



Espacios de nombres (name spaces)

- Los espacios de nombres tienen por objetivo poder referirse a objetos distintos, con un mismo identificador en el mismo ámbito.
- En lenguaje C no hay un manejo de espacios de nombre por parte del programador, pero hay definidos 4 espacios de nombre.
- Podemos repetir el mismo nombre en distintos espacios, pero no repetir uno dentro de un mismo espacio.



Espacios de nombres en C

- Las etiquetas, reconocidas por su sintaxis (seguida de :)
- Los tags (el nombre) de estructuras, uniones y enumeraciones. Al estar en un mismo espacio de nombre no puedo usar, por ejemplo, para una estructura el mismo nombre que use para una enumeración.
- Los miembros de las estructuras y uniones, cada tag crea su propio espacio de nombres, por eso pueden repetirse de, por ejemplo, una estructura a otra.
- Los identificadores ordinarios (comunes): todos los demás, incluyendo variables, funciones, constantes de enumeración y alias de tipos introducidos con typedef.



Ejemplo

```
5  int fun(void)
6  {
7      return 3;
8  }
9  int fun;
10 //main.c:9:5: error: 'fun' redeclared as different kind of symbol
11
12 int main(void)
13 {
14     int nombre; //ok
15     struct nombre {int campo;}; //ok
16     struct otra {int campo;}; //ok, campo se repite
17                               //pero en otro espacio de nombre
18     enum otra {UNO, DOS, nombre}; // dos errores
19     //se repite el tag otra y se repite nombre en el espacio ordinario
20     //main.c:18:7: error: 'otra' defined as wrong kind of tag
21     //main.c:18:23: error: 'nombre' redeclared as different
22     //                      kind of symbol
23     int fun; //Distinto ámbito, oculta lo declarado a
24             //nivel de archivo
25 nombre: //ok, las etiquetas tienen su propio espacio de nombres
26     return EXIT_SUCCESS;
27 }
```



Tipos

- El significado de un valor guardado en un objeto o devuelto por una función está determinado por el tipo de expresión usado para accederlo.
- La expresión más simple es un identificador y su tipo fue especificado al declararlo.
- Los tipos se dividen en **tipos de objetos** (variables) y **tipos de funciones** (que describen su firma)
- Los tipos de objetos pueden, en un determinado punto de la unidad de traducción, estar incompletos (y ser completados luego). Cuando esto ocurre no se puede especificar el tamaño necesario para guardarlo, y por lo tanto no se puede declarar variables de un tipo incompleto (pero si punteros).
Casos: arreglos sin dimensión, estructuras o uniones sin sus miembros.



Tipos de Objetos (Básicos)

- void es un tipo especial que tiene un conjunto vacío de valores, se lo considera incompleto y no puede completarse.
- _Bool es considerado de tipo entero sin signo con la capacidad necesaria para almacenar un 0 y un 1 (típicamente un byte). Si se le asigna el valor de una expresión entera su valor será cero si la expresión era igual a cero y 1 en cualquier otro caso.
- Tipos carácter: char, unsigned char, signed char. Se los considera 3 tipos diferentes, char termina siendo signed o unsigned dependiendo de la implementación. Tiene espacio necesario para guardar un carácter del juego básico (típicamente un byte).
- Tipos enumerados: cada vez que defino un enum tag se lo considera un tipo enumerado.



Tipos de Objetos (Básicos)

- Tipos enteros estándar: int y char con y sin signo. Incluye la variantes de int como short, long y long long y los tipos de enumeración.
- Tipos enteros: los enteros estándar más _Bool
- Tipos reales flotantes: float, double y long double.
- Tipos complejos: su implementación es optativa. float _Complex, double _Complex y long double _Complex.
- Tipos flotantes: la unión de reales flotantes y complejos.
- Tipos reales: la unión de reales flotantes más los tipos enteros.
- Tipos aritméticos: la unión de flotantes más los tipos enteros.
- Tipos básicos: la unión de tipos de carácter con tipos enteros y tipos flotantes. Son tipos completos. (todo lo nombrado hasta aquí excepto void)



Tipos Derivados

- A partir de los tipos de objetos y de funciones se pueden derivar otros (y se puede derivar recursivamente):
- Tipo arreglo, define una área contigua de memoria donde se ubican una cantidad no vacía de miembros que son objetos de un determinado tipo. Si llamamos T al tipo de sus elementos, entonces decimos que es un "arreglo de T". Un arreglo se caracteriza por el tipo de sus elementos y la cantidad de los mismos.
- Tipo estructura, aloja una secuencia no vacía de miembros objetos que pueden ser de diferentes tipos.
- Tipo unión, similar a la estructura pero memoria asignada a los miembros se solapa (todas comienzan al principio de la unión)



Tipos Derivados

- Tipo Función, las funciones se caracterizan por el tipo que devuelven y la cantidad y tipo de sus parámetros. Dado un tipo T la derivación en función es una función que devuelve T .
- Tipo puntero, pueden derivar de un tipo objeto o un tipo función al que genéricamente llamaremos tipo referenciado.
- Las reglas para derivar tipos se pueden aplicar recursivamente y así tener, por ejemplo “arreglo de estructuras” o “arreglos de arreglos de punteros a funciones”



Tipos, otras consideraciones

- Tipos agregados: la unión de tipo estructura y tipo arreglo.
- Tipos escalares: la unión de tipo aritmético y tipo puntero.
- ValorL: es una expresión que permite designar (ubicar) un objeto y recuperar su valor. El caso más simple es el identificador de una variable, por ejemplo de tipo básico. Otros ejemplos: un elemento de un arreglo o lo apuntado por un puntero.
- Un valorL es modificable si no fue calificado como constante, ni es una estructura o unión con un miembro calificado como constante. Tampoco son valorL modificables los arreglos (no hablo de sus elementos) y los tipos incompletos.
- ValorR: es el valor en si mismo es una expresión



Categorías Gramaticales

- Son los LIC que se usan en la sintaxis del lenguaje
- Se dividen en
 - Expresiones
 - Declaraciones y Definiciones
 - Sentencias
- Todas se completan con restricciones, sean sintácticas o semánticas.



Expresiones

- Básicamente es una secuencia de operadores y operandos que:
 - Producen un valor, con el eventual uso de paréntesis para la precedencia
 - Designan un objeto o función
 - Generan un efecto lateral
 - O bien combinaciones de los anteriores
 - Las constantes, variables y llamados a funciones forman parte de las expresiones primarias o elementales



Expresiones - Tipos

- Booleanos: en lenguaje C “*no hay*” valores booleanos en el sentido de otros lenguajes, sino que forma parte de los tipos enteros.
- Se considera, a efectos de evaluación, 0 como falso y distinto de cero como verdadero. Si una expresión, por ejemplo usando el operador `<`, es verdadera, devuelve un 1 como valor estándar de verdadero (tipo `int`).
- Para conveniencia de uso se definen en el encabezado `stdbool.h` `bool` como alias de `_Bool` y los valores `true` y `false`.
- Los operadores aritméticos necesitan tipos iguales, si no lo son aplican las llamadas “conversiones aritméticas habituales” que un casting implícito para llevar al tipo con mayor capacidad de representación de valores.



Expresiones - evaluación

- Las precedencias quedan definidas en la gramática por la cercanía o lejanía del operador con respecto al axioma
- La asociatividad está dada por la recursividad (a izquierda o a derecha) de la producción.
- La precedencia junto con la asociatividad nos da el agrupamiento, que indica el orden de las operaciones. Esto nos permite saber que

– $a + b * c$ Se evaluará como $a + (b * c)$

– $a = b = c$ Se evaluará como $a = (b = c)$



Efecto lateral o secundario (side effect)

- Partiendo de la idea que una función o expresión debe calcular un valor e idealmente no hacer nada más, un efecto lateral es todo cambio adicional al entorno de ejecución. Ejemplos:
 - modificación de una variable global
 - modificación de un argumento real de una función
 - escribir en un archivo.
- Estos efectos son comunes en el paradigma imperativo. El tema a tener en cuenta es como ocurren en el tiempo. Al realizar el cálculo de una expresión puedo iniciar un efecto secundario, pero, cuando cuando se realiza, cuando puedo estar seguro que finalizo?. En el estándar de C trata el tema definiendo secuenciación.



Secuenciación

- Se trata de establecer una línea de tiempo, en que orden se suceden los eventos. Tomando en cuenta un solo hilo de ejecución tenemos:
- Se dice que una evaluación A esta secuenciada antes que otra B si la ejecución A ocurre antes que la de B.
- Si A y B no están secuenciados significa que puede ejecutarse primero A y luego B o al revés, pero no se especifica.
- Punto de secuencia: es un instante estable que divide dos ejecuciones, supongamos primero A y luego B, si hay un punto de secuencia entre ambos cumplirá con haber terminado todas las evaluaciones y efectos laterales de A y no haber comenzado ni las evaluaciones ni los efectos laterales de B.



Secuenciación - Operandos

- El estándar aclara que salvo en casos específicos, que mencionaremos luego, los efectos laterales y los cálculos de valores de subexpresiones no están secuenciados.
- Al margen de algunos comportamientos no definidos que pueden surgir con algunas combinaciones de efectos laterales, lo que implica es que el orden de evaluación de los operandos no está definido, dando lugar a que el compilador haga optimizaciones
- Si tengo $a() + b() * c()$ el orden de ejecución de las tres funciones puede ser cualquiera de los 6 posibles. Si sabemos que primero hará el producto de $b()$ con $c()$ y luego le sumará el resultado de $a()$, pero puede ejecutar $a()$ primero y guardar su valor para su uso posterior.



Secuenciación - Operandos

- El orden de evaluación de los operandos **no** está definido, salvo en los siguientes casos, mediante puntos de secuencia:
 - Operadores `&&` y `||` aseguran la evaluación de izquierda a derecha. Detienen la evaluación en cuanto el resultado esté definido.
 - Operador ternario `?:` asegura evaluar el primer operando y luego el segundo o tercero según corresponda
 - El operador coma garantiza que las expresiones (operandos de coma) se evalúan de izquierda a derecha, todas menos la última son evaluadas como `void`, y el resultado es del tipo al que evalúe la última expresión.



Puntos de Secuencia

- El anexo C del estándar los lista todos, ya hemos nombrado algunos y estos son otros de los más relevantes:
 - No hay un orden para evaluar los argumentos reales, pero si hay un punto de secuencia luego de haberlos evaluados todos y antes de invocar a la función.
 - Luego de evaluar la expresión de un return hay un punto de secuencia antes se resumir la ejecución de la función que invocante.
 - Luego de cada expresión completa (full expression):
 - La expresión que controla un: if, switch, while, do-while
 - Cualquiera de las expresiones de un for
 - Una sentencia expresión.



Operadores de incremento y decremento

- Postfijo: el operando debe ser un valorL modificable. El resultado es el valor (valorR) del operando y como efecto lateral se suma o resta 1 al operando. La obtención del valor se secuencia antes que el efecto lateral. En otras palabras, se hace intervenir el valor original en la expresión en la cual se halla y luego se incrementa.

```
b = 2; c = 3;  
a = b++ * c; //a vale 6 y b vale 3
```

- Prefijo: Primero se incrementa o decrementa el operando (efecto lateral), el valor (valorR) incrementado o decrementado es el resultado del operador. En otras palabras, primero se incrementa y luego se lo aplica a la expresión en la que se encuentra.

```
b = 2; c = 3;  
a = ++b * c; //a vale 9 y b vale 3
```



Secuenciación, casos no definidos

- Si aplico más de un efecto lateral a un mismo objeto escalar sin un punto de secuencia en medio, o si tengo el efecto lateral y la evaluación del objeto no secuenciados, entonces el comportamiento es no definido:

- `i = ++i + 1;`
- `v[i++] = i;`
- `a++ + a++ * a++;`

- Pero si son correctos:

- `i = i + 1;`
- `a[i] = i;`
- `a++ || a++ && a++;`



Expresiones BNF

- En el estándar en lugar de comenzar con el axioma e ir alejándose, lo hace en orden inverso. Comienza con:

```
primary-expression:  
    identifier  
    constant  
    string-literal  
    ( expression )  
    generic-selection
```



Expresiones BNF

- Luego introduce los operadores de más alta precedencia, noten que “baja” a `primary-expression` que es la producción con la que inició:

`postfix-expression`:

`primary-expression`

`postfix-expression` [`expression`]

`Postfix-expression` (`argument-expression-list`_{opt}
)

`postfix-expression` . `identifier`

`postfix-expression` -> `identifier`

`postfix-expression` ++

`postfix-expression` --

(`type-name`) { `initializer-list` }

(`type-name`) { `initializer-list` , }



Expresiones BNF

- Resaltamos la precedencia y asociatividad:

multiplicative-expression:

cast-expression

multiplicative-expression * cast-expression

multiplicative-expression / cast-expression

multiplicative-expression % cast-expression

additive-expression:

multiplicative-expression

additive-expression + multiplicative-expression

additive-expression - multiplicative-expression



multiplicative-expression

- Restricciones
 - Los operandos deben ser de tipo aritmético
 - Pero para el operador % deben ser de tipo entero
- Semántica
 - Se aplican a los operandos las conversiones aritméticas usuales (empareja el tipo de dato con el de mayor capacidad de representación)
 - Para los operadores / y % si el segundo operando es cero, el comportamiento es no definido.



Expresiones BNF

- Y va subiendo hasta llegar al axioma, notar que coma es el operador a nivel de axioma y por tanto el de menor precedencia:

assignment-expression:

conditional-expression

unary-expression assig-operator **assignment-expression**

assig-operator: one of

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

expression:

assignment-expression

expression , assignment-expression



Declaraciones y Definiciones

- Declarar implica dar a conocer, es decir especificar la interpretación y atributos de un identificador.
- Una definición es una declaración que además:
 - Si es un objeto reserva espacio en memoria.
 - Si es una función da el código de la misma.
 - Para los typedef y las constantes de enumeración la declaración es definición.
 - Decimos que los tag los declaramos, pero en estructuras y uniones definimos su contenido (sus miembros).



Declaraciones y Definiciones

- Ejemplo con objetos

```
int largo; //Definición
```

```
extern double peso; //Declaración
```

- Ejemplo con funciones

```
//Declaración, autodocumentada
```

```
double dividir(double dividendo, double divisor);
```

```
//Declaración con solo tipo de dato
```

```
double dividir(double, double);
```

```
//Definición
```

```
double dividir(double dividendo, double divisor) {  
    return divisor != 0 ? dividendo / divisor : 0;  
}
```



Modificadores

- Storage Class: altera donde se almacena
 - extern: nada, se resuelve en otra unidad de traducción
 - static: en heap en lugar de en stack
 - register: si puede en registro del CPU
- Type-qualifier: califican un comportamiento particular
 - const: no puede alterarse su valor
 - restrict: es el único puntero para acceder a la memoria apuntada
 - volatile: otro programa o hilo puede modificar su valor
- Function-specifier:
 - inline: se incrusta su lógica en cada lugar que es invocada
 - _Noreturn: no se genera código para resumir al ejecución de la función que la invoque



Nombres de Tipos

- A veces hace falta especificar un tipo de datos, por ejemplo para un cast o para declarar una función. La especificación es igual a declarar un identificador con ese tipo de datos pero sin poner el identificador.
- El orden en que se combinan las declaraciones siguen las precedencias de los símbolos usados considerados como operadores

`int id → int ⇒ entero`

`int *id → int * ⇒ puntero a entero`

`int *id[3] → int *[3] ⇒ arreglo de 3 punteros a entero`

`int (*id)[3] → int (*)(3) ⇒ puntero a arreglo de 3 enteros`

`int *id(void) → int *(void) ⇒ función sin parámetros que devuelve puntero a entero`

`int (*id)(void) → int (*)(void) ⇒ puntero a función sin parámetros que devuelve entero`

`int (*id[])(double) → int (*[])(double)`

arreglo de punteros a función con un parámetro double que devuelve entero



Sentencias

- BNF de sentencias:
statement:
 labeled-statement
 compound-statement
 expression-statement
 selection-statement
 iteration-statement
 jump-statement



Sentencias

- Sentencia etiquetada, 3 posibilidades
 - `identifier : statement`
 - `case constant-expression : statement`
 - `default : statement`
- Los casos de `case` y `default` obviamente deben estar dentro de un `switch`
- El primer modo es para usar en `goto`.
- Noten que siempre después de los dos puntos viene una sentencia, eso implica que no puedo por ejemplo, poner la definir una variable inmediatamente después de los dos puntos.



Sentencias

- Sentencia compuesta (o bloque)
 - Entre { }, puede que solo estén las llaves (no hace nada)
 - Define un nuevo ámbito.
- Sentencia expresión
 - Expresión con ; al final. En general asignaciones.
 - El ; marca un punto de secuencia.
 - Puede ser nula (solo ;)
 - Si pones algo como $2*3$; da warning **unused value** pero NO es un error



Sentencias

- Sentencias de selección

selection-statement:

if (expression) statement

if (expression) statement else statement

switch (expression) statement

- El tipo de dato de la expresión que controla el if debe ser escalar
- El else es optativo, pero de estar, empareja con el if más cercano.
- La expresión que controla un switch debe ser entera.
- Recordar que luego de : debe ir una sentencia.
- Los case no son obligatorios pero no tiene sentido no ponerlos
- Las sentencias etiquetadas no generan un “corte de estructura”, sin el break se sigue ejecutando.



Sentencias

- Iteración
 - La expresión que controla una iteración debe ser de tipo escalar. Se itera hasta que la expresión de control sea cero.
 - **while** itera mientras la expresión sea verdadera

```
while (exp)
    sentencia
```
 - **do while**: similar pero con la comprobación abajo

```
do
    sentencia
while (exp);
```



Sentencias

- **for**: modo tradicional de explicarlo, si tiene la forma

```
for (exp1 ; exp2 ; exp3)  
    sentencia
```

es equivalente a

```
exp1;  
while (exp2) {  
    sentencia  
    exp3;  
}
```

- TODAS las exp_i son opcionales, las expresiones 1 y 3 se evalúan como void si no están, en tanto que exp₂ se reemplaza por una constante verdadera si se lo omite.



Sentencias

- **for:** hoy día el for se especifica como:
 for (clausula₁ ; exp₂ ; exp₃)
 sentencia
 - Donde clausula 1 puede definir variables que cuyo ámbito es solo el for.
 - for (;;) es un loop infinito



Sentencias

- Salto
 - return exp_{op}
 - Sale de la función y retorna el control al código que invocó la función. Si la función devuelve un valor la exp_{op} es el valor retornado
 - Si el tipo devuelto por la función no coincide con el tipo de la expresión del return, se convierte de igual modo que en una asignación.
 - goto etiqueta
 - Salto incondicional a la sentencia etiquetada con la etiqueta que acompaña al goto.
 - El salto es dentro de una función



Sentencias

- Salto
 - break
 - Sale del ciclo iterativo o del switch que lo contiene y pasa a ejecutar la sentencia siguiente.
 - continue
 - Saltea lo que queda de esa iteración pero vuelve al control del ciclo y si corresponde sigue iterando
 - En el caso de un for vuelve a evaluar la expresión 3 antes de comprobar la expresión de control



Licencia

*Esta obra, © de Eduardo Zúñiga, está protegida legalmente bajo una licencia Creative Commons, **Atribución-CompartirDerivadasIgual 4.0 Internacional**.*

<http://creativecommons.org/licenses/by-sa/4.0/>

Se permite: copiar, distribuir y comunicar públicamente la obra; hacer obras derivadas y hacer un uso comercial de la misma.

Siempre que se cite al autor y se herede la licencia.

