ÍNDICE

Índice **1**

III Clase 3 **2**
1. Introducción . . . 2
2. Esquema de la clase . . . 2
3. Precedencia . . . 2
4. Asociatividad . . . 2
5. Sistema de tipos (Borrador) . . . 3
   - 5.1. Inferencia de tipos . . . 4
     - 5.1.1. Resumen . . . 5
     - 5.1.2. Mezcla de operandos con y sin signo . . . 5
   - 5.2. El costo de las operaciones . . . 5
   - 5.3. Definición de nuevos tipos . . . 5

Referencias **6**

III

CLASE 3

Estos son los apuntes no oficiales del curso CC3301-1 cursado en Primavera 2022 bajo las cátedras del docente Luis Mateu B. El material usado es directamente obtenido de cátedras, en adición a los materiales del curso. [1].

1 Introducción

Este capítulo está construido bajo las explicaciones de la clase grabada de 2020-1. El texto oficial de referencia se encuentra aquí.

2 Esquema de la clase

1) Asociatividad / Precedencia

2) Sistema de tipos

3) Operadores

3 Precedencia

Para entender bien este concepto pensemos en el siguiente caso: ¿Qué pasa si escribimos la expresión $a + b * c$?

Conocemos dos interpretaciones (con paréntesis como la orden de operación en aritmética), la primera $(a + b) * c$, la segunda es $a + (b * c)$. En este caso el orden que preferimos es la segunda opción.

Notemos que la multiplicación $(*)$ en este caso tiene más precedencia que la suma, pero en el caso de la división, por ejemplo, esta tiene la misma precedencia que la multiplicación de este caso.

En los apuntes podemos encontrar la precedencia de todos los operadores (un montón, pues C posee una basta cantidad de operadores). Pero la precedencia no resuelve la ambigüedad que existe en algunas expresiones, atendamos el caso de la sección siguiente.

4 Asociatividad

Basándonos en el ejemplo de $a+b*c$ donde elegimos la precedencia de la multiplicación, nos damos ahora la expresión $a-b+c$. De esta expresión resultan los casos: $(a-b)+c$ y $a-(b+c)$ donde no podemos usar el concepto de parecencia para escoger entre estos dos casos. En estos casos el concepto que nos quita este problema es el concepto de **Asociatividad**.

Hay dos posibilidades cuando hay dos operadores con la misma preferencia: Se asocia a la izquierda o se asocia a la derecha. En el caso de la suma y la resta la asociatividad ocurre hacia la izquierda: $(a - b) + c$. Gran mayoría de operadores asocia hacia la izquierda pero hay excepciones, e.g: el operador de asignación.

En el caso del operador de asignación tenemos, por ejemplo: $a = b = c$, que nuevamente genera los casos $(a = b) = c$ y $a = (b = c)$, donde la segunda es la interpretación correcta, pues el lenguaje C exige que lo que esté al lado izquierdo sea una variable (véase el concepto de *left value*).

5 Sistema de tipos (Borrador)

Nota: Esto es solo un borrador de la clase dictada en video, por ello se recomienda leer estas secciones directamente del apunte disponible en la web.

Supongamos que tenemos declarada una variable:

```
double x = ...;
int k = (int)(x+5)/3
```

Código 1: Ejemplo de código donde int es una *cast* de tipo primitivo y x+5 es de tipo double

Nos preguntamos ¿Cómo se determina el tipo de dicha expresión?. Recordemos que en el lenguaje C los tipos son estáticos, esto quiere decir que los tipos de cada expresión son determinados en tiempos de compilación (en *Python* son dinámicos y se determinan en tiempo de ejecución)

Para ellos tenemos que saber que aquí se usa la operación *value* en x (para el caso de 5 es una constante y ya es un valor, no hace falta pedir *value*).

Aquí x es un real y 5 un entero, entonces hace falta una conversión. Para ello se aplica un *cast* de tipo primitivo implícito (double) a 5. Una vez tenemos el mismo tipo a ambos lados, podemos realizar la operación double+ (suma real).

Después de realizar la suma existe un *cast* de tipo explícito en el código (int) para convertir este número real en un entero, y dado que 3 es entero y la suma real se ha convertido en entero, se realiza una división entera (entre int(x+5) y 3)

5.1 Inferencia de tipos

Consideremos otro ejemplo. Este ejemplo se encuentra detallado en el apunte disponible en `https://wiki.dcc.uchile.cl/cc3301/temario`. Para el propósito de la clase el capítulo de tipos está disponible bajo el mismo nombre de esta subsección.

```
int a=1;
int b=2;
double x = a/b;
```

Código 2: Segundo ejemplo

¿Cuanto vale x? Parece obvio: 0.5. ¡Pero vale 0! La razón es que el tipo de a/b se determina a partir del tipo de sus operandos, no a partir del uso que se de al resultado. Ya que el destino es una variable real, se tendería a pensar que la división debería ser con números reales, pero así no razona el compilador. La regla es que si ambos operandos son enteros, entonces la división es entera y el resultado tendrá 32 bits. Por eso 1/2 es 0. [2]

Otra regla para los operadores binarios es que si un operando es double, el otro operando se convierte implícitamente a double también. Por lo tanto para lograr la divisón real hay que usar un cast para convertir uno de los operandos a double: [2]

```
double x= (double)a / b;
```

Aquí (double) es un cast y tiene mayor precedencia que /. El código es equivalente a:

```
double x= ((double)a) / ((double)b);
```

Cuidado: ¡el siguiente código también entrega 0!

```
double x= (double) (a / b);
```

Consideremos otro ejemplo:

```
char a= 127;
char b= 1;
int c= a+b;
```

¿Cuanto vale c? ¿128? ¿o podría ser -128? De acuerdo al texto de más arriba, se podría pensar que la suma debería realizarse en 8 bits con signo. Pero 128 no es representable en 8 bits con signo. De hecho el resultado de la suma sería el valor binario 10000000, ¡que resulta ser -128 en 8 bits con signo! Pero el valor de c sí resulta ser 128. ¿Por qué?

Hay una regla en C que dice que todas las operaciones aritméticas deben considerar al menos el número de bits del tipo int, es decir 32 bits (casi siempre). Por lo tanto la asignación de c es equivalente a:

```
int c= (int)a + (int)b;
```

Pero cuidado, el número positivo más grande representable en 32 bits es 2147483647. Considere este código:

```
int a= 2147483647;
double x= a + 1;
```

En este caso, la asignación de x es equivalente a:

```
double x= (double)(a+1);
```

Por lo tanto a+1 se realiza en 32 bits con signo. El resultado en binario es un 1 seguido de 31 ceros, que corresponde al valor entero -2147483648. Ese es el valor incorrecto que queda almacenado finalmente en x. Y no 2147483648 como debería ser.

Ejercicio: Reescriba la instrucción de asignación cambiando todas las conversiones implícitas a conversiones explícitas.

```
double x;
char c;
long long ll;
...
int i= ll + c/2 + x*2;
```

```
double x= (double)(a+1);
```


```
double x= (double)(a+1);
```


5.1.1 *Resumen*

- El rango de un tipo numérico es el intervalo de números que puede representar.
- Los tipos numéricos están estrictamente ordenados por su rango:

  char <short <= int <= long <= long long <float <double
- Cuando se realiza una operación numéricas entre tipos distintos, el operando de un tipo con rango menor se convierte implícitamente al tipo del otro operando.
- No se realizan operaciones aritméticas con un tipo de rango inferior a int. Es decir cuando un operando es de tipo char o short se convierte implícitamente al menos a un int.

char <short <= int <= long <= long long <float <double Cuando se realiza una operación numéricas entre tipos distintos, el operando de un tipo con rango menor se convierte implícitamente al tipo del otro operando. No se realizan operaciones aritméticas con un tipo de rango inferior a int. Es decir cuando un operando es de tipo char o short se convierte implícitamente al menos a un int.

5.1.2 *Mezcla de operandos con y sin signo*

A partir de Ansi C se especifica que si un operando es con signo y el otro sin signo, entonces el operando sin signo se convierte implícitamente a un tipo con signo y la operación se realiza con signo. Esto significa que si un operando es unsigned int y se suma con un int, entonces el primero se convierte a int. ¡Cuidado! En esta conversión se podría producir un desborde. [2]

5.2 El costo de las operaciones

Dado que C es un lenguaje en donde interesa la eficiencia, resulta importante conocer cual es el costo de las operaciones aritméticas en términos de su latencia. La latencia de una operación es la cantidad de ciclos del reloj del procesador que deben pasar para poder usar el resultado de esa operación. [2]

Símbolo	Tipo de datos	Latencia (ciclos)	Observaciones
+, -	int	1	text
*	int	3	en un procesador reciente
/	int	8, 16 o 32	1 ciclo por cada 1, 2 o 4 bits del divisor
+, -, *	float/double	3	en un procesador reciente
/	float	8 a 32 ciclos	1 ciclo por cada 1, 2 o 4 bits del divisor
/	double	16 a 64 ciclos	1 ciclo por cada 1, 2 o 4 bits del divisor

Observe que la suma y la resta de enteros son las operaciones más eficientes, seguidas de la suma, resta y multiplicación de números en punto flotante. Por otro lado, la división es lejos la operación más lenta. Esto se debe a que su implementación en circuitos requiere un ciclo por cada 4 bits del divisor, en los procesadores más recientes (Haswell). En los procesadores menos recientes (Sandy Bridge) pueden ser 1 ciclo por cada 2 bits del divisor e incluso 1 ciclo por cada bit del divisor (más lento).

5.3 Definición de nuevos tipos

En C se definen nuevos tipos con struct, union, enum y typedef. Estos temas serán abordados más adelante. [2]

REFERENCIAS

[1] Mateu, Luis: *Programación de Software de Sistemas - Novedades.* `https://www.u-cursos.cl/ingenieria/2022/1/CC3301/1/novedades/`, 2022.

[2] Mateu, Luis: *Programación de Software de Sistemas - Apuntes.* `https://wiki.dcc.uchile.cl/cc3301/temario/`, 2022.