Tipos enumerativos y tablas de búsqueda

Sebastián Santisi

2019-04-15

¿Cuándo se usan los tipos enumerativos?

Los tipos enumerativos en C permiten definir un valor *con nombre propio* entre un conjunto cerrado de casos posibles. Por ejemplo, para modelar los días de la semana, los enumerativos nos permiten crear variables a las que podremos asignarle la etiqueta `LUNES` en vez de convenir asignar `0` o `1` o la letra `'L'`. Cada vez que haya que modelar en qué estado está una variable entre un número finito de posibilidades donde esas posibilidades se identifican con un nombre usaremos tipos enumerativos. (Puede expresarse esto de otra manera: Cada vez que estés inventando valores arbitrarios para categorizar cosas estás haciendo mal las cosas y deberías haber definido un enumerativo.)

La declaración de un tipo enumerativo es simplemente una secuencia de valores:

```
enum reino {ANIMALIA, PLANTAE, FUNGI, BACTERIA, PROTISTA, ARCHAEA};
enum reino especimen_a = ANIMALIA;
```

(Eventualmente podríamos haber hecho un `typedef` para redefinir `enum reino` a algo más cómodo y sintético como `reino_t`.)

Sólo porque va a utilizarse más adelante se menciona lo siguiente: Las etiquetas tienen valores numéricos, consecutivos y comienzan por el número cero.

Si bien internamente el tipo `enum` se representa como un entero y los valores de las etiquetas son numéricos en *la mayor parte* de las ocasiones **está mal** preocuparse por cuánto vale ese valor (sea forzándolo o asignándolo con un literal). También siendo enteros podría asignárseles cualquier valor numérico no incluido entre las etiquetas definidas, esto sería **muy grave** dado que representaría una violación al invariante del enumerativo. Ambas cosas debe evitarse a toda costa. (Y puede expresarse también de otra manera: Si es importante el valor numérico preguntate si no deberías haber elegido una cosa distinta a un enumerativo.)

Cuando utilizamos enumerativos el compilador puede hacer chequeos sobre los valores de los mismos. Por ejemplo, si entráramos en un `switch` con un enumerativo (esto se puede porque `switch` funciona sobre enteros y un enumerativo **es** un entero) el compilador podría avisarnos si nos estamos olvidando de incluir un `case` para un valor puntual del mismo.

Es importante destacar que los `enum` sirven para hacer más claro y legible el código y son una herramienta **para el programador**. Como tales no hay herramientas para convertir, por ejemplo, la etiqueta `ANIMALIA` en la cadena `"ANIMALIA"` ni hay herramientas para leer un enumerativo.

Tablas de búsqueda para traducir enumerativos a cadenas (o a cualquier tipo)

Si necesitáramos traducir entre la representación interna del enumerativo y otras representaciones adecuadas para interactuar con el exterior vamos a aprovechar la propiedad de la numeración de las etiquetas. Si un enumerativo tiene `N` etiquetas sabemos que las mismas serán numéricas y numeradas entre `0` y `N-1` por lo que se hace inmediato aparear enumerativos a arreglos. Es decir, se pueden

construir "tablas de búsqueda" que *asocian* un valor arbitrario en algún tipo con un enumerativo según la posición en el mismo del valor.

Por ejemplo:

```
const char *nombres_reino[] = {
  "animalia", "plantae", "fungi", "bacteria", "protista", "archaea"
};

char *reino_a_cadena(enum reino r) {
  return nombres_reino[r];
}
```

Notar una de las precondiciones implícitas de la función es asumir que se cumple el invariante del enumerativo, es decir, que el mismo contiene una etiqueta válida de la definición del `enum`.

Hay otras dos precondiciones que se asumen, y que deben ser garantizadas por el programador, que son que el enumerativo está declarado con sus claves consecutivas e iniciando desde cero y que tanto el enumerativo como el arreglo de cadenas están declarados con el mismo orden en sus elementos.

Dentro del ISO-C99 podemos evitar los problemas que ocurrirían de "desfasarse" la declaración del enum con la del arreglo de cadenas declarando el arreglo con sus índices de forma explícita:

```
const char *nombres_reino[] = {
  [ANIMALIA] = "animalia",
  [PLANTAE] = "plantae",
  [FUNGI] = "fungi",
  [BACTERIA] = "bacteria",
  [PROTISTA] = "protista",
  [ARCHAEA] = "archaea",
};
```

Esto es más engorroso pero también más robusto.

Si quisiéramos realizar el proceso contrario, esto es, implementar una función `enum reino cadena_a_reino(const char *s);` podríamos iterar comparando los elementos de `nombre_reino` hasta que encontremos la cadena y sencillamente devolver el índice del vector (que es entero y en el rango del enumerativo, por lo que no hay inconveniente en castearlo a tipo `enum reino`). Ahora bien, van a surgir dos complicaciones: la primera es que no tenemos manera de preguntarle al enumerativo cuántas etiquetas posee (esto podría saltearse preguntándoselo al vector con `sizeof(nombres_reino)/sizeof(nombres_reino[0])`); y la segunda es que no sabríamos qué devolver en el caso de que la cadena `s` no coincidiera con ningún elemento de `nombres_reino`. Si bien ambos problemas pueden evadirse, probablemente convenga manejarlos de otra forma.

Tablas de búsqueda para traducir a enumerativo

En el caso de que supiéramos que lo que hay que traducir viene bien formado y ya pasó por un proceso previo de verificación que garantiza la correctitud de lo datos (suele ser una etapa recomendable en cualquier programa) bien podemos hacer el reverso que para traducir de enumerativo a otra cosa:

```
// Precondición: La cadena pertenece a nombre_reino
enum reino cadena_a_reino(const char *s) {
  size_t cantidad = sizeof(nombre_reino)/sizeof(nombre_reino[0]);
  for(size_t i = 0; i < cantidad; i++)
    if(!strcmp(nombre_reino[i], s))
      return i;
}
```

Notar que `i` es de tipo `size_t` y la función devuelve `enum reino`, no hay nada peligroso en este casteo porque por construcción estamos **asegurando** que `i` *entra* en un `enum reino`. Notar además que no

hay un `return` al final de la función, más allá de la precondición en una implementación real podríamos devolver un reino por omisión, por ejemplo `return ANIMALIA;`.

En el caso de que pueda haber una categoría no definida dentro de mi enumerativo, y esto sea algo plausible de ocurrir a tal punto de que *tenga sentido* incluir ese caso como un caso particular del enumerativo, podemos agregar una etiqueta más al enumerativo que represente el estado indefinido. (Preguntarse: ¿Necesito agregarla o debería resolver ese problema antes de asignar el enumerativo?)

Por ejemplo:

```
typedef enum {
  OPC_AYUDA, OPC_VERSION, OPC_RECURSIVO, OPC_TODOS, OPC_INDEFINIDA
} opcion_t;

const char opciones_letras[] = {'h', 'v', 'r', 'a'};
const char *opciones_cadenas[] = {"help", "version", "recursive", "all"};

opcion_t leer_opcion_letra(char opcion) {
    for(size_t i = 0; i < OPC_INDEFINIDA; i++)
        if(opciones_letras[i] == opcion)
            return i;
    return OPC_INDEFINIDA;
}
```

Notar que la tabla `letras_opciones` no define valor para `OPC_INDEFINIDA`, esto es porque esa no es una opción que se ingrese sino una condición de error. Notar que agregamos prefijos a las etiquetas de nuestro enumerativo, esto es importante porque dado que las etiquetas se definen internamente *exactamente igual* que si se hubiera escrito `#define OPC_AYUDA 0`, etc. no puede haber dos etiquetas con el mismo nombre en el programa, para evitar esto es preferible usar prefijos únicos para las etiquetas de un tipo.

Tablas de búsqueda para traducir cosas en cosas

A veces podemos tener que traducir un conjunto finito de cosas que están en un formato A en otro formato B. Un camino para realizar esto puede ser primero convertir de A en un enumerativo adecuado y luego convertir del enumerativo en B. También podemos omitir el paso intermedio sencillamente construyendo una tabla de búsqueda mediante el apareo de dos arreglos donde coincidan los índices de los elementos equivalentes.

Por ejemplo:

```
char *opcion_letra_a_cadena(char letra) {
  size_t cantidad = sizeof(opciones_letras)/sizeof(opciones_letras[0]);
  for(size_t i = 0; i < cantidad; i++)
    if(opciones_letras[i] == letra)
      return opciones_cadenas[i];
  return NULL;
}
```

Si bien podríamos pensar a `i` como un elemento de `opcion_t` la realidad es que ese fragmento funciona independientemente de la declaración del enumerativo.

Con este enfoque se pueden hacer múltiples arreglos asociados por el índice. Ahora bien los mismos quedan desacoplados en el código y es el programador el que los asocia. Otra manera de resolver el problema es implementando un vector que ya contenga los pares a traducir:

```
struct opcion {
  char letra;
  char *cadena;
```

```
};

const struct opcion opciones[] = {
  {'h', "help"},
  {'v', "version"},
  {'r', "recursive"},
  {'a', "all"}
};

char *opcion_letra_a_cadena(char letra) {
  size_t cantidad = sizeof(opciones)/sizeof(opciones[0]);
  for(size_t i = 0; i < cantidad; i++)
    if(opciones[i].letra == letra)
      return opciones[i].cadena;
  return NULL;
}
```

Tablas de búsqueda para traducir enumerativos en acciones

Si quisiéramos realizar acciones diferentes en función de los valores de un enumerativo una de las maneras sería implementando un `switch` donde cada `case` *atienda* una (o varias) de las etiquetas del enumerativo y realice una acción. Si bien esto no está mal puede pasar que el `switch` quede como un bloque monolítico de muchas líneas de código que hacen cosas disímiles entre sí.

Otra forma de resolver esto es implementando un vector de punteros a función asociado a la resolución de cada una de las opciones, es decir:

```
// Manejadores de las acciones:
bool accion_ayuda();
bool accion_version();
bool accion_recursivo();
bool accion_todos();

typedef bool (*manejador_t)(void);

bool ejecutar_accion(opcion_t opcion) {
  manejador_t manejadores[] = {
    accion_ayuda, accion_version, accion_recursivo, accion_todos
  };

  if(opcion == OPC_INDEFINIDA)
    return false;

  return manejadores[opcion]();
}
```

En este caso se cambia el enfoque monolítico por un enfoque modular con funciones que saben hacer cada acción. Este esquema *requiere* que todas las funciones tengan la misma firma. En el caso de que las funciones por construcción necesiten parámetros no homogeneos puede forzarse en la interfaz que los mismos se enmascaren de una forma homogenea (como por ejemplo pasándolos como `void *`). También pueden declararse tablas auxiliares que contengan más información, como por ejemplo el número de los argumentos de cada manejador (o declarar una única tabla que encapsule toda esa información en un `struct`). En el caso de que esto sea demasiado engorroso, se recomienda implementar un `switch` y no hacer uso de tablas de búsqueda.