MODULO 7

Estructuras de datos dinámicas

Breve descripción

En este documento se explica qué es la memoria dinámica, cómo se define y cómo se utiliza.

Versión 1.1

Martin Jerman
Martin.jerman@inspt.utn.edu.ar

Introducción

Anteriormente hemos visto cómo trabajar con la memoria estática en cualquier nivel de dimensiones (vectores y matrices). Pero la desventaja de la memoria estática es que es justamente, estática. No se puede redimensionar, lo que nos obliga a una de dos cosas: 1) a saber de antemano la cantidad de datos a procesar para poder almacenarlos o 2) pensar en el peor caso posible en términos de cantidad de datos para poder procesar o 3) establecer un límite a nivel programa para poder procesar los datos.

Para poder crear programas más genéricos y que no tengan limitaciones de memoria, existe la memoria dinámica. ¿Cuál es la idea? La idea es tener un puntero sin apuntar al que le asignaremos un espacio de memoria. Es decir que crearemos vectores "on demand". Para ello, tenemos 4 funciones dedicadas.

Malloc()

Esta función reserva un espacio de memoria de un tamaño definido en bytes que se pasa por parámetro. Este es su prototipo:

```
void* malloc(size_t tamaño);
```

Veamos un ejemplo:

```
int *ptr, i;
ptr=(int*)malloc(sizeof(int));
*ptr=7;
printf("ptr=%d\n",*ptr);
```

Este caso parece muy trivial. Pero dado que el valor que se le pasa por parámetro es un entero, podemos reservar un espacio de memoria más amplio. Veamos cómo:

```
int *vec,i;
vec=(int*)malloc(DIM * sizeof(int));
for (i=0;i<DIM;i++)
    vec[i]=7;
```

En las líneas anteriores utilizamos malloc para reservar espacio suficiente para un vector de tamaño DIM de enteros.

Calloc()

Esta función es similar a la anterior, asigna el número especificado de bytes y los inicializa a cero. Pero en sus parámetros, esta función recibe dos:

- El número de elementos a ubicar
- El tamaño de cada elemento en bytes

Con esto, calloc() es ideal para crear vectores. El prototipo es el siguiente:

```
void* calloc (size_t num, size_t size);
```

En un ejemplo:

```
int *vec;
```

```
vec=(int*)calloc(DIM,sizeof(int));
for (i=0;i<DIM;i++)
    vec[i]=8;
```

en el ejemplo, creamos un vector de tamaño DIM donde cada elemento tiene el tamaño de un entero. Esto suena trivial en este momento, peso si necesitamos un vector para guardar estructuras, este parámetro deberá indicar el tamaño de la estructura. Por ejemplo:

```
vec=(int*)calloc(DIM,sizeof(miEstructura));
```

Realloc()

Esta función sirve para redimensionar el vector a un tamaño mayor o menor. Su prototipo es:

```
void *realloc(void *ptr, size_t nuevo_tamaño);
```

como se ve, en sus parámetros se requiere:

- el puntero al espacio de memoria que se desea redimensionar
- el nuevo tamaño en bytes. Si este es menor al actual, se reducirá (y por ende se perderán datos); y si es mayor se agrandará.

Veamos un ejemplo:

```
int *vec,*vec2,i,dimNueva;
/* creamos un vector */
vec=(int*)calloc(DIM,sizeof(int));
for (i=0;i<DIM;i++)
      vec[i]=8;
/* redimensionamos el vector */
dimNueva=DIM*3;
vec2=(int*)realloc(vec, dimNueva * sizeof(int));
/* mostramos el vector */
for (i=0;i<dimNueva;i++)
      printf("%d-",vec2[i]);
/* asignamos valores para operar */
for (i=0;i<dimNueva;i++)
      vec2[i]=10+i;
```

En el ejemplo, utilizamos el vector creado y apuntado por `vec` a tres veces su tamaño y lo apuntamos a `vec2`.

Free()

Esta función es fundamental utilizarla siempre. Dado que las funciones anteriormente vistas sirven para pedir memoria dinámicamente al sistema operativo, la memoria pedida queda asignada al proceso mientras se esté ejecutando. De esta forma, si las funciones de pedido de memoria se encuentran en un ciclo, puede saturarse la memoria del equipo.

Para evitar que haya memoria pedida dinámicamente y que no se este utilizando, existe la función free. Esta función libera la memoria apuntada por un puntero.

Un ejemplo:

```
int *vec,i;
vec=(int*)calloc(DIM,sizeof(int));
for (i=0;i<DIM;i++)
       vec[i]=8;
/* valor contenido en la posicion cero */
printf("vec=%d\n",vec[0]);
free(vec);
/* valor apuntado despues de ser liberado el espacio */
printf("vec=%d\n",vec[0]);
```

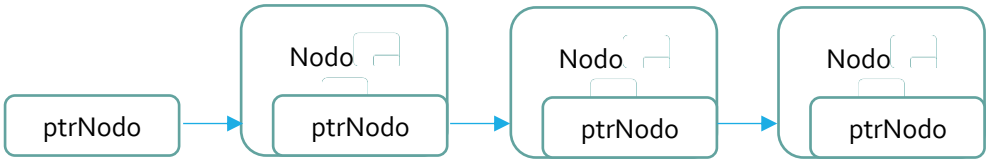
Si ejecutan estas funciones, verán que el valor mostrado en el último `printf` es absurdo. Eso es porque el espacio de memoria ya no está asignado y por ende contiene basura.

Uso avanzado: listas, pilas y colas

Habiendo visto cómo es posible pedir memoria dinámicamente,

- ¿Qué ocurriese si un puntero fuese parte de una estructura?
- ¿Y qué ocurriese si dicho puntero que es un campo de la estructura apunta a otra estructura de su mismo tipo?

Pues bien, de esto se tratan las listas, pilas o colas: de estructuras llamadas "nodos" en la que uno de sus campos es un puntero a otro nodo.



De este modo la idea es crear nodos que contengan la información que deseemos y engancharlos en la lista, como si se tratase de un eslabón de una cadena.

Veamos un ejemplo de declaración:

```
typedef struct nodo {
    int dato;
    struct nodo* sgte;
} nodo;
```

Como se ve en el ejemplo, la definición de la estructura es sutilmente diferente. Primero, el nombre se repita tanto en la parte inicial de la declaración como en la final. Esto es porque uno de los campos de la propia estructura es del mismo tipo de dato que la propia estructura. Es como si fuera una **definición recursiva**.

Para las listas, pilas o colas, se puede tener cualquier cantidad de campos. El único requerimiento es tener al menos un campo puntero a sea del mismo tipo de dato que la propia estructura para poder enlazar el **siguiente nodo**.

Dado que manejar estas estructuras puede ser algo confuso al principio, se sugiere trabajar con funciones bien simples y especificas para cada necesidad. Por ejemplo, para crear un nodo o eslabón con un dato, podemos definir la siguiente función:

```
nodo* crearNodo(int dato){
    nodo* unNodo;
    unNodo = (nodo*)malloc(sizeof(nodo));
    unNodo->dato = dato;
    unNodo->sgte = NULL;
    return unNodo;
}
```

Analicemos la función:

1- La función recibe por parámetros todos los datos que irán en el nodo. En este caso un entero.
2- El nodo que se crea es una variable local puntero al nodo que se creara. Debe ser asi para que dicho espacio de memoria no se pierda terminar la ejecución de la función.
3- Para crear el nodo, se utiliza la función `malloc()` como se muestra en el ejemplo
4- Por cuestiones de prolijidad y evitar futuros problemas, **se deben completar todos los campos** del nodo, especialmente el campo siguiente (o `sgte`) con **NULL**.
5- La función retorna el nodo cargado. A pesar de ser una variable local, por estar el espacio de memoria pedido en tiempo de ejecución con `malloc()`, este no se pierde al terminar de ejecutar la función. Por ende, sigue siendo utilizable.

Una vez creado el nodo, resta engancharlo en la lista. Para ello hay que tener en cuenta que los punteros son referencias a memoria que, si se reasignan, la referencia al espacio de memoria anterior se pierde y por ende no se puede recuperar.

Dicho esto, veamos el ejemplo siguiente:

```
int main()
{
    nodo *lista=NULL, *aux, *nuevo;
    int i;

    aux=lista;
    puts("Cargando lista...");
    for (i=0; i<5 ; i++){
        nuevo = crearNodo(i*7);
        printf("Nodo valor: %d\t",nuevo->dato);
        if (lista == NULL){
            puts("Primer elemento de la lista agregado");
            lista=nuevo;
        }
        else{
            puts("Agregando elemento a la lista");
            aux=lista;
            while (aux->sgte!=NULL)
                aux=aux->sgte;
            aux->sgte = nuevo;
        }
    }

    puts("Mostramos la lista");
    aux = lista;
    for (i=0; aux != NULL ;i++){
        printf("Elemento %d) %d\n",i,aux->dato);
        aux=aux->sgte;
    }
    return 0;
}
```

En el ejemplo, tenemos dos partes:

1- Código que genera la lista
2- Código que muestra la lista

Generando la lista

Para generar la lista, primero generamos el nodo con el dato necesario. Para este caso un ciclo que genera valores a agregar a la lista utilizando la función `crearNodo()` anterior para generar el nodo.

Luego hay que agregarlo a la lista. En este caso, se desea agregar el nodo al final de la lista. Para hallar el final de la lista tenemos dos opciones, o la lista esta vacía (que valga NULL) o que el ultimo nodo tenga en el campo siguiente un NULL. Estos dos casos hay que contemplarlos siempre, como se ve en el código. Además, hay que tener en cuenta que si recorremos la lista con la propia variable lista que apunta al primer nodo de esta, se perdería la referencia al nodo y por ende a toda la lista. De ahí que siempre se utiliza una variable auxiliar para recorrer la lista. Una vez que se llega al nodo que tiene NULL en el campo siguiente, se asigna el nuevo nodo (que ya tiene el NULL también en el campo siguiente porque así lo creamos).

Mostrando la lista

Para mostrar la lista, el algoritmo es mas sencillo. Basta con asignar el valor de la lista a una variable auxiliar y recorrer la misma con ciclo hasta que dicha variable valga NULL.

```
aux = lista;
for (i=0; aux != NULL ;i++){
    printf("Elemento %d) %d\n",i,aux->dato);
    aux = aux->sgte;
}
```

Como se dijo anteriormente, si se utiliza una variable auxiliar, no se corre riesgo de perder la lista original.

Diferencias entre listas, pilas y colas

Ahora bien, cual es la diferencia entre las listas, pilas y colas. Desde un punto de vista técnico, no hay diferencias. En los tres casos se utiliza una estructura que tiene un campo "siguiente" que apunta a un dato del mismo tipo que la propia estructura. ¿Dónde está la diferencia? En el cómo se trabaja.

Pilas

Estas estructuras se trabajan como una lista FILO (First In, Last Out). Esto quiere decir que el primer elemento que se agrega a la pila es el último que sale. Lo que significa que siempre se agrega un nodo al final y cuando se saca un nodo, se saca el primero. O lo que es lo mismo, se agrega un nodo al principio y siempre se saca el primero. Como analogía podemos utilizar la pila de platos: si necesito uno, tomo primero el que está más arriba de todos, que fue el último en ser apilado.

Con estas condiciones, las funciones de `agregarNodoPila`() o `quitarNodoPila`() deben respetar este esquema.

Colas

Como en el caso anterior, la diferencia no es técnica sino lógica. Las colas siguen el sistema FIFO (First In, First Out), es decir que el primero que entra, es el primero que sale. De esta forma al agregar nodos siempre al final, cuando se saquen será desde el principio de la cola. Análogamente a lo que ocurre en la fila de un supermercado.

Con esto la función `agregarNodoCola`() agregara al final de la cola, mientras que el `quitarNodoCola`() quitara desde el principio.

Listas

A diferencia de las anteriores, las listas no tienen restricciones a nivel lógico. Es decir que se puede agregar nodos al principio, al medio o al final según se desee. Lo único que se debe tener en cuenta

aquí al agregar en medio es que se deben reapuntar todos los nodos para que el nodo nuevo quede bien enlazado sin perder referencia a la lista. Lo mismo ocurre al borrar, si se borra un nodo, hay que reapuntar los nodos que quedan en la lista para que siga siendo consistente.