



## Unidad 05: Tipos, Operadores, y Expresiones

5

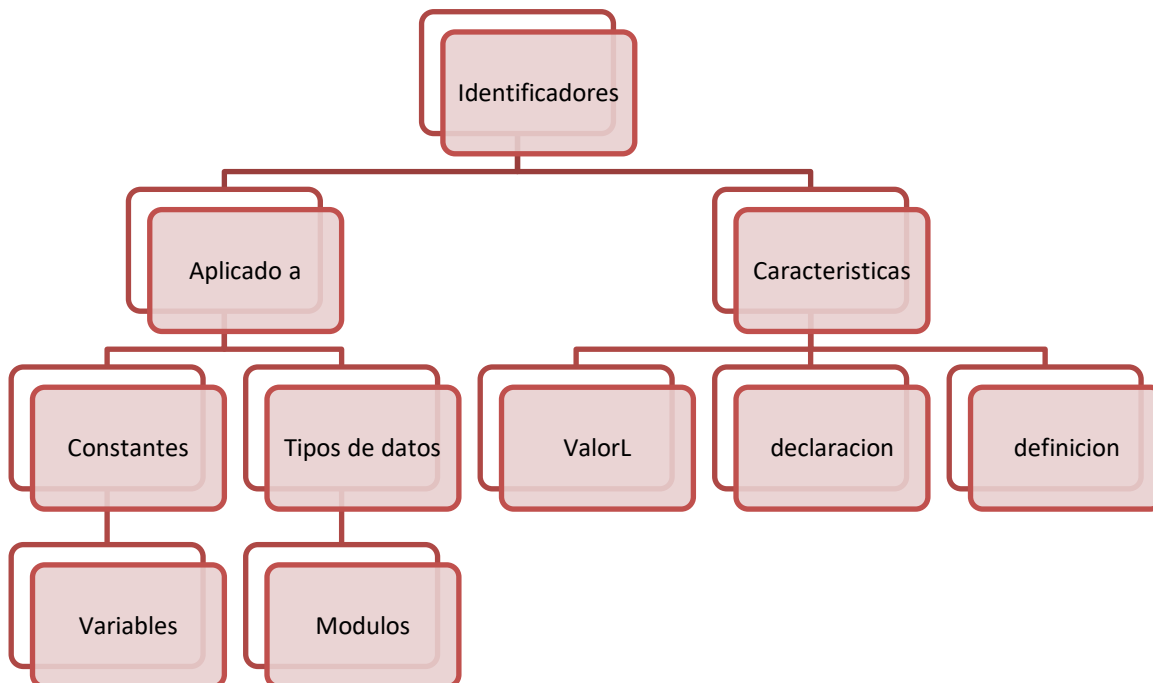
### Contenidos analíticos

#### Parte II: Constructos del Lenguaje: Expresiones, Declaraciones, Sentencias y Estructura de Programa

##### Unidad 5: Tipos, Operadores, y Expresiones

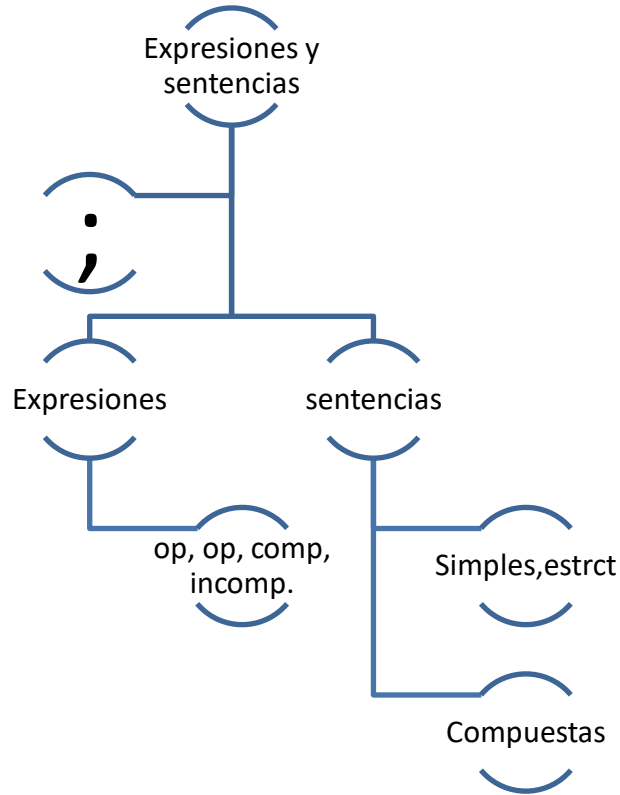
Estructura general de un programa. Concepto de bibliotecas, introducción al concepto de la descomposición: prototipos, firmas y desarrollo de funciones. Concepto de espacios de nombres. Expresiones y sentencias: sentencias simples estructuradas, compuestas. Análisis de caso simples, múltiples, completos, incompletos. Composiciones iterativas: la necesidad de iterar. Composiciones exactas, no exactas, precondicionales, poscondicionales. Iteración versus recursividad. Composiciones recursivas.

### Identificadores





## Expresiones y sentencias



## Fundamentos C++

Sin declaración using	Con declaración using
<pre>//programa para imprimir texto #include &lt;iostream&gt;  int main() {     std::cout &lt;&lt; "Hola\n";      return 0; }</pre>	<pre>//programa para imprimir texto #include &lt;iostream&gt; using std::cout; // using std::cin; using std::endl;  int main() {     cout &lt;&lt; "Hola" &lt;&lt; endl;      return 0; }</pre>



Instruccion	Descripcion
#include	Directiva del preprocesador
<iostream>	Componente de entrada/salida (objetos cin, cout, cerr)
using	Declaración que elimina necesidad de repetir el prefijo std.
int main()	Función principal que retorna un entero
{ }	Definición de un bloque de programa
std::cout	Uso del nombre cout del espacio de nombres std, dispositivo std de salida
::	Operador binario de resolución de alcance
<<	Operador de inserción en flujo
"Hola\n"	Literal Hola + salto de línea (también << std::endl;
;	Finalización de una sentencia
return 0	Punto de finalización correcta de la función

Si la función, como en este caso tiene un encabezado `int main()` debe tener al menos un `return` de un valor entero. Una función `void nombre()` puede finalizar con la instrucción `return` o sin ella.

Programa que muestra la suma de dos enteros
<pre>#include &lt;iostream&gt; int main() {     // declaracion de variables     int numero1;     int numero2;     int suma;      std::cout &lt;&lt; "Escriba el primer entero";     std::cin &gt;&gt; numero1;      std::cout &lt;&lt; "Escriba el segundo entero";     std::cin &gt;&gt; numero2;      suma = numero1 + numero2;      std::cout &lt;&lt; "La suma de " &lt;&lt; numero1 &lt;&lt; " + " &lt;&lt; numero2 &lt;&lt; " es: " &lt;&lt; suma &lt;&lt; std::endl;     return 0; }</pre>



Instrucción	Descripción
Cin	Dispositivo std de entrada
>>	Operador de extracción de flujo
+	Operador de suma
-	Operador de resta
*	Operador multiplicativo
/	Operador de división
%	Operador de modulo o resto
( )	Operador para agrupar expresiones ej: a * (b+c)
==	Operador de igualdad
>	Mayor
>=	Mayor igual
<	Menor
<=	Menor igual
!=	Operador de desigualdad
=	Operador de asignación
+=	Asignación y suma x+=3; equivale a x = x + 3;
-=	Resta y asignación
*=	Multiplicación y asignación
/=	División y asignación
++	Operador de incremento
--	Operador de decremento

Programa que comprara dos enteros, utiliza la declaración using

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
    int numero1;
    int numero2;
    cout << "Escriba dos enteros para comparar";
    cin >> numero1 >> numero2;

    if (numero1 > numero2)
        cout << numero1 << " > " << numero2 << std::endl;

    if (numero1 == numero2)
        cout << numero1 << " == " << numero2 << std::endl;

    if (numero1 < numero2)
        cout << numero1 << " < " << numero2 << std::endl;
    return 0;
}
```



### Palabras Reservadas

#### C y C++

Auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while

#### Solo C++

and and\_eq asm bitand bitor bool catch class compl const\_cast delete dynamic\_cast explicit export false friend inline mutable namespace new not not\_eq operator or or\_eq private protected public reinterpret\_cast static\_cast template this throw true try typeid typename using virtual wchar\_t xor xor\_eq

### Tipos de datos fundamentales y jerarquía de promoción

Tipos de Datos
long double
double
float
unsigned long int
long int
unsigned int
int
unsigned short int
short int
unsigned char
char
bool

### Concepto de biblioteca

Su nombre surge del término inglés “library” cuya traducción inmediata es biblioteca.

Son archivos predefinidos conteniendo funciones, procedimientos, definiciones que el compilador puede incorporar siempre que se las incluya en el código fuente.

Existe una gran cantidad de estas bibliotecas, algunas estándar y otras que pueden ser definidas por los usuarios o desarrolladores

Al incluirlas se pueden utilizar todas las funciones que contienen sin necesidad de una nueva definición

Para su declaración se requiere incluirla a través de una directiva de preprocesador

#### **#include <nombre de la biblioteca>**

Se pueden incluir varias bibliotecas, cada una de las cuales debe tener su propia directiva #include.

Estas directivas no son parte del compilador sino previas de pre procesador, por lo tanto no son parte del lenguaje y no deben terminarse con ; como las sentencias del lenguaje

**iostream** Es la biblioteca que debe estar presente siempre en todas las aplicaciones de la materia. Está especializada en la lectura y escritura y es exclusiva de C++. Otras bibliotecas son también utilizadas pero no en todos los casos

### Espacios de nombre

Es una región declarativa que proporciona un ámbito a los identificadores (nombres de tipos, funciones, variables, etc.) de su interior. Los espacios de nombres se utilizan para organizar el código en grupos lógicos y para evitar conflictos de nombres que pueden producirse, especialmente cuando la base de código incluye varias bibliotecas. Todos los identificadores del ámbito del espacio de



nombres son visibles entre sí sin calificación. Los identificadores que están fuera del espacio de nombres pueden tener acceso a los miembros si usan el nombre completo de cada identificador (por ejemplo `std::cin`), o bien mediante una declaración `Directivas using`

La directiva `using` permite usar todos los nombres en un namespace sin emplear `namespace-name` como calificador explícito.

Es un error tener una variable de espacio de nombres con el mismo nombre que una variable global. Una directiva `using` puede colocarse en la parte superior del archivo `.cpp` (en el ámbito del archivo),

En general, evite colocar directivas `using` en un archivo de encabezado (`*.h`) porque cualquier archivo que incluya ese encabezado pondrá todo en el espacio de nombres en el ámbito, lo que puede ocasionar problemas de ocultación de nombres y colisión de nombres que son muy difíciles de depurar.



*Universidad Tecnológica Nacional*  
*Facultad Regional Buenos Aires*

Doctor Oscar Bruno – Director de cátedra



## Sentencias

<sentencia> ->

<sentencia expresión> |  
<sentencia compuesta> |  
<sentencia de selección> |  
<sentencia de iteración> |  
<sentencia etiquetada> |  
<sentencia de salto>

<sentencia expresión> ->

<expresión>? ;

<sentencia compuesta> ->

{<lista de declaraciones>? <lista de sentencias>?}

<lista de declaraciones> ->

<declaración> |  
<lista de declaraciones> <declaración>

<lista de sentencias> ->

<sentencia> |  
<lista de sentencias> <sentencia>

- La sentencia compuesta también se denomina *bloque*.

<sentencia de selección> ->

**if** (<expresión>) <sentencia> |  
**if** (<expresión>) <sentencia> **else** <sentencia> |  
**switch** (<expresión>) <sentencia>

La expresión que controla un **switch** debe ser de tipo entero.

<sentencia de iteración> ->

**while** (<expresión>) <sentencia> |  
**do** <sentencia> **while** (<expresión>) ; |  
**for** (<expresión>? ; <expresión>? ; <expresión>?) <sentencia>

<sentencia etiquetada> ->

**case** <expresión constante> : <sentencia> |  
**default** : <sentencia> |  
<identificador> : <sentencia>

Las sentencias **case** y **default** se utilizan solo dentro de una sentencia **switch**.

<sentencia de salto> ->

**continue** ; |  
**break** ; |  
**return** <expresión>? ; |  
**goto** <identificador> ;

- La sentencia **continue** solo debe aparecer dentro del cuerpo de un ciclo. La sentencia **break** solo debe aparecer dentro de un **switch** o en el cuerpo de un ciclo. La sentencia **return** con una expresión no puede aparecer en una función **void**.





### Asignación

Asignar significa dar valor a cierta dirección de memoria. Se asocia a un espacio de almacenamiento en la memoria de un ordenador que contiene un valor que puede ser modificado (variable) durante el proceso. La asignación puede ser:

- 1) Interna
  - a) Nombre del identificador = expresión;
- 2) Externa
  - a) De entrada → lleva a memoria un valor:
    - i) Desde el teclado → cin>>
    - ii) Desde otro flujo de datos → archivo
  - b) Sa salida muestra o imprime una dirección de memoria:
    - i) Por pantalla → cout<<
    - ii) Por otro flujo → archivo

### Asignación interna

Dar valor al espacio almacenamiento en memoria asociado a un identificador que refiere a una variable de un determinado tipo de dato, mediante el operador de asignación “=”.

La asignación interna es destructiva, esto es reemplaza el valor contenido previamente por el valor de la expresión que se asigna.

La forma genérica de la asignación es:

**Nombre del identificador = expresión;**

Se debe cumplir con:

El identificador debe ser valorL

La expresión debe ser del mismo tipo que el identificador

La asignación es un operador por lo que **a=b=d=expresión;** le asigna el valor de la expresión a todos los identificadores, asociando de derecha a izquierda

La expresión puede ser completa o incompleta.

int a = 15; declara a como entera e inicializa asignándole el valor de 15.

a = 3\*4; destruye el valor anterior de a reemplazándolo por el resultado de la nueva expresión.

En el ejemplo anterior se asigna al identificador a el valor de la expresión aritmética 3\*4.

También es posible asignar expresiones lógicas o llamadas booleanas, con dos únicos valores posibles. En C toda expresión verdadera tiene el valor de 1 y expresiones falsas el valor es 0.

int a = 5>4; le asigna al identificador el valor de 1 por ser una expresión verdadera. Que cree que asigna, en caso de considerar que sea una expresión válida a = 3>2<35. Justifique su respuesta.

C++ tiene como tipo de dato primitivo el tipo bool, en cuyo caso el identificador solo puede tener dos valores posibles.

La asignación a = a+1; le asigna al identificador el valor que tenía incrementado, en este caso en 1. Obsérvese que el identificador a tiene significado diferente según se encuentre a la izquierda de la expresión o a la derecha. A la izquierda es el valor que se modificara, a la derecha refiere al contenido antes de la modificación.

Por último recuerde que se pueden combinar los operadores aritméticos con el de asignación y que existen los operadores ++ y – que pueden ser post o pre ya sea para incrementar o disminuir



### Asignación Externa

Se puede dar valor a un espacio de almacenamiento asociado a un identificador obteniéndolo de un dispositivo externo (asignación externa de entrada) o derivarla hacia un dispositivo externo (asignación externa de salida).

#### Asignación externa de entrada

En C++ el dispositivo externo de entrada es el teclado, está asociado el identificador **cin** definido en el espacio de nombre STD. Otros flujos dispositivos para obtener datos son los flujos que pueden ser de texto o binarios. Estos serán objeto de estudio más adelante.

La asignación externa de entrada con el dispositivo estándar es **cin>>identificador**; donde:

**cin** es el dispositivo definido en el espacio de nombre std.

**>>** operador de extracción de flujo

**Identificador** una variable valor L

La asignación externa de entrada es destructiva.

#### Asignación externa de salida

En C++ el dispositivo externo de salida es la pantalla, está asociado el identificador **cout** definido en el espacio de nombre STD. Otros flujos dispositivos para obtener datos son los flujos que pueden ser de texto o binarios. Estos serán objeto de estudio más adelante.

La asignación externa de salida con el dispositivo estándar es **cout<<expresión<<expresión**; donde:

**cout** es el dispositivo definido en el espacio de nombre std.

**<<** operador de inserción de flujo

**expresión** puede ser completa, incompleta, literales

La asignación externa de salida no es destructiva.

Ejemplos

Sea int a=10, b=5;

cout<<a;

muestra 10 por pantalla y la próxima asignación la hace inmediatamente después

cout<<a<<endl;

muestra 10 por pantalla y la próxima asignación la hace en la línea siguiente endl produce un salto de línea

cout<<a<<"\n";

muestra 10 por pantalla y la próxima asignación la hace en la línea siguiente "\n" produce un salto de línea.

cout<<" el valor de a es : "<<a<<" el de b "<<b<<" el producto es "<<a\*b<<endl;

muestra → el valor de a es 10 el de b 5 el producto 50, la próxima asignación es en la línea siguiente.

### Estructuras de selección o análisis de caso

Si bien las acciones en un algoritmo deben realizarse una después de otra, muchas veces existe la necesidad de realizar acciones diferenciadas según una determinada condición. En este caso los lenguajes ofrecen sentencias estructuradas llamadas **estructuras de selección o análisis de casos**. Se debe decidir entre un conjunto de alternativas y hacer en consecuencia. También es muy común que un conjunto de acciones, un bloque de sentencias debe repetirse una determinada cantidad de veces. Para esto los lenguajes ofrecen ciclos o estructuras de repetición.

En este apartado el objeto de estudio serán las estructuras de selección y los ciclos-



### Estructuras de selección o análisis de caso

- 1) Estructura de selección simple
  - a) incompleta
  - b) completa
- 2) Estructura de selección múltiple
  - a) incompleta
  - b) completa

#### Estructura de selección simple

Sentencia de selección if incompleta  
forma

if (expresión) sentencia;
---------------------------

if palabra reservada

(expresión) una expresión encerrada entre parentesis, en general una expresión booleana y la sentencia se ejecuta si esa expresión es verdadera. Si la expresión es aritmética y no booleana considera verdadero si la expresión es diferente de cero y falso en caso de ser cero.

La sentencia solo se cumple bajo la condición de verdadero de la expresión. En caso de ser falso se ejecuta la acción inmediata siguiente al condicional en caso de existir. Sentencia puede ser simple, estructurada o compuesta, en este caso se hace necesario crear el bloque de sentencias entre llaves.

#### Ejemplos

Sea int a=10, b=5;

If (a > b) cout <<"hola";

Como a es mayor que b la expresión es verdadera por lo que muestra Hola

If (a < b) cout <<"hola";

Como a es mayor que b la expresión es falsa por lo que no muestra nada

If (a == b) cout <<"hola";

Como a es mayor que b la expresión es falsa, no son iguales, por lo que no muestra nada

If (a = b) cout <<"hola";

Aquí no se utiliza el operador de relación, es el operador de asignación, se le asigna a a el valor de b, 5. Por eso esa expresión vale 5, 5 es distinto de cero, por lo que se entiende como verdadero en cuyo caso muestra Hola

If(a>b) {cout<<"hola"; cout <<"Chau"};

Como a es mayor que b ejecuta las dos sentencias simples compuestas en una sentencia mediante las llaves

If(a>b) {cout<<"hola"; cout <<"Chau"};

Como a es mayor que b ejecuta las dos sentencias simples compuestas en una sentencia mediante las llaves

If(a>b) cout<<"hola"; cout <<"Chau";



Como a es mayor que b muestra Hola, al no tener llaves ejecuta la sentencia simple, Chau lo muestra independientemente del valor de verdad de la expresión

Ejemplo

Ingresar un valor entero e informar si es positivo, negativo o cero

```
int a;
```

```
cin >> a;
```

```
if(a>0) cout << "positivo";
```

```
if(a<0) cout << "negativo";
```

```
if(a==0) cout << "cero";
```

se analiza cada caso y se responde en consecuencia, de las tres sentencias dos de ellas serán falsas

Sentencia de selección if completa

forma

```
if (expresión) sentencia1; else sentencia2;
```

if palabra reservada

(expresión) una expresión encerrada entre parentesis, en general una expresión booleana y dos sentencias. La sentencia1 se ejecuta si esa expresión es verdadera. Si la expresión es aritmética y no booleana considera verdadero si la expresión es diferente de cero y falso en caso de ser cero.

En caso de ser falsa la expresión se ejecuta la sentencia 2.

Ejemplos

Sea int a=10, b=5;

```
If (a > b) cout << "hola"; else cout << "chau";
```

Como a es mayor que b la expresión es verdadera por lo que muestra Hola

```
If (a < b) cout << "hola"; else cout << "chau";
```

Como a es mayor que b la expresión es falsa por lo que muestra Chau

```
If(a>b) {
    cout << "Hola";
    cout << " que tal"
};
else {
    cout << "chau";
    cout << "hasta luego"
}
```

Muestra hola que tal por ser una sentencia completa

```
If(a>b)
    cout << "Hola";
    cout << " que tal"
;
else {
    cout << "chau";
    cout << "hasta luego"
```



}

Al haberse omitido las llaves ejecutara solo hola y allí da por finalizado el condicional, al encontrar luego un else no puede determinar a que le corresponde por lo que generara un error sintáctico ya que la estructura de la freas del condicional es sintácticamente incorrecta.

### If anidados

```
int a;
cin>>a;
if (a>0)
    cout << "positivo";
else
    if(a<b)
        cout<<"negativo";
    else
        cout<<" es cero";
```

### Analisis de caso multiple completo (con clausula default)

```
switch( ordinal){
    case valor1: accion1;
    case valor2: accion2;
    case valor3: accion3;
    ...
    case valorN: accionN;

    default: accionD;
}
```

default es usada para los valores que no correspondieron en casos anteriores

En general se utiliza un break para salir del switch que permitan comportamientos diferentes según el valor del ordinal

```
Switch( ordinal )
{
    case 1: sentencia1; break;
    case 2: sentencia2; break;
    default: sentenciaD; /* break; */
}
```

El analisis de caso multiple incompleto es similar pero sin clausula default, por lo que si la entrada no se corresponde con algun valor no se ejecuta ninguna accion. Cada caso debe responder a un valor unico o una lista no permitiendose subrangos.

### La necesidad de iterar

Estas estructuras permiten repetir sentencias una determinada cantidad de veces, en C existen tres sentencias que lo permiten y estas son: **for; while; do...while**. Cada una de estas estructuras tienen una sintaxis diferente, cada una con sus particularidades aun cuando tienen muchos puntos en comun. Tanto es asi que si bien cada una tiene su aplicacion particular en general todas pueden adaptarse simulando el comportamiento de las otras. Solo existe la excepcion de la condicional



poscondicional que es la única que al menos se ejecuta una vez. Una primera descripción de las mismas puede ser:

- 1) Exactas controlan cantidades
- 2) No exactas evalúan expresiones, se ejecuta siempre que estas sean verdaderas
  - a) Precondicional → evalúa condición al inicio y requiere
    - i) Asignación previa del dato de la expresión lógica
    - ii) Evaluación de la expresión, si es verdadera se ejecuta, de lo contrario finaliza
    - iii) Acciones que correspondan
    - iv) Nueva asignación del dato de la expresión lógica para ver si continúa o termina
  - b) Poscondicional
    - i) Acciones que correspondan
    - ii) Asignación del dato de la expresión para evaluar si continúa o termina

Para la utilización de estas estructuras se requiere controlar:

- ¿Cuál es el tamaño del bloque o, dicho de otro modo, cuántas veces se debe repetir.
  - Esto se determina con una expresión de control y se ejecuta siempre que esa expresión sea verdadera
    - `for(;expresion de control;) [0..N]`
    - `while(expresion de control) [0..N]`
    - `do{ } while(expresion de control); [1..N]`
- ¿Qué acciones se repiten
  - Una sentencia, que puede ser:
    - Vacía
    - Simple
    - Estructurada
    - Compuesta

### Ciclo de repetición for

`for(expresion de inicialización; expresion de control; expresion de salto)`

Esta compuesta por tres sentencias cada una de las cuales finaliza con `;` y, desde el punto de vista sintáctico pueden no estar.

Ejemplo:

```
for(i=0; i<5; i++)
```

la primera vez el iterador `i` parte de 0, en cada paso se incrementa en 1 `i++` y se hace mientras la expresión `i<5` sea verdadero

```
for(i=5; i>0; i--)
```

la primera vez el iterador `i` parte de 5, en cada paso disminuye su valor en 1 `i--` y se hace mientras la expresión `i>0` sea verdadero

```
for(i=0; i<10; i+=2)
```

la primera vez el iterador `i` parte de 0, en cada paso se incrementa en 2 `i+=2` y se hace mientras la expresión `i<10` sea verdadero

```
i=0;
```

```
for(; i<5; )
```

```
  i++;
```



la primera vez el iterador `i` parte de 0 que está inicializado fuera del cuerpo del `for`, en cada paso se incrementa en 1 `i++`, con una acción en el `for` y se hace mientras la expresión `i<5` sea verdadero. Esto es un ejemplo de lo dicho que de las tres sentencias no son obligatorias dentro del encabezado del ciclo `for`

### Otros ejemplos

Cuántas veces se repite y por qué?

```
for(i=0,j=0; i<10&& j<6; i+=2,j++)
```

```
for(i=0,j=0; i<10 | j<6; i++,j++)
```

```
for(i=0;i<10;i++)
    for(j=0;j<5;j++)
```

### Dados los siguientes ejemplos determinar qué imprime

La sentencia vacía

```
for(i=0; i<5; i++);
```

```
    cout<< "hola";
```

repite 5 veces la sentencia vacía, vea el `;` después del `)`. Al final imprime hola una vez

Una sentencia simple

```
for(i=0; i<5; i++)
```

```
    cout<< "hola";
```

repite 5 veces la sentencia de asignación externa de salida, vea que no hay `;` después del `)`.

Una sentencia compuesta

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

```
    cout<< "hola";
```

```
    cout<< "buenas tardes";
```

```
}
```

repite 5 veces la sentencia compuesta por dos asignaciones externas de salida, vea las llaves que componen el bloque

Se busca una sentencia compuesta pero se omiten las llaves

```
for(i=0; i<5; i++)
```

```
    cout<< "hola";
```

```
    cout<< "buenas tardes";
```

repite 5 veces la sentencia simple y al final una vez buenas tardes, vea que no hay llaves que componen el bloque

cuántas veces se repiten estas sentencias

```
cin>>numero;
```

```
for(i=0;i<10&&numero>0;i++){
```

```
    cout<< numero;
```



```
cin>>numero;  
};
```

### Ciclo de repeticion while

Se ejecuta [0..N] veces, evalua una expresion antes del inicio del ciclo y repite mientras esta sea verdadera, en general necesita una asignacion previa del dato de la expresion, una evaluacion de la expresion y una nueva asignacion del dato de la expresion para determinar si continua o termina.

```
while(expresion) sentencia;
```

se ejecuta la sentencia (vacía, simple, estructurada o compuesta) mientras la expresion sea verdadera

```
cin>>numero;  
while(numero>0){  
    cout<< numero;  
    cin>>numero;  
};
```

Lo siguiente es correcto? Es equivalente a lo anterior, justifique

```
while(cin>>numero)  
    cout<<numero;
```

El resto de los ejemplos son similares a lo visto con la sentencia for

do while

```
do{  
    lista de sentencias;  
} while (expresion);
```

A diferencia del ciclo while este ciclo se ejecuta al menos una vez ya que la evaluacion de la expresion se hace al final.

### Repeticion: Iteracion versus recursividad

Las repeticiones se pueden implementar con composiciones iterativas o funciones recursivas- Estas funciones se invocan a si mismas. En cada paso resuelven un problema del mismo tipo, pero menor, de este modo se acerca a la finalización al invocar al caso base

Desde el punto de vista de los recursos, por el hecho de las multiples invocaciones utiliza mas memoria. De cualquier modo hay problemas cuyas características son naturalmente recursivas. Retomaremos este tema cuando avancemos en el estudio de funciones

Es un proceso que se basa en su propia definición. Una función puede invocarse a sí misma como parte de los tratamientos de cálculo que necesita para hacer su tarea

Parte de instancias complejas y las define en términos de instancias más simples del mismo problema, llegando a un punto donde las instancias más simples son definidas explícitamente.

Define el problema en términos de un problema más simple de la misma naturaleza.





Debe disminuir el espacio del problema en cada llamada recursiva

Hay una instancia particular que se conoce como caso base o caso degenerado

Divide el problema original en subproblemas más pequeños. Cuando es lo suficientemente chico se resuelve directamente y se combinan soluciones del subproblema hasta que queda resuelto el problema

Tiene:

- ✓ Una ecuación de recurrencia, en función de términos anteriores  $T_n = F(T_{n-1}, T_{n-2}, T_0)$ .
- ✓ Uno o varios términos particulares que no dependen de los anteriores.  $T_i = G(i)$  (base)

### Funcion Factorial

- ✓ Ecuación de recurrencia :  $n! = n * (n-1)!$
- ✓ Condiciones particulares:  $0! = 1$

Instancias que permanecen en memoria:

### Funcion PotenciaNatural

- ✓ Ecuación de recurrencia :  $a^n = a^{(n-1)} * a$  si  $n > 1$
- ✓ Condiciones particulares:  $a^0 = 1$

### Funcion DivisionNatural

Dados dos valores num y den, con  $den \neq 0$  se puede definir el cálculo del cociente y el resto del siguiente modo:

- ✓ Si  $num < den \rightarrow$  el cociente es 0 y el resto num.
- ✓ Si  $num \leq den$  y si c y r son el cociente y resto entre num-den y den  $\rightarrow$  cociente = c+1 y resto r.

### Introduccion al concepto de descomposición

$$SI L = L_1 + L_2$$

Entonces

$$Esfuerzo_{(L)} > Esfuerzo_{(L_1)} + Esfuerzo_{(L_2)}$$

### Modularizacion

Es conveniente, e importante descomponer por varias razones:

- Favorece la comprensión.
- Favorece el trabajo en equipo.
- Favorece el mantenimiento.
- Permite la reusabilidad del código.
- Permite la generalidad de tipos y de criterios
- Permite además separar la lógica de la algoritmia

### Beneficios del uso de acciones y funciones

Una acción o función tiene cuatro propiedades esenciales. Ellas son:

1. Generalidad
2. Ocultamiento de información
3. reusabilidad
4. Modularidad



De estas propiedades, se deducen una serie de beneficios muy importantes para el desarrollo de algoritmos.

1. Dominar la complejidad
2. Evitar repetir código
3. Mejorar la legibilidad
4. Facilitar el mantenimiento
5. Favorecer la corrección
6. Favorecer la reutilización

En el próximo módulo profundizamos estos conceptos



## Estilos de Indentación

A continuación se muestran diferentes estilos, la elección es puramente subjetiva, pero su aplicación al largo de un mismo desarrollo debe ser consistente.

### Estilo K&R – También conocido como "The One True Brace Style"

El libro [K&R1988] usa siempre este estilo, salvo para las definiciones de las funciones, que usa el estilo **BSD/Allman**. Las Secciones principales de **Java** también usan este estilo. Este es el estilo que recomienda y que usa la Cátedra para todas las construcciones.

```
while( SeaVerdad() ) {  
    HacerUnaCosa();  
    HacerOtraCosa();  
}  
HacerUnaUltimaCosaMas();
```

### Estilo BSD/Allman

Microsoft Visual Studio 2005 impone este estilo por defecto. Nuevas secciones de Java usan este estilo. Es un estilo recomendable.

```
while( SeaVerdad() )  
{  
    HacerUnaCosa();  
    HacerOtraCosa();  
}  
HacerUnaUltimaCosaMas();
```

### Estilo Whitesmiths

```
while( SeaVerdad() )  
{1  
    HacerUnaCosa();  
    HacerOtraCosa();  
}  
HacerUnaUltimaCosaMas();
```

### Estilo GNU

```
while( SeaVerdad() )  
{  
    HacerUnaCosa();  
    HacerOtraCosa();  
}  
HacerUnaUltimaCosaMas();
```

### Estilo Pico

```
while( SeaVerdad()  
{ HacerUnaCosa();  
HacerOtraCosa(); }  
HacerUnaUltimaCosaMas();
```