Autor: Lic. Kloster Daniel
Carrera: Técnico Universitario en Programación
Materia: Programación II
Tema: Asignación dinámica de memoria

Asignación dinámica de memoria

En el transcurso de las clases de las materias relacionadas con programación, hemos avanzado en el conocimiento y el uso de las estructuras de datos. Empezamos manejando variables simples, luego arrays (vectores y matrices), y punteros, y llegamos a crear, mediante el desarrollo de clases, variables registro que se ajustan a nuestras necesidades de procesamiento de información. De este tipo de variables también podemos declarar y utilizar arrays y punteros. La posibilidad de conocer variables más complejas nos permite abordar problemas más complejos también.

En esta dirección seguimos, y aprendimos a trabajar con archivos. Sabemos que un archivo puede definirse como un conjunto de información relacionada, almacenada en un dispositivo externo –generalmente un disco u otro dispositivo de almacenamiento masivo-, que tiene un nombre para su identificación, y una ubicación –ruta- dentro del dispositivo. También podríamos definirlo como una **estructura de datos almacenada externamente** compuesta por un conjunto de elementos relacionados, de longitud variable (recuérdese que las estructuras de datos residentes en memoria principal tienen un tamaño fijo establecido en su definición). La longitud de la estructura de datos archivo tiene como límite la capacidad de la unidad donde reside, que en general es mucho más grande que la memoria RAM. La posibilidad de almacenar grandes volúmenes de información en archivos permite independizar los datos de los programas que usan estos datos.

A pesar de todo esto, seguimos teniendo una limitación: todas las variables de memoria que conocemos son estáticas: debemos conocer previamente su tamaño para utilizarlas, ya que C/C++ nos exige declarar las variables antes de su uso. Si bien lo anterior no afecta a las variables simples, si resulta crítico para vectores y matrices.

Veamos el siguiente ejemplo:

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 2020, por lo que 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, sabiendo que son 120 productos identificados con números a partir del 1, la siguiente:

```
void puntoB(){
 FILE *pv;
  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");
    return;
    }
  while(fread(&reg, sizeof reg, 1, pv)==1){
    if(reg.getAnio()==2020){
      impArt[reg.getCodigoProducto()-1]+=reg.getImporte();
      }
    }
  fclose(pv);
  mostrarPuntoB(impArt);
  }
}
```

Suponemos la existencia de la clase Venta para la resolución.

Como sabemos que son 120 productos, acumulamos en un vector de float de esa dimensión los importes parciales. Para saber en qué posición del vector acumular los importes parciales, utilizamos el código de producto -1. Luego la función mostrarPuntoB(), buscaría cuál es el mayor importe, y a partir de la posición que corresponde al mayor importe dentro del vector se le suma un 1 y se obtiene el código de producto. Luego se puede ubicar el registro coincidente con el código de producto dentro del archivo y mostrarlo.

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 de productos–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 cómo resolver el ejercicio mediante la asignación dinámica:

```
void puntoB(){
 FILE *pv;
  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;
      return;
    }
  while(fread(&reg, sizeof reg, 1, pv)==1){
    if(reg.getAnio()==2020){
       impArt[reg.getCodigoProducto()-1]+=reg.getImporte();
      }
    }
  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. 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];
Se pide memoria para un vector de cantreg elementos de tipo float.

El operador de C++ para asignar memoria es new. Su sintaxis es:

puntero_tipoX=new tipoX[cantidad_elementos]

Devuelve una dirección; se le debe indicar el tipo del vector y la cantidad de componentes entre corchetes.

Si la asignación pudo hacerse, la dirección que devuelve es 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.

Volvamos a analizar la línea donde se pide memoria.

impArt=new float[cantreg];
como **new** devuelve una dirección se necesita un puntero para almacenarla

La última línea agregada fue

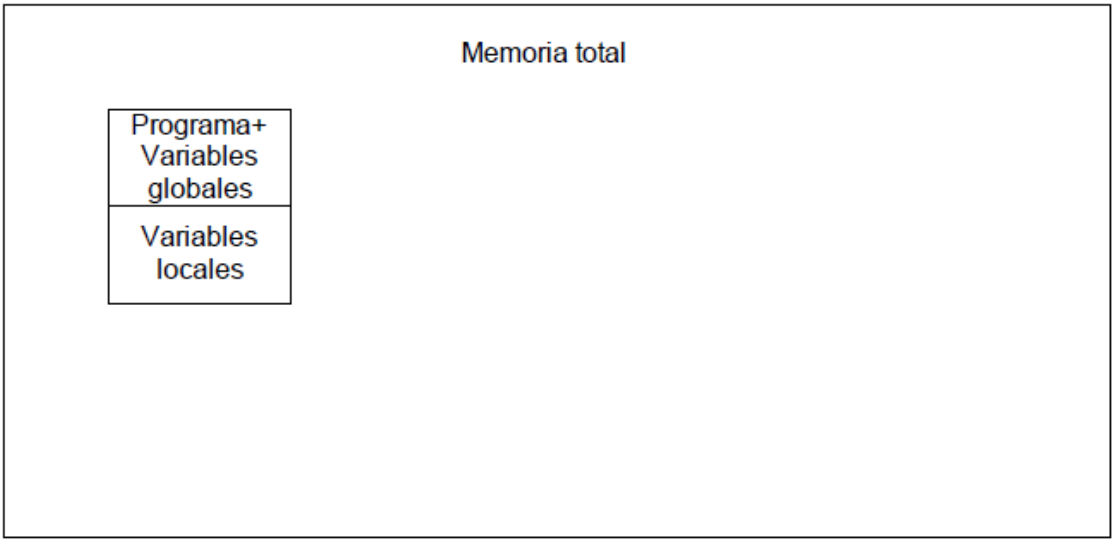
delete impArt;

delete permite liberar la memoria pedida. Debe agregarse el puntero en el cual se hizo la asignación de memoria.

Funcionamiento del mecanismo de asignación dinámica de memoria

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

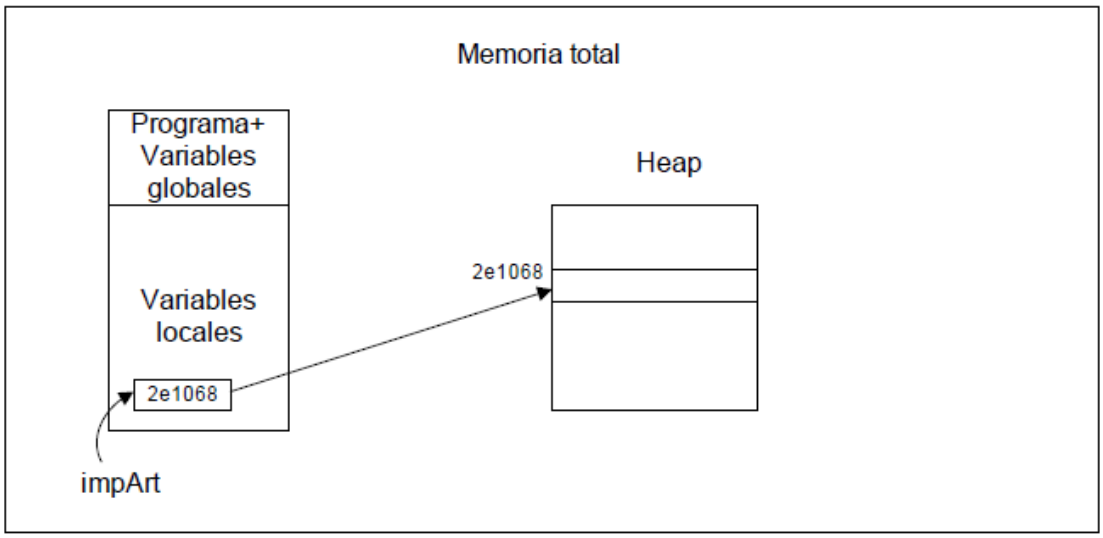
Al compilarse y enlazarse un programa, se reserva un espacio de memoria para el programa, las variables globales, y las variables locales a las funciones. El programa y las variables globales ocuparán un espacio fijo, equivalente a la suma de las necesidades del programa y la suma del tamaño de las variables 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 denominada *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 de **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 **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) puede ser la siguiente:

```
void puntoA(){
  FILE *pv;
  Venta reg;
  float mVentas[120][12];
  ponerCeroMatrizF(mVentas, 120, 12);
  pv=fopen("ventas.dat", "rb");
  if(pv==NULL){
    cout<<"ERROR DE ARCHIVO"<<endl;
    return;
    }
  while(fread(&reg, sizeof reg, 1, pv)==1){
    if(reg.getAnio()==2020)
   mVentas[reg.getCodigoProducto()-1][reg.getMes()-
1]+=reg.getImporte();
  }
  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 resolver el problema con un algoritmo sencillo.

Supongamos que desconocemos la cantidad de productos, ya que varía en el tiempo. Lo anterior nos impide declarar la matriz estática. Pero como contamos con el archivo de productos, y sabemos 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), 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;
  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.getAnio()==2020)
       mVentas[reg.getCodigoProducto()-1][reg.getMes()-
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 (las filas), no se la puede declarar. Se declara un puntero a 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, 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.

Como lo que deseamos construir una matriz de cantreg filas y 12 columnas, el puntero tiene que ser un puntero fila de 12 columnas

La última línea agregada fue

delete mVentas;

para liberar la memoria pedida.

En el caso presentado como ejemplo se solicita memoria para crear un vector y una matriz de un tipo de datos primitivo (float), pero también es posible aplicar el mismo mecanismo a cualquier tipo de datos.

En la siguiente clase se usa asignación dinámica para hacer listados ordenados del archivo de un archivo de municipios.

```
class ArchivoMunicipio{
private:
    char nombre[50];
public:
    ArchivoMunicipio(const char *n="municipios.dat"){
        strcpy(nombre,n);
    }
    void grabarRegistro(Municipio obj);
    void modificarRegistro(Municipio obj, int pos);
    Municipio leerRegistro(int pos);
    int contarRegistros();
    void listarArchivo();
    bool listarOrdenado();
    int buscarRegistro(int num);
    void copiarRegistrosEnVector(Municipio *v, int cant);
    void ordenarPorHabitantes(Municipio *v,int cant);
    void ordenarPorNombre(Municipio *v,int cant);
};

void ArchivoMunicipio::grabarRegistro(Municipio obj){
    FILE *p=fopen(nombre,"ab");
    if(p==NULL){
        cout<<"ERROR DE ARCHIVO"<<endl;
        return;
    }
    fwrite(&obj,sizeof obj,1,p);
    fclose(p);
}
```

```
void ArchivoMunicipio::modificarRegistro(Municipio obj, int pos){
    FILE *p=fopen(nombre,"rb+");
    if(p==NULL){
        cout<<"ERROR DE ARCHIVO"<<endl;
        return;
    }
    fseek(p,sizeof obj*pos,0);
    fwrite(&obj,sizeof obj,1,p);
    fclose(p);
}

Municipio ArchivoMunicipio::leerRegistro(int pos){
    FILE *p=fopen(nombre,"rb");
    Municipio obj;
    obj.setNumero(-1);
    if(p==NULL){
        cout<<"ERROR DE ARCHIVO"<<endl;
        obj.setNumero(-2);
        return obj;
    }
    fseek(p,sizeof obj*pos,0);
    fread(&obj,sizeof obj,1,p);
    fclose(p);
    return obj;
}
int ArchivoMunicipio::contarRegistros(){
    FILE *p;
    p=fopen(nombre,"rb");
    if(p==NULL){
        cout<<"ERROR DE ARCHIVO"<<endl;
        return -1;
    }
    fseek(p,0,2);
    int cantBytes=ftell(p);
    fclose(p);
    return cantBytes/sizeof (Municipio);
}

void ArchivoMunicipio::listarArchivo(){
    int cant=contarRegistros();
    Municipio obj;
    for(int i=0; i<cant; i++){
        obj=leerRegistro(i);
        if (obj.getEstado()){
        obj.Mostrar();
        cout<<endl;
        }
    }
}
```

```
int ArchivoMunicipio::buscarRegistro(int num){
    int cant=contarRegistros();
    Municipio obj;
    for(int i=0; i<cant; i++){
        obj=leerRegistro(i);
        if(num==obj.getNumero()){
            return i;
        }
    }
    return -1;
}

bool ArchivoMunicipio::listarOrdenado(){
    Municipio *vMuni;
    int cant=contarRegistros();
    vMuni=new Municipio[cant];
    if(vMuni==NULL)return false;
    ///copiar los registros del archivo en el vector
    copiarRegistrosEnVector(vMuni, cant);
    ordenarPorNombre(vMuni,cant);
    ///ordenarPorHabitantes(vMuni,cant);
    for(int i=0;i<cant;i++){
        vMuni[i].Mostrar();
        cout<<endl;
    }
    delete []vMuni;
    return true;
}

void ArchivoMunicipio::copiarRegistrosEnVector(Municipio *v, int
cant){
    for(int i=0;i<cant;i++){
        v[i]=leerRegistro(i);
    }

}

void ArchivoMunicipio::ordenarPorHabitantes(Municipio *v,int cant){
    int i, j, posMin;
    Municipio aux;
    for(i=0;i<cant-1;i++){
        posMin=i;
        for(j=i+1;j<cant;j++){
            if(v[j].getCantHab()<v[posMin].getCantHab()){
                posMin=j;
            }
        }
        aux=v[i];
        v[i]=v[posMin];
        v[posMin]=aux;
```

```
    }
}

void ArchivoMunicipio::ordenarPorNombre(Municipio *v,int cant){
    int i, j, posMin;
    Municipio aux;
    for(i=0;i<cant-1;i++){
        posMin=i;
        for(j=i+1;j<cant;j++){
            if(strcmp(v[j].getNombre(), v[posMin].getNombre())<0){
                posMin=j;
            }
        }
        aux=v[i];
        v[i]=v[posMin];
        v[posMin]=aux;

    }
}
```