Asignación dinámica de memoria en C++

Operadores new y delete

En el capítulo de asignación dinámica de la primera parte de la materia vimos las razones por las que nos resultaba de utilidad crear variables con esta técnica, y que las funciones de C que teníamos que usar eran malloc() y free().

En C++ en cambio se utilizan los operadores **new** (para pedir memoria) y **delete** (para liberar la memoria).

La sintaxis para estos operadores es la siguiente:

```
int main(){
      int *v, tam;
      cout<<"INGRESE LA CANTIDAD DE ELEMENTOS PARA EL VECTOR: ";
      cin>>tam;
      v=new int[tam];
      if(v==NULL) return -1;
     ///si el programa llega a esta línea se puede usar v de manera
     /// idéntica a que si se hubiera declarado de manera estática
     …….
      ….……
      ///luego de usar el vector, se libera la   memoria
      delete v;
      return 0;
      }
```

Donde:
v=puntero del mismo tipo de datos que el vector que se quiere construir.
new operador para pedir memoria.
int tipo de datos del vector que se quiere construir. Tiene que coincidir con el tipo del puntero
[tam]: cantidad de componentes del vector. No confundir con cantidad de bytes que necesita malloc() como parámetro

Como puede observarse, el uso de estos operadores simplifica la sintaxis de la asignación dinámica. No es necesario hacer un "casteo" ni calcular la cantidad de bytes.

Veamos la aplicación de estos operadores en los ejemplos vistos en el apunte anterior:

1) Un comercio tiene un archivo (ventas.dat) con la información de las ventas con el siguiente formato:
- Nº de venta
- Fecha (dia, mes, año)
- Código de producto (1 a 120)

- Importe
- Cantidad vendida

La gerencia necesita hacer un análisis de las ventas del año 2010, por lo solicita se desarrolle un programa para calcular e informar:

a) El importe recaudado por cada producto y mes del año
b) El producto por el que se recaudó más

La resolución del punto b) sería, suponiendo que son 120 productos, que sus códigos son cadenas, y que contamos con el archivo de productos, la siguiente:

```
void puntoB(){
 FILE *pv;
  struct venta reg;
  float impArt[120];
  int fila;
  ponerCeroVectorF(impArt, 12);
  pv=fopen("ventas.dat", "rb");
  if(pv==NULL){
    cout<<"ERROR DE ARCHIVO"<<endl;
    system("pause");
    exit(1);
    }
  while(fread(&reg, sizeof reg, 1, pv)==1){
    if(reg.anio==2010){
      fila=buscarPosicionRegistro(reg.codigoProducto);
      if(fila!=-1)
        impArt[fila]+=reg.importe;
      }
    }
  fclose(pv);
  mostrarPuntoB(impArt);
  }
}
```

Como sabemos que son 120 productos, acumulamos en un vector de float de esa dimensión los importes parciales. Para saber en que posición del vector acumular los importes parciales, utilizamos la posición que el producto ocupa en el archivo. Luego la función mostrarPuntoB(), buscaría cuál es el mayor importe, y a partir de la posición que corresponde al mayor importe dentro del vector, ubicamos y mostramos el registro coincidente del archivo.

Ahora ¿cómo resolver el problema si no sabemos la cantidad de productos?: no podemos declarar el vector ya que nos falta el tamaño, por lo cual el algoritmo sería mucho más complejo que el anteriormente visto.

Podríamos hacer una función que nos devuelva la cantidad de registros del archivo –lo cual nos resolvería el problema de no conocer la cantidad de productos- pero igualmente seguiríamos

sin la posibilidad de declarar el vector, ya que sólo podemos conocer la cantidad de registros en el momento que se ejecute el programa (se dice en tiempo de ejecución), y no al momento de escribir el programa (se dice en tiempo de diseño). Para resolver esto existe un mecanismo que nos permite declarar dinámicamente las variables (recordemos que hasta ahora todas las variables son estáticas), que se denomina **asignación dinámica de memoria**.

Veamos como resolver el ejercicio mediante la asignación dinámica:

```
void puntoB(){
  FILE *pv;
  struct venta reg;
  int fila;
  float *impArt;
  int cantReg;
  cantReg=contarRegistros();
  impArt=new float[cantReg];
  if(impArt==NULL){
            cout<<”Error de asignación de memoria”<<endl;
            system(“pause”);
            return;
  }
  ponerCeroVectorF(impArt, cantReg);
  pv=fopen("ventas.dat", "rb");
  if(pv==NULL){
    cout<<"ERROR DE ARCHIVO"<<endl;
    system("pause");
    exit(1);
    }
  while(fread(&reg, sizeof reg, 1, pv)==1){
    if(reg.anio==2010){
      fila=buscarPosicionRegistro(reg.codigoProducto);
      if(fila!=-1)
        impArt[fila]+=reg.importe;
      }
    }
  fclose(pv);
  mostrarPuntoB(impArt);
  delete impArt;
  }
}
```

Las líneas de código agregadas o diferentes al programa anterior son:

```
float *impArt;// se declara un puntero a float
int cantreg;     // se declara una variable entera
cantreg=contarRegistros(); // se llama a una función que devuelve la cantidad de
registros //del archivo de productos
```

```
impArt=new float[cantreg];// se pide memoria para un vector de cantreg elementos de
tipo //float
if(impArt==NULL){// se chequea que se halla podido asignar la memoria pedida
        cout<<"Error de asignación de memoria"<<endl;
        system("pause");
        return;
        }
/////////
delete impArt; // se libera la memoria pedida
```

El resto de las líneas permanece igual.

Veamos en detalle cada línea:

float *impArt;
Como no se sabe la cantidad de elementos del vector no se lo puede declarar de manera estática. Se declara un puntero a float, sobre el cual se pedirá luego memoria.

cantreg=contarRegistros();
Se llama a una función que devuelve la cantidad de registros del archivo de productos; se asigna el valor devuelto a la variable cantreg

impArt=new float[cantreg];

El operador new permite pedir memoria. Se debe indicar el tipo y entre corchete la cantidad de componentes que se solicita.

Se pide memoria para un vector de cantreg elementos de tipo float.

Si la asignación pudo hacerse, new devuelve la dirección donde empieza en la memoria el espacio solicitado; si no pudo hacerse la asignación, new devuelve NULL. Por esa razón es que antes de continuar con el programa se analiza el valor del puntero sobre el que se solicitó memoria

La última línea agregada fue

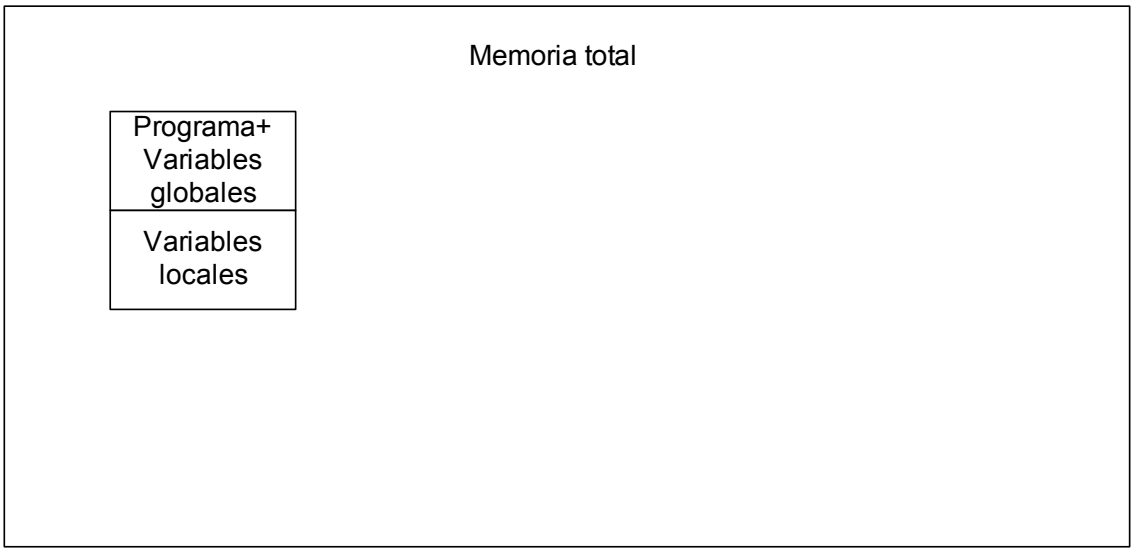
delete impArt;

El operador delete permite liberar la memoria pedida. Se debe indicar el puntero en el cual se hizo la asignación de memoria.

Funcionamiento del mecanismo de asignación dinámica de memoria

Hemos visto como utilizar el operador **new** para asignar dinámicamente memoria, y **delete** para liberar la memoria pedida. Veamos ahora como es su funcionamiento.

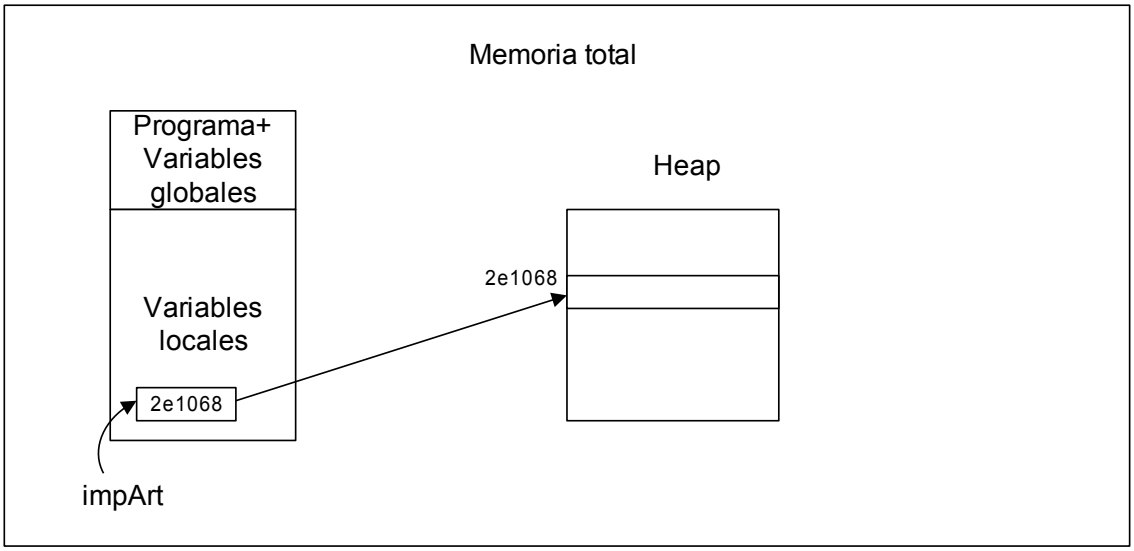
Al compilarse y enlazarse un programa, se reserva un espacio de memoria para el programa, las variables globales, y las variables locales a las funciones. El programa y las variables globales ocuparán un espacio fijo, equivalente a la suma de las necesidades del programa y la suma del tamaño de las variables globales declaradas; en otro espacio contiguo se reserva una pila para las variables locales, ya que no se utilizan todas al mismo tiempo. Podríamos representar lo anterior de la siguiente manera:



El programa sólo puede ejecutarse dentro del espacio de memoria que le ha sido asignado al cargarse por el sistema operativo. No se le permitirá acceder a direcciones de memoria fuera de esos límites.

Dentro del sistema de memoria, existe una zona denominanada *heap* que puede ser utilizada de manera compartida por los programas en ejecución, usando asignación dinámica. Cuando los programas necesitan acceder a esas posiciones deben solicitarle al administrador de memoria del sistema operativo la cantidad de bytes requeridos, y en caso de disponer un bloque disponible de ese tamaño, el sistema operativo devolverá la dirección de memoria donde comienza ese bloque; caso contrario la respuesta será NULL. En C++ hacemos esta operación por medio del operador new; al terminar de usar el bloque de memoria, se la debe dejar libre para el caso que sea necesaria por otro programa. En C++ la liberación de memoria se hace por medio de la función delete.

Un esquema simplificado para el programa visto sería:



Dentro del área de memoria asignado al programa, se declara el puntero impArt. Cuando se ejecuta la línea:

impArt**=new** float [cantreg];

se le pide al sistema operativo un bloque de cantreg*sizeof (float) bytes contiguos para construir el vector; como hay disponibilidad en el *heap*, devuelve la dirección donde comienza el bloque (por ejemplo 2e1068), y esta dirección se asigna al puntero impArt.

Luego de la asignación, podemos utilizar impArt del mismo modo que utilizamos cualquier vector declarado de manera estática. Cuando no necesitamos más el vector, se libera la memoria solicitada con delete.

Veamos otro ejemplo:

La resolución del punto a) del ejercicio 1) fue la siguiente:

```
void puntoA(){
  FILE *pv;
  struct venta reg;
  float mVentas[120][12];
  ponerCeroMatrizF(mVentas, 120, 12);
  pv=fopen("ventas.dat", "rb");
  if(pv==NULL){
    cout<<"ERROR DE ARCHIVO"<<endl;
    system("pause");
    exit(1);
    }
  while(fread(&reg, sizeof reg, 1, pv)==1){
    if(reg.anio==2010)
         mVentas[reg.codigoProducto-1][reg.mes-1]+=reg.importe;
```

```
    }
  fclose(pv);
  mostrarPuntoA(mVentas);
  }
```

Como sabíamos que la cantidad de productos era de 120, y la cantidad de meses 12 pudimos declarar una matriz donde almacenar la información requerida (float mVentas[120][12];), y resover el problema con un algoritmo sencillo.

Supongamos que, como en el ejemplo 2 de los ejercicios de la clase anterior, contamos con el archivo de productos, y que éste contiene un registro por cada producto, y que cada producto está identificado por un número (el primer producto está identificado por el número 1, y el último registro por el número N, siendo N la cantidad de registros), pero desconocemos cuántos son los productos, lo cual nos impide declarar estáticamente la matriz. Podríamos entonces usar la función que nos devuelva la cantidad de registros del archivo, y construir dinámicamente la matriz.

La resolución sería:

```
void puntoA(){
  FILE *pv;
  struct venta reg;
  float (*mVentas)[12];
  int cantreg;
  cantreg=contarRegistros();
  mVentas=new float[cantreg] [12];
  if(mVentas==NULL){
            cout<<”Error de asignación de memoria”<<endl;
            system(“pause”);
            return;
            }
  ponerCeroMatrizF(mVentas, cantreg, 12);
  pv=fopen("ventas.dat", "rb");
  if(pv==NULL){
    cout<<"ERROR DE ARCHIVO"<<endl;
    system("pause");
    exit(1);
    }
  while(fread(&reg, sizeof reg, 1, pv)==1){
    if(reg.anio==2010)
         mVentas[reg.codigoProducto-1][reg.mes-1]+=reg.importe;
    }
  fclose(pv);
  mostrarPuntoA(mVentas, cantreg);
  delete mventas);
  }
```

Las líneas de código agregadas o diferentes al programa anterior son:

```
float (*mVentas)[12];// se declara un puntero a una fila de 12 columnas
int cantreg;     // se declara una variable entera
cantreg=contarRegistros();   // se llama a una función que devuelve la cantidad de registros
                             //del archivo de productos
mVentas=new float[cantreg] [12]; // se pide memoria para una matriz de cantreg filas y 12
                                 //columnas  de tipo float
if(mVentas==NULL){// se chequea que se halla podido asignar la memoria pedida
        cout<<"Error de asignación de memoria"<<endl;
        system("pause");
        return;
        }
/////////
delete mventas; // se libera la memoria pedida
```

El resto de las líneas permanece igual.

Veamos en detalle cada línea:

float (*mVentas)[12];
Como no se sabe una de las dimensiones de la matriz (la cantidad de filas), no se la puede declarar. Se declara un puntero a una fila de 12 columnas, sobre el cual se pedirá luego memoria.

cantreg=contarRegistros();
Se llama a una función que devuelve la cantidad de registros del archivo de productos; se asigna el valor devuelto a la variable cantreg

mVentas= new float[cantreg] [12];

Se pide memoria para una matriz de cantreg filas y 12 columnas de tipo float.

Si la asignación pudo hacerse, la dirección que devuelve es donde empieza en la memoria el espacio solicitado; si no pudo hacerse la asignación, malloc() devuelve NULL. Por esa razón es que antes de continuar con el programa se analiza el valor del puntero sobre el que se solicitó memoria.

La última línea agregada fue

delete mVentas;
para liberar la memoria pedida.

Ejemplo de asignación dinámica en una matriz de enteros cuando no se conoce ninguna de las dimensiones de la matriz

```
int main() {
     int filas, columnas,i ;
     int **vp;
     cout<<"INGRESE LA CANTIDAD DE FILAS DE LA MATRIZ: ";
     cin>>filas;
     cout<<"INGRESE LA CANTIDAD DE COLUMNAS DE LA MATRIZ: ";
     cin>>columnas;
     vp=new int*[filas];///se crea un vector de punteros
     if(vp==NULL)return -1;
     for(i=0;i<filas;i++){
          vp[i]=new int[columnas];///en cada posición del vector de
                    //punteros se pide memoria para crear la fila
          if(vp[i]==NULL)return -1;
     }
     cargarMatriz(vp,filas,columnas);///se utiliza de la misma
                                   //manera que una matriz estática
     mostrarMatriz(vp, filas, columnas);
     for(i=0;i<filas;i++){///para liberar la memoria, se procede de
                           ///manera inversa. Primero las filas
          delete vp[i];
     }
     delete vp; ///por último se libera el vector de punteros
     return 0;
}

///Funciones. La única diferencia con la matriz estática es que debe
///recibirse con un puntero a puntero (int **vp)

void cargarMatriz(int **vp,int tam1,int tam2){
     int i, j;
     for(i=0;i<tam1;i++)
          for(j=0;j<tam2;j++)
               vp[i][j]=i+j;

}
void mostrarMatriz(int **vp,int tam1,int tam2){
     int i, j;
     for(i=0;i<tam1;i++){
          for(j=0;j<tam2;j++){
               cout<<vp[i][j]<<"\t";
          }
          cout<<endl;
     }
}
```