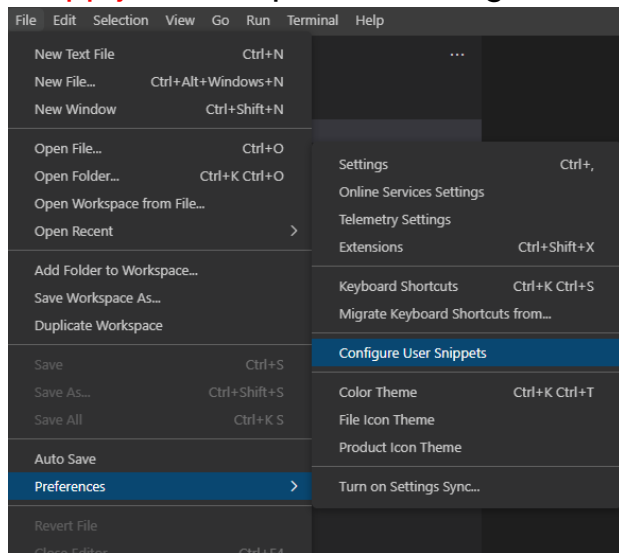
```
int a, b, c, d;  
cin >> a >> b >> c >> d;  
if (1.0 * a / b <= 1.0 * c / d) {  
    // compare  
}
```



```
Long a, b, c, d;  
cin >> a >> b >> c >> d;  
if (a * d <= b * c) {  
    // compare  
}
```

# Truquitos extras

- ❑ Si alguna vez olvidas qué hace una función o algo sobre las librerías de C++, puedes revisar la documentación de C++ en la página <https://cplusplus.com/>
- ❑ En VSCode puedes crear un **snippet**, lo cual te permite guardar una plantilla de código que puedas invocar para ahorrar tiempo. Ve a *File -> Preferences -> Configure User Snippets* y busca la opción de **cpp.json**. Ahí puedes configurar el nombre de tu comando y la plantilla. Puedes ayudarte de [aquí](#).



```
template": {  
  "prefix": "template",  
  "body": [  
    "#include <bits/stdc++.h>",  
    "#define debug(x) cout << #x << \" = \" << x << endl",  
    "using namespace std;",  
    "",  
    "using Long = long long;",  
    "",  
    "int main() {",  
    "  ios_base::sync_with_stdio(false);",  
    "  cin.tie(0);",  
    "  ",  
    "  $0",  
    "  ",  
    "  return 0;",  
    "}",  
    ""  
  ]  
}
```

# Naming convention

- ❑ Si es posible, intenta utilizar nombres en inglés.
- ❑ Utiliza nombres (en variables, funciones, clases) que tengan sentido. Es normal utilizar nombres cortos para algunas variables principales como  $n$  para el tamaño de algo. Pero para variables intermedias más complejas, es mejor ponerles un nombre con significado como *count* o *speed*.
- ❑ Para las variables de los iterados en un for se suelen usar las letras  $i, j, k$  pero no es obligatorio.
- ❑ Cuando vas a nombrar algo que tenga más de 1 palabra, existen algunos estilos comunes:
  - ❑ *snake\_case*
  - ❑ *UPPER\_SNAKE\_CASE*
  - ❑ *camelCase*
  - ❑ *PascalCase*
- ❑ No hay un estilo único aceptado en todo C++, lo importante es **ser consistente**, sobre todo para cuando compites en team. Personalmente yo uso lo siguiente **en C++**:
  - ❑ Para los nombres de las variables y funciones uso camelCase.  
**Ejemplos:** *carSpeed*, *getElements()*
  - ❑ Para los nombres de las clases uso PascalCase.  
**Ejemplo:** *UniversityCourse*
  - ❑ Los nombres de las constantes uso UPPER\_SNAKE\_CASE.  
**Ejemplo:** *MAX\_SPEED*

# Referencias

- ❑ [1] Practical C++ Programming. Steve Oualine.
- ❑ [2] Programming Abstractions in C++. Eric S. Roberts.  
<http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1134/materials/CS106BX-Reader.pdf>
- ❑ [3] Fundamentals of programming – Chapter 2. FTMS College. [https://ftms.edu.my/v2/wp-content/uploads/2019/02/PROG0101\\_CH02.pdf](https://ftms.edu.my/v2/wp-content/uploads/2019/02/PROG0101_CH02.pdf)
- ❑ [4] Understanding Programming Languages. Ben – Ari.  
<https://tigerweb.towson.edu/aconover/Documents/Progammg%20and%20Algorithms/Understanding%20Programming%20Languages.pdf>
- ❑ [5] Floating Point Numbers. <https://www.learncpp.com/cpp-tutorial/floating-point-numbers/>

# ¡Gracias por su atención!

