Abstracciones, Listas Enlazadas, y For

Esp. Ing. José María Sola, profesor.

Revisión 1.0.0
Abr 2017

Tabla de contenidos

1

Introducción

En este artículo vamos a tratar los siguientes temas, tanto en el lenguaje C como en en C++, aunque varios conceptos aplican a otros lenguajes de programación:

- Nodos y listas enlazadas.
- Uso de forward declaration.
- `struct` y `typedef`.
- `auto` en C++.
- Namespasces (espacios de nombnre).
- Recursividad.
- Teorema fundamental de la ambigüedad.
- for versus `while`.

Los temas los vamos a tratar con un caso de estudio simple:

> *Impresión del contenido de una lista enlazada de enteros.*

2

Implementación Clásica

Del punto de vista de *estructura de datos*, la implementación clásica se basa en la declaración de la estructura `Node` con un un valor entero `val` y un puntero al siguiente nodo `next`. A nivel *algorítmico*, se imprime el valor y se sigue el puntero al siguiente nodo hasta que el siguiente sea nulo.

2.1. Estructura de Datos

```
typedef struct Node {
 int val;
 struct Node *next;
} Node;
```

Notamos que la declaración es autoreferencial, aunque no recursiva. En la misma declaración de `Node`, inclusive antes de finalizar la declaración, utilizamos `Node` para declarar un puntero a un `Node`. Esta declaración es semánticamente correcta porque para declarar un puntero no es necesario conocer la composición del tipo que apuntado por el puntero. Esta característica del lenguaje se llama *declaración adelantada*. Por otro lado, la declaración recursiva *sí* es un error semántico.

```
typedef struct Node {
 int val;
 struct Node next; // Semantic error :(
} Node;
```

En C las estructuras tienen su propio espacio de nombres, por eso, para utilizarlas es necesario prefijarlas con `struct`. Para evitar ese prefijo utilizamos `typedef`, que permite declarar `Node` como sinónimo de `struct Node`. Es importante

destacar que los `tag` o nombres de las estructuras son opcionales, es decir, podemos tener *estructuras anónimas*, pero en nuestro caso es necesaria nombrarla para declarar el miembro `next`, que es autoreferencial. Por último, notamos que la declaración declara varias entidades:

- el nombre de la estructura: `Node`;
- sus dos datos miembro: `val` y `next`; y
- y `Node` como sinónimo de `struct Node`.

Otra opción es declarar la estructura `Node` en una declaración separada del `typedef`, en este caso particular, es simplemente una cuestión de estilo, en otros casos no:

```
struct Node {
 int val;
 struct Node *next;
};
typedef struct Node Node;
```

En C++ las estructuras no tienen su propio espacio de nombres, por eso, para referenciarlas no es necesario prefijarlas con `struct`, lo cual hace innecesaria la declarción `typedef`.

```
struct Node {
 int val;
 Node* next;
};
```

2.2. Algoritmo

La repetición la logramos con la *estructura de control de flujo de ejecución* **iteración**, en particular la *sentencia while*. La variable `n` es un puntero al primer `Node`, la variable auxiliar `a` también es del tipo puntero a `Node`, pero la calificamos `const` para que el compilador verifique que no modifiquemos el contenido apuntado por `a`.

```
const Node *a=n;
while(NULL != a){
 printf("%d ", a->val);
```

```
 a=a->next;
}
```

La versión en C++ tiene algunas diferencias, usamos:

- `auto` para inferir el tipo sin tener que conocerlo.
- `nullptr` en vez de `NULL`.
- `cout` y el operador `<<` en vez de `stdout` y `printf`, respectivamente.

```
auto a=n;
while(nullptr != a){
 std::cout << a->val << ' ';
 a=a->next;
}
```

En ambos lenguajes, podemos simplificar la expresión de control de la *sentencia while* a `while(a)`, ya que `NULL` y `nullptr` valen `0`.

2.2.1. Recursividad

Aunque la declaración no es recursiva, la naturaleza autoreferencial de `Node` permite implementar la repetición que requerie `Print` con invocación recursiva:

```
void Node_Print(const Node *n){
 if(NULL == n)
  return;
 printf("%d ", n->val);
 Node_Print(n->next);
}
```

La versión recursiva no posee efecto de lado y es, subjetivamente, más legible, dada la autoreferencia de `Node`. Objetivamente, es menos eficiente en tiempo y espacio, aunque algunos compiladores pueden optimizar este tipo de recursividad.

C++ presenta la facilidad *namespaces*, que nos permite evitar nombrar las funciones `Node_Print` y utilizar el operador de alcance `Node::Print` o simplemente `Print` si estamos usando el espacio de nombre `Node`. Más allá de eso, no hay diferencias substanciales:

```
void Print(const Node* n){
 if(nullptr == n)
  return;
 std::cout << n->val << ' ';
 Print(n->next);
}
```

3

¿Puntero a Nodo o Lista?

De la implementación anterior surge un problema: dada la declaración `Node *p;` podemos tratar a `p` por lo que objetivamente es, *un puntero a una estructura Node*, o podemos tratarlo como una *lista enlazada*, mas particularmente, el *comienzo de una lista enlazada*. Para el caso especial `p` igual a `NULL`, podemos interpretar a `p` como un *puntero nulo* ó como una *lista enlazada vacía*.

Esta ambigüedad surge por la falta de abstracción, al tratar con punteros a nodos estamos manejando una *estructura concreta*, en nuestro caso *lista simplemente enlazada de nodos de enteros* y no una *estructura abstracta*, en nuestro caso *lista de enteros*.

En *How to Think Like a Computer Scientist* [HTTLACCCPP] se le da un nombre a esta situación:

> The fundamental ambiguity theorem[1] theorem describes the ambiguity that is inherent in a reference to a node: A variable that refers to a node might treat the node as a single object or as the first in a list of nodes.
>
> — Allen B. Downey *How to Think Like a Computer Scientist*

Es decir, podemos considerar una referencia a un nodo como un nodo independiente o como el primero de una lista de nodos.

[1] http://www.openbookproject.net/thinkcs/archive/cpp/english/chap18.htm#6

4

Solución: Tipo Lista

La solución a la ambigüedad que propongo es la aplicación de *abstracción de datos* y de *ocultamiento de información*. La *abstracción de datos* la aplicamos con la construcción de la abstración *List*, y el *ocultamiento de información* con la declaración de `Node` en un lugar no accesible por los consumidores de la abstracción *List*. Para usar la abstracción *List* no es necesario saber que se implementa con `Node`, es más, esa información no se refleja en la interfaz en las declaraciones de las funciones y, adicionalmente, es inaccesible para los consumidores de *List*.

La abstracción la construimos con de la defínción de la *interfaz* o *parte pública de la implementación* y la *parte privada de la implementación*.

4.1. En C

En la interfaz se declara pero no se define la estructura `Node`. `Node` solo se utiliza para declarar el único dato miembro de `List`: `first`. La declaración de la estructura `Node` se hace durante la declaración del dato miembro `first`:

List.h.

```
typedef struct List {
 struct Node *first;
} List;
```

En la parte privada de la implementación se define la estructura `Node`:

List.c.

```
typedef struct Node {
```

```
 int val;
 struct Node* next;
} Node;
```

- ¿Qué ventajas y desventajas tiene la abstracción `List` por sobre el simple manejo de `Node`?

El siguiente paso en la evolución de la abstracción es ocultar la estructura `List`. Podemos hacerlo mediante la técnica conocida como *tipo de dato opaco*, porque no podemos ver "dentro del tipo"; la técnica también se la conoce como *pimpl*, porque el único dato miembro es un ***p**untero a la **impl**ementación*.

- La construcción la dejamos como ejercicio.

4.2. En C++

En C++ podemos aprovechar la *incialización por defecto* para. Para un nueva lista vacía, hacemos que `first` apunte `nullptr`.

List.h.

```
namespace List{
 struct List {
  struct Node* first = nullptr;
 };
```

List.cpp.

```
namespace List{
 struct Node {
  int val;
  Node* next;
 };
```

5

Iterar la Lista Enlazada con la Sentencia For

En las versiones iterativas presentadas, usamos el siguiente patrón algorítmico:

- Inicializar el puntero iterador.
- Iterar mientras el puntero no es nulo:
  - Ejecutar la sentencia que envía el valor a la salida estándar.
  - Actualizar el puntero al siguiente nodo.

En esa solución aplicamos la *sentencia while* de la siguiente forma en C:

```
const Node *a=n;
while(NULL != a){
 printf("%d ", a->val);
 a=a->next;
}
```

y en C++:

```
auto a=n;
while(nullptr != a){
 std::cout << a->val << ' ';
 a=a->next;
}
```

Recordemos la sintaxis de la *sentencia while*, y anlicemos la solución semánticamente:

```
while( expresión ) sentencia
```

1. *sentencia expresión* que inicializa del puntero al primer nodo.
2. *sentencia iteración* `while`, con:
    i. *expresión* de control sobre el puntero.
    ii. *sentencia compuesta* formada por:
        A. *sentencia expresión* que envía el valor a la salida estándar.
        B. *sentencia expresión* con la actualización del puntero al siguiente nodo.

Notamos que la *sentencia while* sola no es suficiente, debemos primero escribir una *sentencia expresión* para incializar el puntero.

Como el patrón algorítmico es tan común, el lenguaje incorpora la *sentencia for*, que es más general que la *sentencia while*:

```
for( expresión1 ; expresión2 ; expresión3 ) sentencia
```

También incluye la variante donde la primera expresión es reemplazada por una declaración:

```
for( declaración expresión1 ; expresión2 ) sentencia
```

Por lo tanto, si reemplazamos la solución basada en `while` por la nueva basada en `for` obtenemos una versión nueva más compacta y que introduce un nuevo *idiom* (*expresión idiomática*) similar al utilizado para recorrer arreglos; hasta utilizamos la varibale `i` para iterar. En C:

```
for(const Node *i=n; i; i=i->next)
 printf("%d ", i->val);
```

y en C++:

```
for(auto i=n; i; i=i->next)
 std::cout << i->val << ' ';
```

6

Síntesis

Ahora, reunamos todos los conceptos en la función `Print` que recibe un `List` en vez de un `Node`, e itera la lista con `for` en vez de con `while`.

List.c.

```
void List_Print(const List *l){
 for(const Node *i=l->first; i; i=i->next)
  printf("%d ", i->val);
}
```

List.cpp.

```
void Print(const List& l){
 for(auto i=l.first; i; i=i->next)
  std::cout << i->val << ' ';
}
```

7

Bibliogrfía

How to Think Like a Computer Scientist es una familia de libros abiertos, hay varias ediciones para diferentes lenguajes de programación.

- [HTTLACCCPP] Allen B. Downey. **How To Think Like A Computer Scientist: Learning with C++**. 2001. http://www.openbookproject.net/thinkcs/archive/cpp/english/
- [HTTLACCPY] Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers. **How to Think Like a Computer Scientist: Learning with Python 3 (RLE)**. Oct 2012. http://openbookproject.net/thinkcs/python/english3e/. Versión interactiva: http://interactivepython.org/runestone/static/thinkcspy/index.html.