ESTRUCTURAS DE DATOS Y ALGORITMOS

TECNÓLOGO EN INFORMÁTICA

LISTA DE ERRORES COMUNES ENCONTRADOS EN LABORATORIOS ANTERIORES.

1. INTRODUCCIÓN

En este documento se intenta resumir diversos errores encontrados en laboratorios de años anteriores así como varios consejos de buenas prácticas 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.

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 conscientemente pero puede provocar efectos inesperados si se hace inadvertidamente. Observe el siguiente ejemplo:

```
int *p1, *p2;
int x;

p1= new int;
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 **NULL** u otro puntero.
Los ejemplos típicos son:

- Al crear una lista vacía, hacer **new** de la variable para luego asignarle **NULL.**
- Teniendo una lista **L** crear una lista auxiliar que compartan memoria (alias) hacer **new** de la variable para luego asignarle **L.**

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

- Antes de hacer **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:

```
struct Lista {
            Elemento valor;
            Lista *sig;
};
```

Y en determinado momento se trunca la lista en un nodo asignando (`pNodo->sig = NULL`) 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 copiada a otra variable antes de asignarle **NULL**.

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:

```
p1= new int;
p2 = p1;
…
```

Luego de esta inicialización, la memoria reservada puede liberarse indistintamente mediante `delete p1` o `delete 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 a 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 desreferenciarlo 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.

```
int *p1, *p2;
int a;

a= 2 * *p1;
*p2 = a;
...
```

Cuando ocurre la primera asignación, si **p1** apunta a una 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 **NULL** o punteros que apuntan a memoria que no ha sido liberada mediante `delete`. Considere el siguiente ejemplo:

```
p1 = new int;
p2 = new int;
...
delete p1;
delete p2;
...
a= 2 * *p1;
*p2 = a;
```

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

```
p1 = new int;
p2 = p1;
...
delete 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 cuál 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 tan 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 identación del código haciendo que éste sea legible y permita la corrección de errores lógicos rápidamente.
Una mala identación 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

Otra práctica 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 dada 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:

```
void CopiaLimpia (Ejemplo *original, Ejemplo *&copia) {
      Vacio (copia);
      if not (original == NULL) {
            while (origianl != NULL) {
                  Agregar (original->valor, copia);
                  origianl = original->sig;
            }
      }
}
```

En este ejemplo es claro que el **if** es redundante.

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

```
void CopiaLimpia(Ejemplo *original, Ejemplo *&copia) {
      Ejemplo *aux;

      if (original != NULL) {
            aux = new Ejemplo;
            aux->resto = NULL;
            copia = aux;

            while (original != NULL) {
                  CopiaLimpiaRac(original->valor, aux->valor);
                  original = original->sig;

                  if (original != NULL) {
                        aux->sig = new Ejemplo;
                        aux = aux->sig;
                        aux->sig = NULL;
                  }
            }
      } else {
            copia = NULL;
      }
}
```

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

```
void CopiaLimpia(Ejemplo *original, Ejemplo *&copia) {
      Ejemplo *aux;

      copia = NULL;
      while (original != NULL) {
            aux = new Ejemplo;
            CopiaLimpiaRac(original->valor, aux->valor);
            aux->sig = copia;
            copia = aux;
            original = original->sig;
      }
}
```

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:

```
Ejemplo *ej1, *ej2;
...
if (ej1 == ej2){
      ...
}
```

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 un TAD desde un módulo externo. Es un **error común** hacer `ej1 = new Ejemplo` desde el módulo *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.

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".