Expresiones en C

- Asociatividad
- Precedencia
- Reglas de inferencia de tipos
- Conversiones de tipos primitivos: cast
- Rango de tipos numéricos
- Tipo de constantes
- La latencia de los operadores

Precedencia

- ¿Cuánto vale a + b * c?
- La expresión es ambigua. Puede ser:
  - (a + b) * c , o
  - a + (b*c)
- La precedencia de los operadores dice cuál parentización es válida: la primera porque la multiplicación tiene mayor precedencia que la suma
- Se parentiza primero los operadores de mayor precedencia
- * y / tienen la misma precedencia y mayor que + y -
- En la Wikipedia puede consultar la tabla de precedencia y todos los operadores de C y C++: link

Operador	Descripción	Asociatividad
`::`	Resolución de ámbito (solo C++)	Izquierda a derecha

`++` `--`
`()`
`[]`
`.`
`->`
`typeid()`
`const_cast`
`dynamic_cast`
`reinterpret_cast`
`static_cast`
	Post- incremento y decremento
	Llamada a función
	Elemento de vector
	Selección de elemento por referencia
	Selección de elemento con puntero
	Información de tipo en tiempo de ejecución (solo C++)
	Conversión de tipo (solo C++)
	Conversión de tipo (solo C++)
	Conversión de tipo (solo C++)
	Conversión de tipo (solo C++)	

`++` `--`
`+` `-`
`!` `~`
`(type)`
`*`
`&`
`sizeof`
`new` `new[]`
`delete` `delete[]`
	Pre- incremento y decremento
	Suma y resta unitaria
	NOT lógico y NOT binario
	Conversión de tipo
	Indirección
	Dirección de
	Tamaño de
	Asignación dinámica de memoria (solo C++)
	Desasignación dinámica de memoria (solo C++)
		Derecha a izquierda

`.*` `->*`	Puntero a miembro (solo C++)	Izquierda a derecha
`*` `/` `%`	Multiplicación, división y módulo	
`+` `-`	Suma y resta	
`<<` `>>`	Operaciones binarias de desplazamiento	
`<` `<=` `>` `>=`	Operadores relaciones "menor que", "menor o igual que", "mayor que" y "mayor o igual que"	
`==` `!=`	Operadores relaciones "igual a" y "distinto de"	
`&`	AND binario	
`^`	XOR binario	
`|`	OR binario	
`&&`	AND lógico	
`||`	OR lógico	
`c?t:f`	Operador ternario	Derecha a izquierda

`=`
`+=` `-=`
`*=` `/=` `%=`
`<<=` `>>=`
`&=` `^=` `|=`
	Asignaciones	

`throw`	Operador Throw (lanzamiento de excepciones, solo C++)	
`,`	Este es el operador de menor prioridad en C. Sirve para separar una colección de expresiones, que se irán evaluando de izquierda a derecha	Izquierda a derecha

Asociatividad

- ¿Cuánto vale a - b + c?
- La expresión es ambigua. Puede ser:
  - (a-b) + c , o
  - a – (b+c)
- La asociatividad de los operadores dice cuál parentización es válida: la primera porque la suma y resta se asocian de izquierda a derecha
- La mayoría de los operadores se asocian de izquierda a derecha
- Hay excepciones: = se asocia de derecha a izquierda
- a = b = c se parentiza como a = (b = c)
- (a = b) = c es una expresión inválida en C
- El mismo link de la página anterior indica la asociatividad para cada operador

Reglas de inferencia de tipos

- ¿Cuánto vale *x* finalmente?

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

- No es obvio
- En C las operaciones + - * / etc. dependen del tipo de los operandos
- Las reglas de inferencia de tipos especifican si una operación es *int*, *long long*, *float* o *double*
- El procesador usa instrucciones de máquina distintas para cada uno de estos tipos
- Ejemplo de regla: si ambos operandos son *int*, la operación es *int*
- Como *a* y *b* son *int*, entonces la división en *a / b* es *int* y ¡el resultado es 0!
- *No importa que el resultado se asigne a una variable double*

Conversiones de tipos: cast

- Un cast permite convertir una expresión al tipo de deseado
- Por ejemplo *(int) x* convierte el valor 3.14 almacenado en *x* al entero 3
- Una instrucción del procesador realiza la conversión
- En general la expresión *(Tipo) exp* es un *cast* de tipo primitivo cuando
  - Tipo es primitivo (*char*, *short*, *int*, *float*, *double*)
  - El tipo de *exp* también es primitivo
- Ejemplo:

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

- El *cast* tiene mayor precedencia que * / + -
- El valor final de *x* es 0.5 porque la división es real

El rango de los tipos numéricos

- Es el intervalo de números que puede representar
- El rango se puede ordenar por la relación $\subset$ y $\subseteq$

  char $\subset$ short $\subseteq$ int $\subseteq$ long $\subseteq$ long long $\subset$ float $\subset$ double
- ¿Cuál es el resultado final de *x*?

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

- No es obvio porque los tipos de los operandos difieren
- Regla de inferencia de tipos: cuando los tipos de ambos operandos difieren el operando con rango menor se convierte implícitamente con un *cast* al tipo de mayor rango
- El resultado es 0.5 porque b se convierte a *double*
- El programa es equivalente al de la página anterior: el compilador genera las mismas instrucciones de máquina

Otra regla de inferencia de tipos

- ¿Cuánto vale *c* finalmente?

```
signed char a = 127;        // 0b0111 1111
signed char b = 1;          // 0b0000 0001
int c = a + b;
```

- No es obvio
- En C no existen los operadores + - * / para *char* o *short*
- Los procesadores Risc (ARM, Risc-V) no tienen instrucciones de máquina para + - * / para *char* o *short,* solo tienen para *int y long long*
- Los procesadores Intel/Amd sí tienen esas instrucciones, pero el compilador de C no las usa
- Nueva regla de inferencia de tipos: en una operación entera si ambos operandos son de rango inferior a *int*, se convierten implícitamente a *int*
- El resultado es correcto: 128

El tipo de las constantes

- ¿Cuánto vale *x* finalmente?

```
int a = 2147483647;          // 0b0111 …. 11111
long long x = a + 1;  // 0b0000 …. 00001
```

- No es obvio
- La suma es de tipo *int* porque ambos son de tipo *int*
- No importa que se almacene en un long long
- El resultado es incorrecto: -2147483648
- Se puede cambiar el tipo de las constantes a long long con el sufijo LL

```
int a = 2147483647;          // 0b0111 …. 11111
long long x = a + 1LL;          // 0b0000 …. 00001
```

- El resultado es correcto: 2147483648
- Porque *a* se convierte a *long long* y la suma es *long long*
- Si no se especifica, el tipo es *int* para los enteros y *double* para los reales

Expresiones complejas

- Una expresión compleja contiene varios operadores
- ¿Cuanto vale 3 / 2 + 1.5?
- No es obvio
- Considere una expresión *e operador f* en donde *e* y *f* son subexpresiones que incluyen otros operadores
- La regla es que se infiere recursivamente el tipo de *e* sin importar el tipo de *f* y viceversa
- Tampoco importa el uso que se dé a *e operador f*
- La parentización implícita es (3 / 2) + 1.5
- Como 3 y 2 son int, el tipo de / es int y el resultado es 1
- Luego se convierte al double 1.0 y se suma a 1.5
- El resultado es 2.5, no 3

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;
```

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.

Tipos estáticos vs. tipos dinámicos

- El sistema de tipos de C es estático
  - Toda variable debe ser declarada indicando su tipo
  - Ejemplo: *double x = 3.14; int i = 2;*
  - Además, para cada expresión y subexpresión el compilador infiere su tipo durante la compilación
  - Ejemplo: en la expresión *i * x* *i* es de tipo *int*, x de tipo *double y i * x* es de tipo *double*
  - Ventaja: los programas en C se compilan directamente a lenguaje de máquina
- El sistema de tipos de Python es dinámico
  - El tipo de las expresiones se determina en tiempo de ejecución
  - En Python una variable *x* puede almacenar valores de cualquier tipo
  - No resulta práctico compilar los programas en Python directamente a lenguaje de máquina
  - Por lo tanto, los programas se interpretan resultado al menos 10 veces más lento que el equivalente en C

La latencia de las operaciones

- Cuando se necesita eficiencia en un programa es importante conocer cuál es la latencia de las operaciones aritméticas
- Los procesadores modernos ejecutan varias instrucciones de máquina en paralelo
- La *latencia* es el número de ciclos del reloj que deben pasar para poder usar el resultado de la operación en una nueva operación

Símbolo	Tipo de datos	Latencia	Observaciones
+ -	int	1	¡siempre!
*	int	3	típicamente
/	int	8, 16, 32	1 ciclo por cada 1, 2 o 4 bits de divisor
+ - *	float/double	3/4	típicamente
/	float/double	8 a 32/16 a 64	1 ciclo por cada 1, 2 o 4 bits del divisor

Ejemplos

- Considere que *a*, *b*, *c* y *d* son *int* y se encuentran en registros del procesador (no en memoria)
- ¿Cuántos ciclos se requiere para calcular (a+b)*(c+d)?
- 4 típicamente porque ambas sumas se realizan en paralelo

- ¿Cuántos ciclos se requiere para calcular (a*b)+(c*d)?
- 7 típicamente porque los procesadores tienen un solo multiplicador de enteros

- El número exacto de ciclos depende de la arquitectura física del procesador (su implementación)