6.- **Cambiando la forma de operar de new y delete.**

C++ nos permite cambiar la forma de operar de varios aspectos del sistema. La asignación dinámica de memeoria no es la excepción. Tanto **new** como **delete** se consideran como operadores en C++, por lo tanto, pueden ser sobrecargados. Con lo cual se tiene un control completo sobre la asignación dinámica de memoria.

Esto puede ser necesario bajo las siguientes circunstancias.

1.- Suponga que esta desarrollando código en C++ para un sistema que va a ser almacenado en ROM, este tipo de programas tienen restricciones sobre la posición y el uso de la memoria. En este caso **new** y **delete** pueden ser inapropiados, el programador puede definir un nuevo tipo de almacenamiento libre redefiniendo estos operadores.

2.- El programador puede realizar una nueva implementación de los operadores **new** y **delete**, de tal forma que su sistema opere de una manera más eficiente.

3.- Un programa que realiza asignación dinámica de memoria puede ser difícil de hacerle un debug. Es posible tener acceso a un paquete de C, él cual nos brinde facilidades de hacer el debug, re-escribiendo versiones de **malloc()** y **free()**, éstas pueden ser utilizadas en C++ redefiniendo de manera global a **new** y **delete**.

A continuación presentamos un ejemplo de redefinición global de **new** y **delete**, se deberá incluir **NEW.H** antes de hacer lo siguiente.

```
void *operator new(size_t az) {
     return malloc(az);
}

y

void operator delete(void *dp) {
     free(dp);
}
```

Un requisito que se debe cumplir cuando se usen **new** y **delete** es no cambiar el tipo de argumeto de **new** (él cual tiene un argumento del tipo size_t), ni el valor regresado, **new**, regresa un (void *). En el caso de **delete**, recibe como argumento un puntero a (void *), es la dirección del inicio del bloque a liberar.

6.a) Agotamiento del área de almacenamiento libre.

En algunas aplicaciones, el agotamiento del área de almacenamiento libre, significa que ha ocurrido un error ya que no se ha realizado una adecuada asignación de memoria, sin embargo, hay situaciones donde la falta de área en el heap es normal, y puede ser recuperada, como sigue:

1.- El sistema puede ser diseñado para compactar la información no válida y generar más espacio en el heap.

2.- El sistema puede haber fragmentado en exceso el heap, se puede diseñar un compactador, para que genere espacio libre suficiente.

3.- El sistema puede estar usando redefiniciones de **new** y **delete**, entonces, se pueden tener metodos alternativos para conseguir más memoria, cuando la memoria local del heap se ha agotado.

4.- Se puede querer usar la mayor cantidad de memoria disponible, por ejemplo, como buffer de E/S para una unidad de disco. Una forma portable de ver ésto, es pedir memoria con **new** y ver si falla la llamada.

Cuando se agota el área de almacenamiento libre, el sistema llama a la función **_new_handler()**, la cual para sorpresa, no hace nada. Sin embargo se puede cambiar el trabajo de esta función creando nuestra propia función e instalandola usando la función predefinida **set_new_handler()**. Ahora, cuando se agota el área de almacenamiento libre, se va a llamar nuestra definición de la función new_handler(), luego, cuando se sospeche que se esta corriendo fuera del área de alamcenamiento libre, se puede hacer que el programa imprima un mensaje y aborte.

Declaración de **set_new_handler()**

Descripción: Define la función a ser llamada cuando el operador **new()** ya no puede responder a una solicitud de memoria.
Por default el operador **new()** va a regresar 0 si no puede asignar memoria. También cuando ya no encuentra memoria **new()**, llama a la función new_handler(), redefinida anteriormente por set_new_handler().
La redefinición de la función new_handler() deberá especificar la acción a seguir si **new** no encuentra memoria para asignar.
La redefinición de la función new_handler() no deberá regresar valor.
El manejador de default es reseteado usando la función: set_new_handler(0).

Portabilidad Esta función no se encuentra disponible en ANSI C.

Sintaxis

```
void( *set_new_handler(void( *new_handler())) ();
```

Ejemplo:

```
// Programa PLCP70.CPP
// Usando un manejador propio
#include <iostream.h>
#include <new.h>
#include <stdlib.h>

void mem_error()
{
  cerr << "\nNo se puede asignar memoria!" << endl;
  exit(1);
}

void main(void)
{
    set_new_handler(mem_error);
    char *ptr;

    for(int i=1;i;i++) {
        ptr = new char[6400];
        cout << "Asignación[" << i << "]: ptr = "
             << hex << long(ptr) << "H" << endl;
    }
    set_new_handler(0);  // Regresa al default
}
```

OBSERVACIONES.

En el programa se esta usando un for infinito, sin embargo, si así se definiera, el compilador marca una advertencia en la instrucción set_new_handler(0); ya que se da cuenta que nunca llegaría a ejecutarse, por ello en la condición del for usamos la variable **i**.

La salida del programa nos resulta:

```
Asignación[1]: ptr = ce10f20H
Asignación[2]: ptr = ce12824H
Asignación[3]: ptr = ce14128H
Asignación[4]: ptr = ce15a2cH
Asignación[5]: ptr = ce17330H
Asignación[6]: ptr = ce18c34H
Asignación[7]: ptr = ce1a538H
Asignación[8]: ptr = ce1be3cH
```

```
Asignación[9]: ptr = ce1d740H
```

No se puede asignar memoria!

Las direcciones que fué regresando **new** a ptr, son en hexadecimal, en donde las 4 cifras menos significativas, representan el offset de la dirección de memeoria, y las 4 más significativas representan la dirección de segmento. (En la salida del programa solo aparecen 3, la otra sería 0).

En este caso, recibe como argumento la dirección de la función mem_error(), siendo instalada como la función a ser llamada cuando **new** ya no encuentra memoria.

6.b) Sobrecarga de **new** y **delete** clase por clase.

Cuando se define a **new** y **delete** como miembros de una clase, el compilador va a llamar a estas funciones solamente cuando se crea un objeto de la clase sobre el área de almacenamiento libre. Mediante esta técnica se puede cambiar de manera selectiva el significado de **new** y **delete** y trabajar de manera diferente que su definición global, la cual también pudo haber sido sobrecargada.

Recuerde que cuando se llama a **new** se pide memoria y a continuación se llama al constructor de la clase. Al llamar a **delete**, el compilador primero llama al destructor y luego ejecuta **delete** para liberar memoria.

La sintaxis para clase por clase de **new** y **delete** es la misma que la de la versión global.

En el siguiente ejemplo, **new** y **delete** son sobrecargados para indicarnos cuando son llamados y cuando se llama a su versión global.

```
// Programa PLCP71.CPP
// Sobrecarga de los operadores new y delete.
#include <iostream.h>
#include <stdlib.h>
#include <new.h>    // Declaración del nuevo manejador



class sola {
    static int v;
    int i [2000];
public:
    void *operator new(size_t az) {
        if(v)
            cout << "sola::operator new" << endl;
        return ::new unsigned char[az];
```

```
    }
    void operator delete(void *dp) {
        cout << "sola::operator delete" << endl;
    ::delete(dp);
    }
    static void define() { v = 0; }
};


int sola::v = 1;

unsigned long cuenta = 0;

void mem_fuera()
{
    cerr << "Fuera del área de memeoria después de "
         << cuenta << " objetos" << endl;
    exit(1);
}

void main()
{
    set_new_handler(mem_fuera);
    sola *ptr = new sola;
    delete ptr;
    sola::define();
    for(int k=1;k;k++) {
        ptr = new sola;
        cout << "Asignación[" << k << "]: ptr = "
             << hex << long(ptr) << "H" << endl;
        cuenta++;
    }
    set_new_handler(0);  // Regresa al default
}
```

OBSERVACIONES.

La clase **sola** se hizo de manera intencional grande, de tal forma que pronto llenara el espacio del heap.

La variable **v** se usa para que cuando ésta sea v = 1, mande imprimir el mensaje dentro de **new**, y cuando sea v = 0, no lo imprima, esto se hace con el fin de no llenar la pantalla de información y que ésta solo aparezca cuando se declara el puntero a la clase y cuando es mandada liberar la memoria, en los casos siguientes también son llamados estos operadores sobrecargados, pero ya no se imprime el mensaje.

En la función main(), después que es definido el nuevo manejador se crea un objeto del tipo **sola** y luego se destruye, aquí es donde aparecen los mensajes, ya que se estan llamando las nuevas versiones de **new** y **delete**.

Después dentro del ciclo for, se llama de manera infinita a la nueva versión de **new**, (ahora ya no aparece el letrero, ya que v = 0). La solicitud de memeoria terminará cuando ya no haya espacio disponible en el heap. Puede ver la cantidad de espacio aproximado de que dispone en su programa como área de almacenamiento libre.

Si cada vez que se llama al heap, se solicitan de manera aproximada 4000 bytes, y se realizan 14 llamadas, entonces, el área disponible para el heap será aproximadamente = 56 KB.

Veamos la corrida del programa:

```
sala::operator new
sola::operator delete
Asignación[1]: ptr = 2e770f74H
Asignación[2]: ptr = 2e771f18H
Asignación[3]: ptr = 2e772ebcH
Asignación[4]: ptr = 2e773e60H
Asignación[5]: ptr = 2e774e04H
Asignación[6]: ptr = 2e775da8H
Asignación[7]: ptr = 2e776d4cH
Asignación[8]: ptr = 2e777cf0H
Asignación[9]: ptr = 2e778c94H
Asignación[a]: ptr = 2e779c38H
Asignación[b]: ptr = 2e77abdcH
Asignación[c]: ptr = 2e77bb80H
Asignación[d]: ptr = 2e77cb24H
Asignación[e]: ptr = 2e77dac8H
Asignación[f]: ptr = 2e77ea6cH
```

"Fuera del área de memeoria después de 15 objetos"

Si se deseara llamar dentro de main(), las versiones globales de **new** y **delete**, estas deberían ir precedidas por el operador de ámbito:

```
char *ptr ::new char[100];
          ::delete(ptr);
```

6.c) Recolección de información no válida.

Como ejemplo de un programa con información no válida, el siguiente programa define una clase llamada **basura**, los objetos **basura** se utilizan para algunos propósitos "misteriosos" y a continuación se eliminan. Cuando el sistema corre fuera del área de

almacenamiento libre los objetos **basura** que no estan en uso, son liberados por el recolector de basura. Para saber si un objeto todavía esta en uso, el objeto es autocuestionado. La variable **en_uso** es actualizada de acuerdo al criterio específico de la aplicación.

Para almacenar los punteros que contienen las direcciones regresadas por **new**, usamos la clase **Arredin** definida anteriormente. Note la conveniencia de usar la clase **Arredin** una vez definida. Primero almacena los punteros como los va regresando **new**, posteriormente colocamos nuestro indice al inicio del **arreglo**, lo vamos recorriendo y vamos liberando memoria.

```
// Programa PLCP72.CPP
// Un ejemplo simple de la manera que puede
// trabajar un colector de basura

#include <iostream.h>
#include <stdlib.h>
#include <new.h>
#include "plcp63.h"

Arredin memtabla;   // Coloca los punteros para luego
                    // libre almacenamiento.


class basura {
    enum { tam = 2000 };
    int memoria[tam];
    int en_uso;
public:
    basura() { en_uso = 0; }
    void *operator new(size_t az) {
        void *g = ::new unsigned char[az];
        memtabla.suma(g);
        return g;
    }
    int vive() { return en_uso; }
    int muere() { return !en_uso; }
};


void colector_basura() {
    cout << "\nRecolectando basura... ";
    cout << dec << memtabla.cuenta();
    memtabla.reinicia();
    do
        if(((basura *)memtabla.actual()) -> muere()) {
            cout << "\nRemoviendo " << hex
```

```
                    << (long)memtabla.actual();
                delete (basura *)memtabla.actual();
                memtabla.remueve(memtabla.indice());
            }
        while( memtabla.sig() );
        cout << "\nHecho!." << endl;
}

void main()
{
    set_new_handler(colector_basura);
    basura *ptr;

    for(int k=1;k < 31;k++) {
        ptr = new basura;
        cout << "\nAsignación[" << dec << k << "]: ptr = "
             << hex << long(ptr) << "H";
    }
    set_new_handler(0);  // Regresa al default
}
```

OBSERVACIONES.

Este programa para su compilación se esta usando la interface y la implementación de la clase **Arredin**. Se define el objeto **memtabla** como global para poder ser utilizado, tanto por el constructor de la clase **basura** como por la función global, colector basura.

El programa muestra la manera en que se va solicitando memoria, cuando el heap, ya se ha llenado, el programa también muestra la actividad de la función colector_basura(), la cual limpia todo el heap, dejandolo listo para una nueva asignación de memoria.

Este programa, igual que los anteriores puede tener un for infinito, en este caso el programa nunca terminará su ejecución, para detenerlo, deberiamos usar [^C]. En el ejemplo solo mostramos ciclo y medio.

La coorida en este ciclo y medio nos resulta:

```
Asignación[1]: ptr = 2e300facH
Asignación[2]: ptr = 2e301f52H
Asignación[3]: ptr = 2e302ef8H
Asignación[4]: ptr = 2e303e9eH
Asignación[5]: ptr = 2e304e44H
Asignación[6]: ptr = 2e305deaH
```

```
Asignación[7]: ptr = 2e306d90H
Asignación[8]: ptr = 2e307d36H
Asignación[9]: ptr = 2e308cdcH
Asignación[10]: ptr = 2e309c82H
Asignación[11]: ptr = 2e30ac28H
Asignación[12]: ptr = 2e30bbfaH
Asignación[13]: ptr = 2e30cba0H
Asignación[14]: ptr = 2e30db46H
Asignación[15]: ptr = 2e30eaecH
Recolectando basura... 15
Removiendo 2e300fac
Removiendo 2e301f52
Removiendo 2e302ef8
Removiendo 2e303e9e
Removiendo 2e304e44
Removiendo 2e305dea
Removiendo 2e306d90
Removiendo 2e307d36
Removiendo 2e308cdc
Removiendo 2e309c82
Removiendo 2e30ac28
Removiendo 2e30bbfa
Removiendo 2e30cba0
Removiendo 2e30db46
Removiendo 2e30eaec
Hecho!.

Asignación[16]: ptr = 2e30ac28H
Asignación[17]: ptr = 2e309c82H
Asignación[18]: ptr = 2e308cdcH
Asignación[19]: ptr = 2e307d36H
Asignación[20]: ptr = 2e306d90H
Asignación[21]: ptr = 2e305deaH
Asignación[22]: ptr = 2e304e44H
Asignación[23]: ptr = 2e303e9eH
Asignación[24]: ptr = 2e302ef8H
Asignación[25]: ptr = 2e301f52H
Asignación[26]: ptr = 2e300facH
Asignación[27]: ptr = 2e30bbfaH
Asignación[28]: ptr = 2e30cba0H
Asignación[29]: ptr = 2e30db46H
Asignación[30]: ptr = 2e30eaecH
```

7.- **Riesgos en la asignación dinámica de memoria.**

Cuando se decide utilizar asignación dinámica de memoria, tanto para tipos predefinidos, como para tipos definidos por el usuario, se puede uno topar con una serie de trampas.

Cuando se crea una variable en el heap su tiempo de vida es arbitrario, es responsabilidad del programador borrar la variable y liberar la memoria. Si no se cuida la limpieza del heap, se correrá el riesgo de salirse de memoria.

Deberá tenerse cuidado, cuando se define una clase que utiliza un constructor que solicita memoria al heap, se deberá implementar su destructor que libere la memoria asignada por el constructor.

Encontraremos más problemas cuando pasemos como argumento de una función un objeto, y se devuelve un objeto desde ella. Si el objeto contiene punteros al heap, se copiará la estructura que esta en la pila, más no la memoria en el heap, (ésta no se copia). Cuando usemos un destructor, y éste libere adecuadamente la memoria, al devolver el objeto, contendra punteros al heap, cuya memoria ya fué liberada.

7.a) Calculando el tamaño de un objeto.

Deberemos saber como clcular el tamaño de una clase cuando se llama a **new** o a otra función que asigne memoria dinámica. De la misma forma que con los tipos predefinidos, se dá a **new** el nombre de la clase, y se creará un objeto del tamaño adecuado.

En el siguiente ejemplo se muestra una clase la cual tiene un puntero a otra clase, y la memoria es solicitada al heap.

```
// El tamaño de los objetos se calculan de manera automática
// cuando se usa new.

class X {
     int i, j, k;
     ...
     ...
};

class Y {
     long a, b, c;
     ...
     ...
};

class Z {
     double d, e, f;
```

```
     ...
     ...
};

class con_class {
     X *xp;
     Y *yp;
     Z *zp;
public:
     con_class() {
          xp = new X;
          yp = new Y;
          zp = new Z;
     }
};
```

En ocasiones es necesario calcular el tamaño de un objeto, mientras se realiza asignación dimámica. Esto se puede hacer usando el operador **sizeof**. En el siguiente ejemplo, una clase contiene un puntero a una **struct** la cual contiene el tamaño de un vector y la dirección de inicio del vector.

```
// Programa PLCP73.CPP
// Calculando el tamaño para asignación
// dinámica a memoria, usando sizeof
#include <iostream.h>
#include <assert.h>
#include <stdlib.h>
class vvec {
    struct vec {
        int tam;
        float v[1];
        } *vv;
public:
    vvec(int az) {
       vv = (vec *)new char[sizeof(vec) + (az -1)*sizeof(float)];
       vv ->tam = az;
    }
    ~vvec() { delete vv; }
    float &operator[](int i);
};

float &vvec::operator[](int i)
{
    if(i > 0 && i < vv -> tam)
        return vv -> v[i];
    static float d = 0;
    return d;
}
```

```
typedef long tipo;
const bt = sizeof(tipo) + 8;

void main(int argc, char *argv[])
{
    assert(argc > 1);
    const sz = atoi(argv[1]);
    vvec A(sz);
    for(int i = 0; i < sz; i++)
        A[i] = (tipo)1 << (i%bt);
    for(i = 0; i < sz; i++)
        cout << "A[" << i << "] = " << A[i] << endl;
}
```

OBSERVACIONES.

Esta forma de trabajar es aceptable, más no recomendable. Note que la **struct** anidada **vec**, la cual es visible solo desde dentro de la clase **vvec**, contiene un indicador de tamaño y un arreglo de tamaño [1] (número de bytes = 6). Sin embargo, solo hay un apuntador, y no un objeto de **vec** dentro de **vvec**. En el constructor se asigna memoria para contener un vector de la longitud deseada. El cálculo se realiza mediante la siguiente expresión:

```
new char[sizeof(vec) + (az -1)*sizeof(float)];
```

La primera parte **sizeof(vec)** es simplemente el tamaño de la estructura, como se debería esperar, sin embargo, la segunda parte **(az -1)*sizeof(float)**, contiene el argumento **az** del constructor, el cual lo usa para calcular el tamaño de un arreglo de numeros de tipo **float**. Pero por qué, menos 1?. La razón es que **vec** ya contiene el primer elemento del arreglo. Ahora la memoria asignada a **vec**, no es un arreglo de 1, sino de cualquier tamaño asignado. Para el caso del ejemplo que sigue, en donde se da en main() un argumento de 10, el número de bytes solicitados al heap es de 42. Este truco es usado en lugares donde el código generado por el compilador puede ser más eficiente.

Note que el miembro **tam** deberá estar definido dentro de la clase. Sin embargo, qué sucede si se desea apuntar **vv** a otro objeto **vec**, él cual puede tener diferente tamaño. Por ello es mejor tener siempre toda la información referente a un objeto junta, dentro de él mismo, haciendo ésto, es imposible que el objeto olvide sus características propias, entre las que se incluye el tamaño del arreglo.

El operador sobrecargado **operator[]()**, simplemente regresa una referencia al elemento seleccionado, si esta dentro del rango y a un **static float** si se encuentra fuera del rango, lo que nos permite detectar un error.

En main(), se recibe un argumento en la línea de comandos, el cual es una cadena, se convierte a número usando la función de la librería de ANSI C, atoi(). Este número se usará para determinar el tamaño de **vvec A**. Entonces **A** es llenado usando la expresión:

```
A[i] = (tipo)1 << (i%bt);,
```

Note en el programa que **tipo** se determina mediante un typedef, por lo que podemos probar el programa para tipos como char, short int, signed y unsigned. Lo que hace la expresión anterior es realizar un corrimiento del 1 << n_bits, donde n_bits es el resultado de la expresión i%bt, esto es, el módulo de i entre bt, para nuestro caso bt = 4 + 8;

A continuación presentamos la corrida del programa, al teclar en la línea de comandos A:\USER>**PLCP71 10** [Enter]

```
A[0] = 1
A[1] = 2
A[2] = 4
A[3] = 8
A[4] = 16
A[5] = 32
A[6] = 64
A[7] = 128
A[8] = 256
A[9] = 512
```

La macro **assert** prueba una condición y si esta no se cumple aborta después de mandar un mensaje a la pantalla usando el archivo **stderr**.

Sintaxis

```
#include <assert.h>
void assert(int prueba);
```

7.b) Usando referencias con asignación dinámica de memoria.

Es posible asignar el valor regresado por **new** a una referencia en vez de a un puntero. Veamos un ejemplo:

```
// Programa PLCP74.CPP
```

```
// Usando referencias en asignación dinámica.

#include <iostream.h>

class magica {
    enum {tam = 80 };
    char cad[tam];
public:
    void lee() {
        cout << "Dame cadena: ";
        cin.get(cad, tam);
    }
    void escribe() {
        cout << "Mensaje: " << cad << endl;
    }
};

void main()
{
    magica &mc = * new magica;
    mc.lee();
    mc.escribe();
    delete &mc;
}
```

OBSERVACIONES.

Las referencias pueden ser atractivas ya que realizan la asignación dinámica de un objeto, como si se tratara de un objeto local, usando la misma sintaxis en la llamada a funciones miembro. Aunque ésto también puede ser engañoso, ya que un destructor es llamado por un objeto local. (una referencia no puede llamar a un destructor) Puesto que una referencia es menos flexible que un puntero (no es posible reasignar una referencia), no se acostumbra utilizarlas en asignación dinámica de memeoria.