

Asignaciones y expresiones: principios básicos

Tema 5



José Ramón Paramá Gabía

Sentencia de asignación

- Una sentencia de asignación especifica una *expresión* que debe ser evaluada, y un *destino* donde guardar el resultado de evaluar la expresión.
- El destino puede ser una *variable*, una estructura de datos, etc.
- La sintaxis es la siguiente:
`variable = expresion;`
`variable :=expresion;`

Expresiones

- Las expresiones se pueden clasificar en:
 - *Expresiones aritméticas*: combinación de operadores aritméticos, operandos, paréntesis y llamadas a funciones. Al evaluarla nos da un valor numérico.
 - *Expresiones relacionales*: es aquella en la que se establece una comparación entre dos operandos a través de un operador relacional, por ejemplo:

== (igual que)	!= (diferente)	>
<	>=	<=
etc.		

Expresiones

Típicamente los operandos de las expresiones relacionales pueden ser de tipo numérico, cadenas de caracteres y tipos enumerados.

Si el lenguaje tiene tipo booleano, entonces el resultado de una expresión relacional es de ese tipo.

Obsérvese la diferencia entre el operador asignación (`=`, en C++) y el operador relacional de igualdad (`==`, en C++)

- *Expresiones Booleanas*: las expresiones booleanas se componen de variables de tipo booleano, literales booleanos (*true* o *false*), expresiones relacionales y operadores booleanos.

Los *operadores booleanos* suelen ser *NOT* (unario), *AND* y *OR* (binarios).

Operadores

- Se pueden clasificar en:
 - *Unarios*: admiten un único operando. Ej: -5 $\text{NOT}(a==b)$
 - *Binarios*: se aplican a dos operandos. Ej: $5+2$ $(x>0)$ AND $(y<0)$

Asignaciones con operadores aritméticos

- Algunos lenguajes como C, C++ y Java proporcionan operadores especiales que se pueden utilizar para combinar una operación aritmética con una asignación:

`x=x+5` es equivalente a `x +=5`

Así, cualquier sentencia de la forma:

```
var = var op expre;
```

Se puede escribir como

```
var op= expre;
```

Donde op puede ser (+, -, *, /, %) y expre puede ser cualquier expresión aritmética.

Esta notación se traduce de manera más eficiente a código máquina.

Incremento y decremento

- Otros operadores soportados por C, C++ y Java son *incremento* (`++`) y *decremento* (`--`).
- `++` suma 1 a su operando.
- `--` resta 1 a su operando.

Así $v = v + 1$ se puede escribir como `v++`

Y $v = v - 1$ se puede escribir como `v--`

- Estos operadores se pueden usar como prefijos (`++v`) o post fijos (`v++`).
- Con notación prefija aumenta el valor de la variable antes de que se use.
- Con notación postfija aumenta el valor después de usar la variable.

Incremento decremento

	Resultado	equivalencia
<code>i=0;</code> <code>x=i++;</code>	<code>x=0</code> <code>i=1</code>	<code>i=0;</code> <code>x=i;</code> <code>i=i+1;</code>
<code>i=0;</code> <code>x=++i;</code>	<code>x=1</code> <code>i=1</code>	<code>i=0;</code> <code>i=i+1;</code> <code>x=i;</code>

Asociatividad y precedencia

- La mayor parte de los lenguajes usan operadores con notación infija.
- Esta notación puede dar lugar a ambigüedad respecto al orden de evaluación. Ej. $a+b*c$ existe ambigüedad acerca si se debe realizar primero la suma o la multiplicación.
- Se puede usar paréntesis para romper la ambigüedad, Ej. $(a+b)*c$ o $a+(b*c)$.
- Para evitar tener que poner paréntesis los lenguajes definen reglas de *asociatividad* y *precedencia*, que determinan el orden en que los operadores son evaluados por defecto.

Reglas de precedencia

- Las reglas de precedencia de los operadores definen el orden en que se evalúan los operadores.
- Operadores aritméticos:
 - La exponenciación tiene la mayor precedencia (los lenguajes que lo tienen), seguida por la multiplicación y división al mismo nivel, seguidas por la suma y resta (al mismo nivel). Ej:
 $a+b*c**d$ sería $a+(b*(c**d))$, aunque los paréntesis siempre pueden variar el orden por defecto. Se podría escribir $(a+b*c)**d$.
- Operadores relacionales: Tienen menor precedencia que los aritméticos.
- Operadores lógicos: generalmente NOT tiene mayor precedencia seguido de AND, y finalmente OR.

Reglas de asociatividad

- Las reglas de asociatividad determinan cuando hay dos ocurrencias adyacentes de operadores en el mismo nivel de precedencia, qué operador se evalúa primero.
- Los operadores con la misma precedencia son típicamente agrupados de izquierda a derecha. Ej. $a-b-c$ se evalúa como $(a-b)-c$.
- Un operador es *asociativo hacia la izquierda* si las subexpresiones en las que en las que interviene varias veces el operador son agrupadas de izquierda a derecha. Son de este tipo la suma, resta, multiplicación y división.
- Un operador es *asociativo hacia la derecha* en caso contrario.

Asociatividad y precedencia

Precedencia	Operadores
Mayor	:: Operador de resolución de alcance
	. Operador punto => Selección de miembro [] Indexado de array () Llamada a función ++ Operador incremento postfijo (colocado después de la variable) -- Operador decremento postfijo (colocado después de la variable)
	++ Operador incremento prefijo (colocado antes de la variable) -- Operador decremento prefijo (colocado antes de la variable) ! Not - Menos unario + Mas unario * Desreferencia new delete delete[] sizeof
	* multiplicación / división % resto (módulo)
	+ suma - resta
	<< Operador inserción (salida) >> Operador extracción (entrada)
	< Menor que <= Menor que o igual > Mayor que >= Mayor que o igual
	== Igual != No igual
	&& And
	Or
Menor	= Asignación /= Divide y asigna += Suma y asigna -= Resta y asigna *= Multiplica y asigna %= Módulo y asigna

- Los situados en la misma celda tienen igual precedencia.
- Los situados en celdas superiores tienen mayor precedencia
- Los operadores unarios y el operador asignación se ejecutan de derecha a izquierda cuando los operadores tienen igual precedencia
- El resto de los operadores se ejecutan de izquierda a derecha cuando tienen igual precedencia

Sobrecarga de operadores

- Algunos operadores se pueden aplicar a distintos tipos de datos. Ej. el operador resta se puede aplicar a enteros y reales.
- Cuando un operador definido para un tipo de dato se define para otro tipo de dato se dice que se está *sobrecargando*.
- Dependiendo de los operandos el compilador traducirá el operador en distintas instrucciones de código máquina.

Conversiones de tipo

- Las conversiones de tipo pueden ser *explícitas* (realizadas por el programador) e *implícitas o coerciones* (realizadas automáticamente por el compilador).
- Las explícitas en C y C++ se denominan *cast*, su sintaxis es:
`(tipo nuevo) expresión`
- Un ejemplo de implícita es cuando se suma un entero y un real. El procesador sólo tiene suma de enteros o suma de reales, por lo que el compilador convierte el entero a real para poder usar la suma real.

Conversiones de tipo

- Las coerciones presentan una desventaja: restan eficacia a la *verificación de tipos* como herramienta para la detección de errores.
- Las coerciones pueden ser fuente de errores difíciles de depurar.
- Por ejemplo, C++ tiene el tipo booleano, pero por compatibilidad con C, se permite emplear expresiones numéricas enteras como expresiones lógicas. De esta forma en C++ se realiza la coerción entre los tipos entero y booleano.
 - *true* tiene valor 1 cuando se convierte a un entero, *false* 0.
 - Los enteros distintos de 0 se interpretan como *true* y el 0 como *false*.

Conversiones de tipo

- Por ejemplo:

```
bool b=7;          //b toma el valor true  
int i=true;        //i toma el valor 1
```

```
bool a=true;  
bool b=true;  
int s=a+b;         //s toma el valor 2  
bool c=s||a;        //como s vale 2, lo interpreta como true  
                    //así, c es el resultado de true OR true
```


Verificaciones de tipo

- Las reglas del sistema de tipos especifican qué usos puede tener cada operador.
- La *verificación de tipos* consiste en comprobar que cada operador está siendo usado en el programa respetando dichas reglas. El objetivo es prevenir errores.
- La verificación puede ser:
 - *Estática*, se realiza una vez, durante la traducción del programa a código máquina.
 - *Dinámica*, durante la ejecución del programa.
- Normalmente es *estática*.

Verificaciones de tipo

- Se dice que un lenguaje es *fuertemente tipado* cuando las reglas del sistema de tipos garantizan que las expresiones pueden ser completamente verificadas estáticamente.
- Java es un lenguaje *fuertemente tipado*, en Java no hay coerciones, cualquier incompatibilidad da lugar a errores de compilación.
- Por ejemplo en C, C++ es posible asignar un valor real a una variable entera, quedando en la variable sólo la parte entera del valor real (si asignamos 3.1415 a una variable entera, la variable almacenaría 3). En Java esta asignación no es posible.