Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

FUNDAMENTOS DE PROGRAMACIÓN

Asignatura correspondiente al plan de estudios de la carrera de Ingeniería Informática

UNIDAD 7
EXPRESIONES, OPERADORES y ESTRUCTURAS de CONTROL en C++

2016

UNIDAD 7

Expresiones, operadores y estructuras de control en C++

Resumen de Conceptos

Introducción

Hemos estudiado las estructuras de control desde el punto de vista algorítmico. Ya conocemos entonces para que sirve cada una de estas estructuras (secuenciales, condicionales y repetitivas) y qué tipo de problemas algorítmicos podemos resolver con ellas. En esta unidad nos proponemos desarrollar en qué forma se implementan las estructuras de control en el lenguaje C++.

Básicamente aprenderemos las equivalencias entre la sintaxis del pseudocódigo y la sintaxis del lenguaje C++. Usted podrá apreciar que estas equivalencias son muy simples y en muchos casos se limitan a casi una traducción entre el castellano (pseudocódigo) y el inglés (C++). Por ejemplo, la estructura que conocemos como **Mientras**-**hacer** se traducirá en **while** (que significa “mientras” en inglés).

Antes de introducirnos en las estructuras de control de C++, estudiaremos la importancia de los operadores, su sintaxis y su jerarquía. Con ellos podremos construir expresiones esenciales para el planteo de las estructuras de control.

Las actividades de esta unidad están diseñadas para trabajar con cualquier entorno de desarrollo de C++ estándar, en modo consola. La mayor parte del tiempo de estudio consistirá en actividades de programación. ¡No hay mejor forma de aprender la sintaxis de un lenguaje de programación que programando!

Expresiones

Toda expresión consiste en *un conjunto de operandos ligados por operadores.* Utilizaremos expresiones en C++ habitualmente para efectuar cálculos, relaciones, asignaciones, etc. Para plantear expresiones en C++ debemos conocer los numerosos operadores que posee este lenguaje.

Operadores

Operadores Aritméticos

Como su nombre lo indica, los operadores aritméticos nos permiten efectuar cálculos aritméticos. La *jerarquía o precedencia* de estos operadores es idéntica a la que empleamos en el álgebra de números. Esta jerarquía se puede alterar empleando paréntesis.

Operador	En tipos Enteros	En tipos Reales	Ejemplo
+	símbolo + unario	símbolo + unario	+3
-	símbolo – unario	símbolo – unario	-5
+	suma	suma	a+b
-	resta	resta	a-b
*****	producto	producto	3*x
/	división entera	división en punto flotante	f/2
%	resto de la división entera	<no aplicable>	x%2
++	pre-incremento	pre-incremento	++i
--	pre-decremento	pre-decremento	--i
++	post-incremento	post-incremento	i++
--	post-decremento	post-decremento	i--

Se debe tener en cuenta que las operaciones algebraicas tienen asociatividad de izquierda a derecha pero respetando la precedencia de los operadores aritméticos.

Ejemplos:

10 + 7 – 4 arroja como resultado **13**. Se resuelve: **(10 + 7) – 4** porque ante igual prioridad de operadores se asocia de izquierda a derecha.

6 + 2 * 9 arroja como resultado **24** . Se resuelve: **6 + (2 * 9)** por que el operador producto tiene prioridad (precede) al operador de la suma.

21 / 4 arroja **5** (entero)

20.0 / 4.0 arroja **5.0** (punto flotante)

21 % 4 arroja **1** (entero)

++x incrementa en **1** el valor de **x**

x++ toma el valor de **x** y luego lo incrementa a en **1**

--x decrementa en **1** el valor de **x**

x-- pos decrementa en **1** el valor de **x**

Nota: obsérvese en los 2 ejemplos siguientes la diferencia entre los operadores de incremento y decremento cuando preceden a una variable y cuando la suceden.

```
int  n=2;
cout << n++ ;  /* Se visualiza un 2. C++ envía el contenido de n a la salida a través de cout y luego incrementa en 1 a n */
```

```
int  n=2;
cout << ++n ;  /* Se visualiza un 3. C++ incrementa en 1 a n y luego muestra el nuevo valor de n */
```

En ambos casos, la variable n se ha incrementado en una unidad al finalizar la segunda instrucción. Pero en esta instrucción hay dos acciones: incrementar n y mostrar n en pantalla. ¿Cual de las dos operaciones se realiza primero? El pre incremento actualiza la variable y arroja como resultado ese nuevo valor, que es lo que en este caso tomará cout. El post-incremento en cambio, a pesar de incrementar el valor de la variable, entrega como resultado (a cout en este caso) el valor viejo (sin incrementar) de la variable.

Operadores de asignación

El operador **=** permite asignar el resultado de la expresión de la derecha a la variable de la izquierda.

```
x = 130;
```

Debemos observar que este operador es asociativo por la derecha, por lo cual podemos hacer asignaciones múltiples o simultáneas.

```
a = b = c = 30;
```

Esto permite asignar simultáneamente a las tres variables el valor 30. El compilador realiza la asociación del modo siguiente: **a = (b = (c = 30))** . Debe observarse que para C++ la proposición **c = 30** tiene doble sentido: 1) se trata de una asignación y 2) se trata de una expresión que arroja el valor de **c**, luego de realizar la asignación (30 en este caso).

```
cout << (n=5) ; /* asigna 5 a la variable n y visualiza 5 (resultado de la expresión)*/
```

Operadores relativos de asignación

C++ dispone de operadores relativos, que permiten hacer más eficiente el código ejecutable resultante de la compilación

Operador	Asignación abreviada	Asignación no abreviada
+ =	x + = y	x = x + y
- =	x - = y	x = x - y
*** =**	x * = y	x = x * y
/ =	x / = y	x = x / y
%=	x % = y	x = x % y

Operadores Relacionales

C++ dispone de operadores relacionales, cuyo concepto es idéntico al que poseen en el álgebra de números. Su simbología también es similar, a excepción de los operadores *igual que* y *distinto que* como veremos a continuación.

Estos operadores nos permitirán plantear expresiones relacionales, las cuales al ser evaluadas arrojarán un valor de verdad: **verdadero** o **falso**. En C++ un valor de verdad corresponde a un valor de tipo **bool**. Los dos posibles valores para este tipo son **true** (verdadero) y **false** (false).

Nota: Opcionalmente, si se intenta utilizar un valor numérico como valor de verdad, C++ lo considerará falso si es igual a 0, verdadero en cualquier otro caso.

Los operadores relacionales permiten relacionar (comprara) operandos de tipos compatibles

Operador	Significado	Ejemplo
==	Igual que	a == b
!=	Distinto que	a != b
<	Menor que	a < b
>	Mayor que	a > b
<=	Menor o igual que	a <= b
>=	Mayor o igual que	a >= b

Utilizaremos expresiones relacionales y lógicas para varias estructuras de control de C++ que nos permitirán plantear decisiones en el programa.

Los operadores relacionales se asocian de izquierda a derecha y tienen menor prioridad que los operadores aritméticos por lo tanto una expresión del tipo:

`a+b<10*c` equivale a **`(a+b)<(10*c)`**

Es posible asignar el resultado de una expresión relacional a una variable:

```
bool m=(12+3<=10);      // asigna false a la variable m
```

Operadores Lógicos

Los operadores lógicos propiamente dichos de C++ son la conjunción o and (**&&**), la disyunción u or (**||**) y la negación o not (**!**). Conceptualmente funcionan de igual forma que en la lógica algebraica.
Recordemos que la conjunción (**&&**) arroja por resultado verdadero solo si ambos operandos son verdaderos; la disyunción (**||**) solo es falsa si ambos operandos son falsos; y la negación (**!**) es un operador unario que invierte el valor de verdad del operador afectado.

Operador	Significado	Ejemplo
!	Negación (no)	! (a <= b)
&&	Conjunción (y)	(a < b) && (n==100)
||	Disyunción (o)	(x == 10) || (a != c)

Evaluación en cortocircuito

C++ dispone la evaluación de una expresión lógica de izquierda a derecha. Si el operando de la izquierda es suficiente para determinar el resultado de la proposición, no se evalúa el operando de la derecha.
Por ejemplo: **(6 < 3) && (z == 4)** el operando **z == 4** no llegará a evaluarse pues **6 < 3** ya decidió el resultado (falso) de toda la proposición.

Operadores de Manipulación de Bits

Estos operadores tratan a la información de cada operando como un conjunto de bits. Solo se pueden emplear con operandos de tipo discreto y no con números de punto flotante (reales).

Operador	Significado	Ejemplo
&	Y (and)	1110 0011 **&**1010 0101 1010 0001
|	O (or)	1110 0011 **|** 1010 0101 1110 0111
^	O excluyente (xor)	1110 0011 **^** 1010 0101 0100 0110
~	No (not)	**~**1110 0011 0001 1100
>>	Desplazam. de bits a izquierda	1110 0011**<<2** 1000 1100
<<	Desplazam. de bits a izquierda	1110 0011**>>2** 0011 1000

Nota 1: Se debe advertir aquí la diferencia entre los operadores && y &, y entreo los operadores | y ||. En general, si el programador introduce por error uno donde corresponde el otro, el programa cambia su significado, pero no deja de ser válido, por lo que no será detectado como error de compilación.

Nota 2: En c++, los operadores varían su comportamiento de acuerdo al tipo de datos sobre el que se lo aplica. Aquí se describe el comportamiento sobre los tipos de datos básicas, y por ello el más habitual. Por ejemplo, los operadores **>>** y **<<** tienen otra función muy diferente cuando se los utiliza con **cin** y **cout** respectivamente.

Otros Operadores

C++ dispone de otros operadores que describiremos más adelante conforme avancemos en el desarrollo de nuevos temas.

Precedencia de Operadores en C++

La precedencia o prioridad de un operador determina el orden de aplicación de los operadores de una expresión. En la tabla siguiente se indican los grupos de operadores según orden de prioridad.

Proiridad y categoría	Operador	Función o significado	Asociati-vidad
1. (Prioridad más alta)	**( )** **[ ]** **→** **::** **.**	Llamada a función Subíndice de arreglos Selector indirecto de miembro Selector de ámbito de resolución Selector directo de miembro	I-D
2. Unarios	**!** **~** **+** **-** **++** **--** **&** ***** **sizeof** **new** **delete**	Negación lógica Complemento a uno (bits) Más (unario) Menos (unario) Incremento Decremento Dirección Indirección Tamaño del operando en bytes Alocación dinámica en memoria Eliminación dinámica	D-I
3. Acceso a miembros	***** **→**	Lee o modifica el valor apuntado Accede a un miembro de un objeto apuntado	I-D
4. Multiplicativos	***** **/** **%**	Multiplicación División entera o flotante Resto de la división entera (módulo)	I-D
5. Aditivos	**+** **-**	Más binario Menos binario	I-D

6. Desplazamiento	**>>** **<<**	Desplazamiento a la derecha Desplazamiento a la Izquierda	I-D
7. Relacional	**<** **<=** **>** **>=**	Menor que Menor o igual que Mayor que Mayor o igual que	I-D
8. Igualdad	**==** **!=**	Igual que Distinto que	I-D
9.	**&**	And (manipulación de bits)	I-D
10.	**^**	Xor (manipulación de bits)	I-D
11.	**|**	Or (manipulación de bits)	I-D
12.	**&&**	Conjunción lógica and	I-D
13.	**||**	Disyunción lógica or	I-D
14. Condicional	**?:**	a ? x : y (significa: if a then b, else c)	D-I
15. Asignación	**=** ***=** **/=** **%=** **+=** **-=** **&=** **^=** **| =** **<<=** **>>=**		D-I
16. Coma (prioridad más baja)	**,**	Evaluación múltiple	

Si en una expresión aparecen operadores consecutivos de igual prioridad debe considerarse la forma de asociarlos para resolver la expresión. Por ejemplo en aritmética: ante la presencia de operadores de suma (+) y resta (-), los cuales tienen igual prioridad, C++ asocia de izquierda a derecha como corresponde al álgebra de números. La expresión **`a+b-c`** se resuelve: **`(a+b)-c`**

Deben tenerse en cuenta las reglas siguientes para el planteo de expresiones:

- Todos los operadores de un mismo grupo tienen igual prioridad y asociatividad.
- Si dos operadores se aplican a un operando, se aplica antes el de mayor prioridad
- Asociatividad I-D significa que primero se aplica el operador de la izquierda y luego el siguiente hacia la derecha. Asociatividad D-I significa hacer lo contrario.
- La precedencia o prioridad de operadores puede alterarse con los paréntesis, quienes tienen máxima prioridad.

Funciones de bibliotecas estándar de C++

Además del conjunto de operadores presentados, C++ dispone de una interesante variedad de funciones en sus bibliotecas estándar. Ellas nos ahorran un importante esfuerzo a la hora de efectuar ciertos cálculos. El programador C++ solo debe considerar el tipo de argumento requerido, el tipo de resultado que devuelve y el nombre del archivo de inclusión donde se halla el prototipo de la función para indicarlo en la sentencia #**include** correspondiente. Más adelante aprenderemos a crear nuestras propias funciones.

Ejemplo

```
#include <iostream>
#include <cmath>
//cmath incluye la función sin(x) y la constante M_PI
int main ( )  {
  int ang_gra;
  cout << "Ingrese un ángulo en grados:";
  cin >> ang_gra;
  float  ang_rad = ang*M_PI/180; // pasa a radianes el ángulo
  cout << "El seno del ángulo es: " << sin(ang_rad) << endl;
  return 0;
}
```

Estructuras de Control

C++ dispone de varias estructuras para controlar la lógica de ejecución de los programas. Estas estructuras pueden clasificarse en: *condicionales o selectivas, repetitivas y de interrupción o salto no condicional*

Tipo de estructura	Sentencia C++

Repetitiva	`while`
	`do-while`
	`for`

Condicional o selectiva	`if-else` `switch`
Salto no condicional o interrupción	`break` `continue`

while

Equivale a la estructura de pseudocódigo Mientras-Hacer. Las acciones abarcadas por esta estructura se ejecutan repetidamente mientras la expresión lógica sea verdadera. La condición se evalúa antes de cada iteración.

Sintaxis	Ejemplo

`while (expresión lógica) {`
`  acciones`
`}`
	`int a=0;`
	`while (a<100) {`
	`  cout << a << endl;`
	`  a++;`
	`}`

Notar que la expresión lógica de control en esta y también en todas las demás estructuras deben ir siempre entre paréntesis.

do-while

Las acciones abarcadas por esta estructura se ejecutan repetidamente hasta que la expresión lógica arroje el resultado falso. La condición se evalúa luego de cada iteración. Es similar a la estructura de pseudocódigo Repetir-HastaQue, pero en este caso la condición funciona de forma inversa.

Sintaxis	Ejemplo

`do {`
`  acciones`
`} while (expresión lógica);`
	`int b=0;`
	`do {`
	`  b++ ;`
	`  cout << b << endl;`
	`} while (b<100);`

Sentencia for

Las acciones abarcadas por esta estructura se ejecutan repetidamente mientras que la exp2 arroje verdadero; exp1 hace de expresión de inicialización y se ejecuta una única vez al comienzo; y exp3 se ejecuta al final del grupo de acciones en cada iteración. Por ello, generalmente se exp1 para incializar un contador, exp2 para compararlo con el número de iteraciones deseadas y exp3 para incrementarlo. Utilizada de esta forma, la estructura actúa de forma similar a la estructura Para presentada en la etapa de pseudocódigo. Sin embargo, en C++ se puede utilizar de forma diferente en muchos otros casos.

Sintaxis	Ejemplo

`for (exp1; exp2; exp3) {`
`  acciones`
`}`
	`for (int a=0; a<100; a++) {`
	`  cout << a<< "\n" ;`
	`}`

Notar que en el ejemplo exp1 no solo inicializa, sino que también define el contador. La definición es opcional, y en ese caso el ámbito de validés de la variable es el bloque de acciones que abarca esta estructura.

Además, C++ permite en la sentencia for, a través del operador coma (,), realizar más de una instrucción en las expresiones involucradas. Esto puede ser útil para, por ejemplo, utilizar dos contadores que avancen de forma diferente en cada iteración.

Ejemplo

```
for (int i=0, j=10; i < 10 ; i++, j--) {
   cout << i << " " << j << endl;
}
```

If-else

Equivale a la construcción Si-Entonces-Sino del pseudocódigo. Se evalúa una expresión lógica planteada a continuación del if y si es verdadera se realizan las acciones indicadas a continuación; mientras que si la expresión lógica es falsa se realizan las acciones a continuación del else. En esta estructura la salida por falso puede omitirse, y en tal caso, si la expresión arroja falso no se ejecutará acción alguna.

Sintaxis	Ejemplo

`if (expresión lógica) {`
`  acción 1;`
`} else {`
`  acción 2;`
`}`
	`int c;`
	`cin>>c;`
	`if ( c%2==0 ) {`
	`  cout << "Es par" << endl;`
	`} else {`
	`  cout << "Es impar" << endl;`
	`}`

switch

Esta sentencia permite efectuar una selección entre múltiples opciones en base al valor de una variable de control que nos permite gobernar la estructura. Es similar a la estructura Según que empleamos en pseudocódigo. La variable de control debe ser de algún tipo de dato numérico y entero.

Sintaxis	Ejemplo

`switch (expresión) {`
`  case valor1:`
`  acción_1;`
`  break;`
`  case valor2:`
`  acción_2;`
`  break;`
`  case valor3:`
`  acción_3;`
`  break;`
`  .....`
`  default:`
`  acción_m;`
`}`
	`cin >> m`
	`switch ( m ) {`
	`  case '1':`
	`  m++`
	`  break;`
	`  case '2':`
	`  m=2*m;`
	`  break;`
	`  case '3´:`
	`  m = m / 2;`
	`  break;`
	`  default :`
	`  m = 100;`
	`}`

La acción propuesta a continuación de default se ejecutará si el valor de la expresión de control no coincide con ninguno de los valores propuestos en la lista. La opción default es opcional; si no se indica y el valor de la expresión no aparece en la lista propuesta, ninguna acción será ejecutada.

Notar que luego de cada grupo de acciones pertenecientes a un posible valor para la variable de control, se debe colocar la instrucción "break;". Si no se lo hace, luego de ejecutar las instrucciones de ese grupo, el programa continuará ejecutando los del siguiente. Esto puede usarse para asignar un mismo grupo de acciones a dos o más posibles valores.

Ejemplo

```
cin >> m
switch ( m ) {
   case '1':
       cout << "m es 1" << endl;
       break;
   case '2':
   case '3´:
       cout << "m es 2 o 3" << endl;
       break;
   default :
       cout << "no está entre 1 y 3" << endl;
}
```

break y continue

Ambas sentencias interrumpen la ejecución del grupo de acciones abarcadas por una estructura repetitiva.

Luego de la interrupción, **break**, se continúa con la sentencia que sigue a la estructura de control. Es decir, se interrumpe la iteración actual y se sale de la estructura repetitiva sin ejecutar las iteraciones restantes. **continue**, en cambio, salta al final de la estructura de repetición pero no la abandona, y permite continuar con la próxima iteración.

Ejemplo de `break`	Ejemplo de `continue`

`int a=0;`
`while (a<6) {`
`  a++;`
`  if (a==4)`
`  break;`
`  cout << a << " ";`
`}`
	`int a=0;`
	`while (a<6) {`
	`  a++;`
	`  if (a==4)`
	`  continue;`
	`  cout << a << " ";`
	`}`

Salida: 1 2 3	**Salida: 1 2 3 5 6**

uso de llaves { }

En todos los ejemplos presentados se utilizaron llaves ("{" y "}") para delimitar el conjunto de instrucciones que abarcaba una estructura de control. El uso de llaves es opcional cuando la estructura abarca una sola instrucción o estructura anidada.

Ejemplo

```
cin >> m;
if (m%2==0)
    cout << "m es par" << endl;
else
    cout << "m es impar" << endl;
cout << "fin"; // esta instrucción está fuera del if-else
```

Para evitar errores y confusiones, se recomienda al principio utilizar siempre las llaves, aunque se valla a escribir dentro una sola instrucción.