Lista de errores comunes encontrados en laboratorios anteriores

Índice

1 Introducción

En este documento se intenta resumir diversos errores encontrados en laboratorios de años anteriores así como varios consejos de buenas practicas de programación y estilo.
Los distintos errores presentados aquí deberían ser claros al momento de implementar el obligatorio, de no ser así se recomienda consultar en clase ya que cada uno de estos errores está asociados con distintos errores conceptuales.

2 Memoria

2.1 Asignación de punteros

Un puntero es una variable que contiene la dirección de memoria de otra variable. Los punteros brindan gran flexibilidad al momento de programar pero deben utilizarse en forma ordenada para evitar problemas difíciles de encontrar. En esta sección describimos algunos errores comunes que pueden cometerse al utilizarlos. (Asumimos que el lector conoce el lenguaje de programación Módula II y en particular el manejo de punteros que ofrece)

2.1.1 Problemas con alias

Cuando a un puntero se le copia el valor de otro puntero, ambos referencian a la misma zona de memoria y se dice que uno es un alias del otro. Esta característica puede ser muy útil cuando se utiliza concientemente pero puede provocar efectos inesperados si se hace inadvertidamente. Observe el siguiente ejemplo

```
VAR p1,p2: POINTER TO CARDINAL;
      x: CARDINAL;

NEW (p1);
p2 := p1;
…..
p1^ := 2;
p2^ := 5;
x := p1^ * p2^;
```

La última asignación da el valor 25 a la variable x.

2.1.2 Inicialización

Un problema repetido en al inicialización de los punteros es el siguiente: está mal hacer NEW de una variable para inmediatamente después asignarle *NIL* u otro puntero.
Los ejemplos típicos son:

- al crear una lista vacía, hacer NEW de la variable para luego asignarle *NIL*.
- teniendo un lista L crear una lista auxiliar que compartan memoria (alias) hacer NEW de la variable para luego asignarle L.

```
Procedure
funcion():LNAT
VAR Temp:LNAT;
Begin
…….
i)    NEW(Temp);
ii)   Temp:=NIL;
……..
End
```

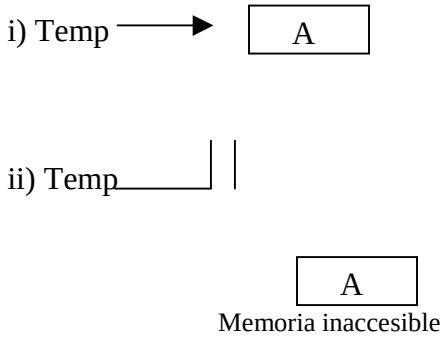


Fig. Función **con error.**

En otras ocasiones se asigna *NIL* a un puntero **sin existir necesidad**, por ejemplo:

- antes de hacerle *NEW*
- antes de asignarle otro puntero
- antes de retornar en un procedimiento.

2.2 Liberación de memoria

2.2.1 Omisión

Un tipo muy común de **error** es omitir la liberación de memoria reservada. En casos donde esto ocurre numerosas veces o con grandes volúmenes de memoria, puede provocarse el enlentecimiento del sistema e incluso agotarse la memoria disponible provocando la falla del programa.

Tenga en cuenta que una vez que se "pierde el rastro" de una zona de memoria reservada, ya no es posible liberarla. Por ejemplo si se cuenta con la siguiente definición de una lista

```
TLista = POINTER TO TNodo
TNodo = RECORD
         info = CARDINAL;
         sig  = TLista;
        END;
```

Y en determinado momento se trunca la lista en un nodo asignando (`pNodo^.sig := NIL;`) la memoria reservada para los nodos que seguían a pNodo en la lista ya no será posible de liberar, salvo que el valor de pNodo^.sig hubiera sido copiado a otra variable antes de asignarle NIL.

2.2.2 Repetición

Si bien no liberar memoria reservada puede ocasionar problemas, liberarla más de una vez es igualmente perjudicial. Es común que esto ocurra inadvertidamente cuando se trabaja con alias como en el siguiente ejemplo

```
NEW (p1);
p2 := p1;
…..
```

Luego de esta inicialización, la memoria reservada puede liberarse indistintamente mediante `DISPOSE(p1)` o `DISPOSE(p2)`, **pero no ambas**.

2.2.3 Incompleta

En general un TAD es implementado utilizando otros ya desarrollados, por lo tanto es necesario al momento de implementar la función destructor, llamar a los destructores de los TAD “hijos”.

2.3 Acceso de variables

Al igual que otras variables, las de tipo puntero deben ser inicializadas antes de ser utilizadas. Un puntero no inicializado contendrá una dirección de memoria no determinada a priori y dereferenciarlo puede provocar que el programa cancele la ejecución por una excepción de violación de acceso a memoria o tenga un comportamiento inesperado.
Considere el siguiente fragmento de programa de ejemplo donde los punteros p1 y p2 se utilizan sin inicialización previa.

```
VAR p1,p2: POINTER TO CARDINAL;
     a: CARDINAL;
BEGIN
 a := 2*p1^;
 p2^ := a;
 ….
```

Cuando ocurre la primera asignación, si p1 apunta a un área de memoria que no fue asignada a nuestro programa por el sistema operativo, ocurre una excepción de violación de acceso a memoria y la ejecución se interrumpe inmediatamente con un mensaje de error. Si casualmente la dirección de memoria contenida en p1 pertenece al espacio de direcciones disponibles para el programa, la ejecución continúa pero el valor asignado a la variable a es impredecible lo cual podría ocasionar un comportamiento inesperado del programa.
Cuando ocurre la segunda asignación, nuevamente si la dirección contenida en p2 no pertenece al espacio disponible para el programa, la ejecución se cancela por una violación de acceso a memoria. En el caso en que dicha dirección sí está disponible para el uso, el efecto puede llevar a comportamientos aún más difíciles de explicar que en el caso anterior. Al modificar un área de memoria cualquiera sin intención, podríamos alterar el valor de variables que no tienen ninguna relación con p2 ocasionando más tarde el comportamiento erróneo de porciones de código que no contienen ningún error. Incluso podríamos alterar el código compilado en lenguaje de máquina, que también se encuentra almacenado en memoria, provocando por supuesto errores de ejecución.

Problemas similares pueden ocurrir cuando se desreferencian punteros con valor `NIL` o punteros que apuntan a memoria que ha sido liberada mediante `DISPOSE`. Considere el siguiente ejemplo

```
……
NEW (p1);
NEW (p2);
…..
DISPOSE (p1);
DISPOSE (p2);
…..
a := 2*p1^;
p2^ := a;
```

Las últimas asignaciones de este ejemplo puede generar el mismo tipo de comportamiento que en el anterior. Esto es particularmente común cuando se trabaja con alias como en el siguiente ejemplo.

```
NEW (p1);
p2 := p1;
…..
DISPOSE (p1);
…..
p2^ := 1;
```

3 Comentarios

Es una buena práctica de programación comentar correctamente los procedimientos y funciones implementadas. Por comentar correctamente se entiende dar una breve descripción de cual es el cometido de la función, así como las precondiciones y poscondiciones que ésta requiera.
Además es importante comentar aquellas secciones cuya lógica no sea fácil de entender a simple vista.
Puede ser de gran utilidad mantener como comentario la ultima fecha de modificación de las funciones o archivos.
Es una mala práctica de programación entregar un producto con código comentado entre sentencias de código útiles, sin explicar lo que estas significan o para que serviría en un futuro.

4 Indentación

Para mayor claridad en los programas se debe tener una buena indentación del código haciendo que éste sea legible y permita la corrección de errores lógicos rápidamente.
Una mala indentación lleva a que un programa sea difícil de entender tanto para los que hicieron el programa como aquellos que no.
Es de notar, que en ámbitos de desarrollo reales donde los problemas son de gran escala es fundamental que otras personas puedan entender su código.

5 Nombres de variables

Otro practica que puede ser de gran ayuda al momento de entender la lógica de los programas es la utilización de nombres de variables que se correspondan con la idea que abstrae cada una de ellas. Por ejemplo, no es una buena práctica de programación llamar a todas las variables auxiliares aux1, aux2, …., sobre todo si estas son de distintos tipos.

6 En general

Cuidar de no repetir más de una vez las mismas preguntas en caso de no ser necesario:

```
PROCEDURE CopiaLimpia ( original : Ejemplo; VAR copia : Ejemplo);
BEGIN
     Vacio(copia);
     IF NOT (original = NIL) THEN
          WHILE original # NIL DO
               Agregar(original^.valor, copia);
               original := original^.sig;
          END;
     END;
END CopiaLimpia;
```

En este ejemplo es claro que el IF es redundante.

Cuidar también no generar lógicas mucho más complicadas de lo que podrían ser:

```
PROCEDURE CopiaLimpia(original: Ejemplo;VAR copia: Ejemplo);
     VAR aux: Ejemplo;
BEGIN
     IF (original#NIL)THEN
          NEW(aux);
          aux^.resto:=NIL;
          copia:=aux;
          WHILE (original#NIL) DO
               CopiaLimpiaRac(original^.elem,aux^.elem);
               original:=original^.resto;
               IF (original#NIL)THEN
                    NEW(aux^.resto);
                    aux:=aux^.resto;
                    aux^.resto:=NIL;
               END;
          END
     ELSE
          copia:=NIL;
END;
```

Esto claramente repite preguntas y es un tanto intrincado. Se podría reestructurar el código para hacerlo más legible y más eficiente:

```
PROCEDURE CopiaLimpia(original:Ejemplo;VAR copia:Ejemplo);
     VAR aux: Ejemplo;
BEGIN
     copia:=NIL;
     WHILE (original#NIL) DO
          NEW(aux);
          CopiaLimpiaRac(original^.elem,aux^.elem);
          aux^.resto:=copia;
          copia:=aux;
          original:=original^.resto;
     END;
END;
```

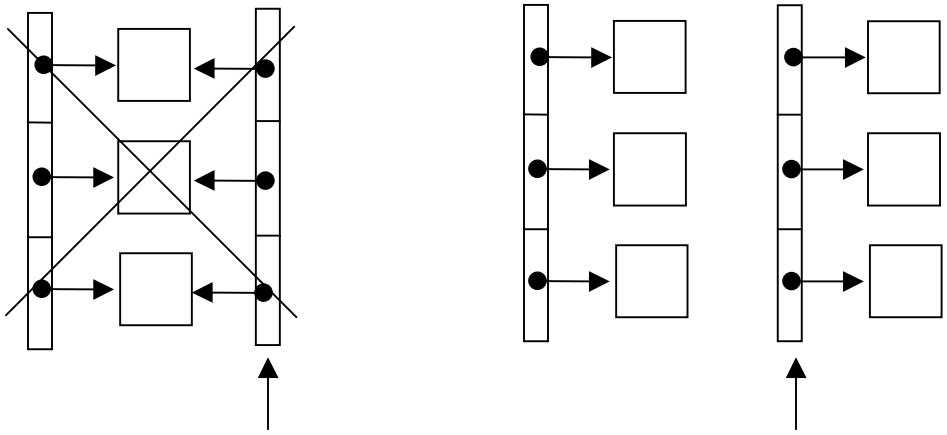
Cuando se maneja un Tipo Abstracto **la única manera de manipularlos** es con las operaciones que brinda su módulo de definición. No se debe hacer lo siguiente:

```
IMPLEMENTATION MODULE Ejemplo;
…….
VAR ej1,ej2: Ejemplo;
…….
IF  ej1 = ej2 THEN
…….
```

Esto se debería realizar con una función brindada por el TAD Ejemplo, en este caso que implemente la igualdad.
Otros dos casos importantes son los operadores de asignación y liberación de memoria sobre TAD desde un modulo externo. Es un **error común** hacer *NEW(ej1)* desde el modulo PadreEjemplo. En este caso se debería llamar a algún constructor del TAD Ejemplo, el caso de la liberación de memoria es análogo.

Por último, todo TAD que se implemente utilizando memoria **tiene que poseer** un operador de “copia limpia”, esto es que genere una copia de un ejemplar del TAD sin compartir memoria. Es de especial atención que cuando el TAD en mención utiliza otro TAD (digamos TAD2) para su implementación no solo deberá duplicar la memoria de la información propia sino que también deberá utilizar el operador de

copia limpia del TAD2. En la figura siguiente, parte a se puede observar un ejemplo en el que la copia limpia solo duplica en forma parcial la información.



7 Entrada / Salida

En muchas ocasiones, se realizan pequeños errores o modificaciones a las especificaciones del laboratorio en la Entrada / Salida, que conllevan a la mala corrección del mismo, principalmente con la corrección automática. A continuación se enumeran los más comunes:

- Agregan mensajes: de bienvenida, de salida, etc.
- No se respetan los mensajes de salida que fueron especificados.
- Se incluyen lecturas no especificadas. Evidentemente este error es inducido por el trabajo desde el Entorno Integrado de Desarrollo. Al ejecutar el programa desde el entorno, aparece la confusión de que "se pierde la salida".