IDENTIFICADORES

Una de las innovaciones que se introdujeron con la aparición de los primeros lenguajes de programación de alto nivel fue la implementación de los denominados "identificadores".
La memoria del computador está compuesta por una serie de celdas que sirven para almacenar información. Cada una de estas celdas se identifica por un número al que se denomina "dirección de memoria". Cada vez que en un programa se tuviera que guardar un dato, se deberá indicar en qué dirección de memoria se va a colocar. Si luego queremos utilizar ese dato para realizar alguna operación, debemos recordar en qué posición de memoria se colocó para poder extraerlo y utilizarlo. Si, por otro lado, en ese programa se quisiera ejecutar una porción de código ubicado en alguna parte de la memoria, tendríamos que hacerlo indicando la dirección de memoria donde se ubica. Como puede apreciar, elaborar un programa, en esas condiciones, sería una tarea muy complicada.
Afortunadamente los lenguajes de programación de alto nivel proporcionan una herramienta que permite que esta tarea sea mucho más simple, esta herramienta se denomina "Identificador".
Un identificador es un nombre, que define el programador, que sirve para denotar ciertos elementos de un programa. Estos elementos pueden ser las denominadas variables, constantes y funciones (elementos que se tratarán más adelante). Cuando se ejecuta el programa, el sistema relaciona estos nombres con alguna dirección de memoria. De este modo, a la hora de programar, ya no se requiere recordar posiciones de memoria sino los nombres dados a estas posiciones de memoria.

Reglas de Formación

La definición de un identificador debe hacerse siguiendo unas reglas que las da el lenguaje de programación, en el caso del C++ las reglas son las siguientes:

- Solo se pueden emplear las letras mayúsculas y minúsculas del alfabeto inglés, esto es: A, B, C, ..., X, Y, Z, y a, b, c, ..., x, y, z.
- También se pueden emplear dígitos decimales: 0, 2, 3, 4, ..., 9. Sin embargo un identificador **NO** puede empezar con un dígito.
- Otro símbolo que se permite utilizar es el _ (signo de subrayar).
- **NO** se pueden emplear letras o símbolos como: ñ, +, &, á, etc.
- Se debe empezar obligatoriamente con una letra o con el signo de subrayar.
- El lenguaje, para efectos de un identificador, considera diferentes las mayúsculas de las minúsculas.

Ejemplo de identificadores válidos son:

actual,x425
No_hay_datos
areaDelTriangulo
_80486,dX_

Ejemplo de identificadores inválidos son (en azul se indican los errores):

año
425x
Nohay+datos
α-β
Tecla£
Tres-Cuatro

Al haber diferencias entre mayúsculas y minúsculas, todos estos identificadores serán considerados diferentes: actual, Actual, ACTUAL, AcTuAl.

TIPOS DE DATOS

El lenguaje C++ define un conjunto de tipos de datos, de los cuales presentamos a continuación algunos de ellos:

ENTEROS		
TIPO	**TAMAÑO**	**RANGO o PRECISIÓN**
char	8 bits	-128 a 127 (Se emplea para representar números pequeños con signo y caracteres de tabla ASCII)
unsigned char	8 bits	0 a 255 (Se emplea para representar números pequeños sin signo y caracteres de la tabla ASCII extendida)
short / short int	16 bits	-32,768 a 32,767
unsigned short	16 bits	0 a 65,535
int	32 bits	2,147'483,648 a 2,147'483,647
unsigned int	32 bits	0 a 4,294'967,295
long	32 bits	-2,147'483,648 a 2,147'483,647(Este tipo de dato quedó en desuso cuando el tipo int pasó a tener 32 bits)
long long	64 bits	± 9,223,372,036,854,775,808
DE PUNTO FLOTANTE		
TIPO	**TAMAÑO**	**RANGO o PRECISIÓN**
float	32 bits	3.4×10^{-38} a 3.4×10^{38}(11 dígitos significativos.)
double	64 bits	1.7×10^{-305} a 1.7×10^{308}(15 dígitos significativos.)
LÓGICOS		
TIPO	**TAMAÑO**	**RANGO o PRECISIÓN**
bool	8 bits	true (1) y false (0).

EXPRESIONES

Una expresión es un conjunto de valores constantes, variables, constantes y funciones unidas por un operador, se agrupan con la finalidad de obtener un resultado.

A continuación, definimos estos términos:

VALORES CONSTANTES

Los valores constantes están referidos a la forma cómo podemos escribir un valor según el tipo de dato al que queremos referirnos. A continuación, se muestra estas formas:
Un entero se debe escribir como: 357, 1243, 38, etc.
Un valor entero octal (en base 8) como: 0473, 0478
Un valor entero hexadecimal (en base 16) como: 0x4AB, 0x4AB
Los valores de punto flotante como: 18.74, 3.94e-3, 1.6523E85, etc.
Un valor de tipo char como: 97 o 'a' (ambos valores son lo mismo, el caracter a va entre apóstrofos)
También se pueden representar los caracteres así:

'\a' carácter de alarma (Bell)
'\n' nueva línea (line feed)
'\t' tabulador horizontal
'\\' diagonal invertida (back slash)
'\?' signo de interrogación
'\'' apóstrofo
'\"' comillas

Un valor lógico se representa por los identificadores true y false que contienen los valores 1 y 0 respectivamente.

VARIABLES Y CONSTANTES

VARIABLES:

Una variable es un identificador relacionado a una posición de memoria donde se pueden colocar datos. Al definirse como una variable, el contenido de esa posición de memoria podrá cambiar o ser modificado a lo largo del programa.
Una variable se define colocando el tipo de dato seguido por los identificadores que desee utilizar. Ejemplos de definición de variables se muestran a continuación:

```
int actual, cont;
char n, rango;
doubleraiz, valor;
```

Las variables no se inicializan por defecto, por lo que, si se desea que éstas tengan un valor definido desde su declaración, se deberá realizar lo que a continuación se indica:

```
int actual=0, cont=24;
char p = 57, n = 'a', rango = '\x18';
double raiz = 2.21, valor = 217.56;
```

Se recomienda, y es una buena práctica de programación, que los identificadores que definan variables debieran empezar siempre con una minúscula, el resto de caracteres deben ser colocados en el formato "Camel case", esto es, si el identificador estuviera formado por varias palabras, la primera letra de cada una excepto la primera debe estar en mayúsculas, el resto estarán en minúsculas. Por ejemplo:

areaDelTriangulo, promedioPonderado, sumaDeNotas, etc.

CONSTANTES:

Una constante es básicamente lo mismo que una variable, con la única diferencia que la constante debe ser inicializada en su definición y este valor no se podrá modificar a lo largo del programa. Se definen de la siguiente manera:

```
const double PI=3.1415926;
constint R = 33;
const char L = 'A';
```

true y false son constantes de tipo bool.

CONSTANTES SIMBÓLICAS:

Los "Nombres simbólicos" o "Constantes simbólicas" son elementos dentro del lenguaje C++ que no pueden considerarse dentro de la misma categoría que las variables o constantes propiamente dichas. Una constante simbólica es un texto que va a ser buscado y reemplazado por otro en el texto del programa, antes de iniciarse el proceso de compilación. Este proceso se denomina pre-compilación. Una constante simbólica se define mediante la cláusula *#define* de la siguiente manera:

```
#define TEXTO_A_BUSCAR texto_de_reemplazo.
```

Se recomienda que los identificadores que definan constantes o constantes simbólicas deban estar escritos en su totalidad en mayúsculas.

El "TEXTO_A_BUSCAR" debe estar formado por una sola palabra, mientras que el "texto_de_reemplazo" puede estar formado por varias palabras, pero todo debe estar en una sola línea.

```
#define ENTERO      int
#define REAL        double
#define PROGRAMA    main
#define INICIO      {
#define FIN         }

ENTERO PROGRAMA () INICIO
    ENTERO n=50;
    REAL f, g;
    …
FIN
```

OPERADORES

Los operadores son elementos de enlace en una expresión y nos sirven para realizar operaciones y obtener un resultado.

Los operadores se pueden considerar como unarios y binarios, se dice que un operador es unario cuando se aplica a un solo elemento, será binario cuando se aplica sobre dos elementos. A continuación, se presentan los distintos operadores que se encuentran en el lenguaje C++.

OPERADORES ARITMÉTICOS:

OPERADOR	SIGNIFICADO
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

El resultado de una operación depende de los operandos que la componen, si uno de ellos es de punto flotante, el resultado lo será también, de lo contrario será un número entero.
Por ejemplo:

3/4 dará como resultado cero (0)
3.0/4 dará como resultado 0.75
3/4.0 dará como resultado 0.75

OPERADORES DE ASIGNACIÓN:

Una operación de asignación es una operación por medio de la cual se coloca información en una variable. La asignación se realiza mediante el signo =, de la siguiente manera:

Variable = Expresión;

Ejemplos de asignación:

```
int a, b, c;
a = 5;
b = 12;
c = 3*a + b;
```

Asignación múltiple:

Un operador de asignación, al igual que los otros operadores, al aplicarse en una expresión devolverá un valor. En el caso de la asignación, el valor devuelto será el valor asignado. Por esta razón la asignación puede concatenarse, de modo que se asigne un valor a muchas variables en una sola expresión, como se indica a continuación:

a = b = c = d = 5;

En el ejemplo el valor de 5 se asigna a la variable d, pero como = es un operador, el resultado (5) sirve para asignarlo a la variable c, que a su vez se le asignará la variable b y luego a la variable a.

Otros operadores de asignación:

+=	-=	*=	/=	%=

Estos operadores sirven para abreviar las operaciones de asignación en las que la variable a la que se le está asignado un valor aparece en la expresión de la derecha. Algunos ejemplos se muestran a continuación:

Expresión	Equivale a hacer
a += 5;	a = a + 5;
b -= 2	b = b - 2;
c *= 11;	c = c * 11;
d /= 7;	d = d / 7;
e %= 8;	e = e % 8;

CONVERSIÓN DE TIPOS EN LA ASIGNACIÓN:

Se debe tener mucho cuidado cuando se manejan expresiones que devolverán valores de un tipo diferente al de la variable que recibe el resultado. El compilador no avisa si la conversión se va realizar de la manera que uno puede esperar. El ejemplo siguiente muestra una de estas situaciones.

```
int a = 527;
char c;
c = a
```

Si ejecutamos el código, veremos que el valor asignado a la variable c es 15. Luego, ¿cómo un valor como 527 se pudo convertir en 15?
La respuesta es simple, el valor hexadecimal de 527 es 00 00 02 $0F_h$ ya que un entero se almacena en 4 bytes. Al pasar este valor a una variable como c, que es de tipo char y por lo tanto se almacenará en un byte, los tres bytes más significativos del valor de la variable a se perderán, por lo que solo se almacenará en la variable c el byte menos significativo, que en este caso es $0F_h$ y que no es otra cosa que el valor 15.

Conversión explícita de tipos:

Cuando desarrollamos expresiones, sobre todo cuando operamos números enteros, muchas veces vamos a encontrar situaciones en las que recibimos respuestas no esperadas, por ejemplo, la siguiente:

```
int a = 8, b = 3, c = 15;
double f;
f = b/a * c;
```

El valor asignado a la variable f será cero. La razón se da por las propiedades de los operadores, primero se evalúa la expresión a/b, como las dos variables son enteras, el resultado de dividir 3/8 es precisamente cero. Al multiplicarlo luego por el valor de la variable c, el resultado permanecerá en cero.
Una forma de corregir esto podría ser como sigue:

```
f = 1.0 * b/a * c;
```

Al multiplicar primero 1.0 (valor de punto flotante) por el contenido de la variable b se obtiene el valor 3.0 (valor de punto flotante), luego al dividir este valor entre el valor de la variable a (en este caso 8) se obtendrá el valor de 0.375 y luego al multiplicarse por el valor de la variable c (15) se obtendría el valor esperado de 5.625.
Sin embargo, existe otra forma de hacerlo, mediante lo que se denomina "conversión explícita de tipos" o también operación "type cast".

La operación type cast consiste en escribir un tipo de dato entre paréntesis al lado de una variable dentro de una expresión. Al hacerlo el valor de la variable será convertido temporalmente al tipo de dato escrito. La expresión siguiente muestra este detalle:

```
f = (double)b/a*c;
```

El valor de la variable b es convertido de 3 a 3.0 por la operación type cast, luego el valor de final de la expresión será el esperado.

OPERADORES UNARIOS DE INCREMENTO Y DECREMENTO:

Se denominan operadores unarios porque se aplican a un solo operando. Un ejemplo de operador unario se aprecia en la expresión: a * -b, el operador - se aplica sólo en la variable b y en este caso sirve para cambiar el signo de esa variable.
Los operadores de los que hablamos en este acápite son:

```
++ incrementa el operando en 1
-- decrementa el operando en 1
```

Estos operadores pueden emplearse como prefijo (++a o --a) o como sufijo (a++ o a--), sin embargo, no será lo mismo que utilicemos el operador en una instrucción simple que en una compuesta. Esto se explicará de la siguiente manera:

Expresión simple:

```
int a = 3;
++a;            // la variable a se incrementa a 4

int a = 3;
a++;            // la variable a se incrementa a 4
```

Expresión compuesta:

```
int a = 3,b;
b = ++a;        // la variable a se incrementa a 4 y b recibe 4

int a = 3;
b = a++;        // la variable a se incrementa a 4, pero b recibe 3
```

En líneas generales, en una expresión compuesta, si el operador se coloca como prefijo, primero se aplica el operador sobre la variable y luego se evalúa la expresión. Si se usa como sufijo, primero se evalúa la expresión y luego se aplica el operador sobre la variable.

OPERADORES DE RELACIÓN:

Los operadores de relación se aplican, como su nombre lo indica, para relacionar dos valores. El resultado de esta operación será uno (true o 1) si la relación es correcta y cero (false u 0) sino lo es.

Operador	Significado
==	Igual
!=	Diferente
>	Mayor que
<	Menor que
>=	Mayor o igual
<=	Menor o igual

OPERADORES LÓGICOS:

Los operadores de lógicos se aplican sobre expresiones lógicas. Una expresión lógica, en C++, aquella que devuelva un valor falso (false o 0) o un valor verdadero (valor diferente de cero, true o 1). El resultado de esta operación será uno (true o 1) si la operación es correcta y cero (false u 0) sino lo es.

Operador (*)	Significado
! o not	Not (negación)
&& o and	And (conjunción)
|| u or	Or (disyunción)

En los antiguos estándares de C++ solo se podía usar los símbolos: !, && y ||, en los más modernos se pueden emplear también las palabras: not, and u or respectivamente.

OPERADOR CONDICIONAL:

El operador condicional sirve para decidir, entre dos expresiones, qué valor se le asignación a una variable. Este operador consta de dos símbolos: "?" y ":" y se forma de la siguiente manera:

$$\text{Variable} = \text{expr}_1 ? \text{expr}_2 : \text{expre}_3$$

Al evaluarse, si la expr_1, da un valor diferente de cero (verdadero) entonces la expresión condicional devuelve el valor de la expr_2, de lo contrario devuelve el valor de la expr_3.
Por ejemplo:

```
b = a > c ? a + 1 : a - 1;
b = a - 3 ? a : c ;
```

PRECEDENCIA DE LOS OPERADORES:

Una expresión por lo general emplea más de un operador, si ese es el caso, entonces debemos tener en cuenta en qué orden se aplicarán los operadores, esto debido a que estos no se evaluarán necesariamente en el orden en se ubican en la expresión.
A continuación, se presenta una tabla en la que se muestra el orden de evaluación o la precedencia de los operadores.

Nivel	Operador
1°	( )
2°	!, not, ++, --, +, - (+, - como operadores unarios)
3°	*, /, %
4°	+, -
5°	<, <=, >, >=
6°	==, !=
7°	&&, and
8°	||, or
9°	? :
10°	=, +=, -=, *=, /=

En el caso que dos operadores del mismo nivel estén presentes en una expresión, se evaluará primero el que esté más a la izquierda.

Ejemplo de uso de operadores:
Leer dos datos y determinar cuál es mayor sin usar "if":

```
//Versión A: Usando operadores de relación
mayor = a*(a>b) + b*(b>a);
cout << "1) El mayor es: " << mayor << endl;

//Versión B: Usandoel operador condicional
mayor = a>b ? a : b;
cout << "2) El mayor es: " << mayor << endl;
```

Cambiar a mayúscula un caracter si es una letra minúscula

```
//Versión A: Usando operadores de relación
carCambiado = car - ('a'-'A')*(car>='a' && car<='z');
o
carCambiado = car - ('a'-'A')*(car>='a' and car<='z');
cout << "1) El caracter cambiado es: " << carCambiado) << endl;

//Versión B: Usando el operador condicional
carCambiado = car - (car>='a' && car<='z' ? 'a'-'A' : 0);
o
carCambiado = car - (car>='a' and car<='z' ? 'a'-'A' : 0);
cout << "2) El caracter cambiado es: " << carCambiado) << endl;
```

VARIABLES PUNTEROS

Las variables puntero o simplemente punteros son, como su nombre lo indica, variables y por lo tanto sus propiedades y comportamiento son los mismos que cualquier variable, esto es, se les puede asignar valores y modificarlos, se deben inicializar porque el sistema no lo hace automáticamente, etc.
Sin embargo, una variable puntero tiene características adicionales que describiremos a continuación.

La manera cómo se declara tiene la forma:

TIPO *nombre.

Donde:

TIPO: Puede ser cualquier tipo de dato de los que se mencionan anteriormente.
*: Identifica a la variable como un puntero.
nombre: Es el nombre que daremos al identificador. Se le puede nombrar solo o anteponiéndole el asterisco.

Por ejemplo:

```
int *ptDatos, *ptValores;        //Se les denomina punteros a enteros
double *r, *s, *t;               // Se les denomina punteros a double
char *nombre, *apellido;         // Se les denomina punteros a caracteres.
```

Observe que cada variable debe tener antepuesto el *, de no colocarse la variable no será considerada como un puntero. Así, por ejemplo:

```
int *a, b, *c;
```

Las variables a y c serán definidas como punteros, pero b no, solo será una variable entera común.

La particularidad de una variable de tipo puntero es que almacena una dirección de memoria. En programación cuando mencionamos "dirección de memoria", nos estamos refiriendo a un valor entero que identifica el lugar en la memoria del computador donde se coloca una variable.

Asignación de valores a un puntero

Una variable de tipo puntero, como ya se indicó, se comporta de manera similar a la de una variable común, sin embargo, tiene características que la diferencian de las demás, una de ellas es que para poder utilizarla debemos inicializarla, de lo contrario el programa se caerá, se interrumpirá abruptamente o simplemente dará resultados incoherentes.

Para explicar lo anterior pensemos primero en una variable común, observe las siguientes líneas de un programa:

```
int a, b;
a = b + 1;
cout<<"A ="<< a<< endl; //Qué valor cree que imprimirá para "a"?
```

La respuesta es que imprimirá un valor incoherente, esto se debe a que el lenguaje C/C++ no inicializa las variables como lo hacen otros lenguajes de programación, por lo tanto, el sistema asigna a las variables "a" y "b" valores indeterminados en la primera línea del programa. En la segunda, a ese valor indeterminado de la variable "b" le suma 1, que finalmente es asignado a la variable "a".

De la misma manera, trabajar con una variable puntero no inicializada ocasionará resultados incoherentes con la gravedad que estaremos manejando una dirección de memoria indeterminada (que no sabremos qué es lo que contiene), por lo tanto, debemos inicializarla de modo que la dirección que contenga la variable puntero

corresponda a un espacio en el que podamos trabajar sin problemas, sin interferir con otros espacios que podrían estar siendo utilizados por el sistema.
Tenemos dos formas de inicializar una variable de tipo puntero, una se denomina "asignación estática" la otra "asignación dinámica".

Asignación estática de memoria

Esta forma se da cuando a una variable puntero se le asigna la dirección de memoria de una variable común, por ejemplo:

```
int a = 31;
int *ptInt; //Puntero a entero

ptInt = &a;  // Donde & se denomina operador de dirección,
             // cuando se aplica a una variable devuelve su
             // dirección de memoria */
```

En la última línea se realiza la inicialización del puntero, allí se está asignando al puntero ptInt la dirección de memoria de la variable "a".

Asignación dinámica de memoria

La asignación dinámica se realiza mediante el operador "new", esta función entrega una dirección de memoria donde se puede colocar un valor, garantizándonos que ese espacio de memoria no está siendo utilizado por el programa que estamos desarrollando ni por algún otro programa que en ese momento se esté ejecutando en nuestro computador. Además, una vez que se asigne la dirección, el espacio de memoria referido por esa dirección no podrá ser utilizado por ningún otro programa, ni la dirección será entregada si se utiliza otra vez el operador new, esto hasta que explícitamente se libere este espacio. Al ser un operador, está definido en el núcleo del compilador por lo que no se requiere incluir ninguna biblioteca de funciones.
La forma cómo se utiliza esta función de describe a continuación:

puntero = new tipo;

Donde:

puntero: es la variable de tipo puntero
tipo: es el tipo de dato con el que se definió la variable puntero

Por ejemplo:

```
int main(){
  int *ptInt;
  double *ptDbl;
  char *n;

  ptInt = new int;
  ptDbl = new double;
  n = new char;

  …
}
```

Asignación con valor nulo

Un puntero puede inicializarse también con un valor nulo, que corresponde a una dirección cero, esta operación se realiza mediante el identificador "nullptr" definida también en el núcleo del compilador. La forma como se utiliza es la siguiente:

```
int *ptInt;
ptInt = nullptr;
```

Indirección o desreferencia de un puntero

Otra característica particular de una variable de tipo puntero es que mediante el puntero podemos manipular el valor contenido en la dirección de memoria que se le asignó al puntero. Esta característica se emplea mediante el operador *, que aplicado a un puntero se le denomina operador de indirección u operador de desreferencia. La forma de utilizar este operador es como se muestra a continuación:

```
int a = 31;
int *ptInt;
double *ptDbl;

ptInt = &a;
ptDbl = new double;

*ptInt = 22;       // El sistema toma el valor que contiene la
                   // variable ptInt (dirección de memoria de la
                   // variable a) se dirige a esa dirección de
                   // memoria y coloca en ella el valor de 22.
                   // En este caso por medio de esta operación
                   // se modificó el valor de la variable a */
*ptDbl = 357.81;   // El sistema toma el valor que contiene la
                   // variable ptDbl (dirección de memoria
                   // asignada por malloc) se dirige a esa
                   // dirección de memoria y coloca en ella el
                   // valor de 357.81 */
```

Un puntero inicializado con valor nullptr no puede desreferenciarse, de hacerlo se producirá un error que interrumpirá la ejecución del programa.