Clase 20. Destructor,constructor de copia, funciones friend, funciones inline, el apuntador this.

Destructor

Destructor

El destructor es una función miembro especial, que se ejecuta cuando un objeto sale del alcance o cuando se llama a **delete** con un apuntador de la clase.

- *El propósito del destructor es liberar los recursos requeridos en el tiempo de vida de la clase.*
- Existe un destructor definido implicitamente (que no hace nada). Siempre es público e inline.
- El destructor NUNCA lleva parámetros.
- Solo puede haber UN destructor por clase.

La sintáxis de declaración del destructor es la siguiente

~nombre_de_clase();

El destructor se llama en alguno de los siguientes casos:

Destructor

El destructor es una función miembro especial, que se ejecuta cuando un objeto sale del alcance o cuando se llama a **delete** con un apuntador de la clase.

- *El propósito del destructor es liberar los recursos requeridos en el tiempo de vida de la clase.*
- Existe un destructor definido implicitamente (que no hace nada). Siempre es público e inline.
- El destructor NUNCA lleva parámetros.
- Solo puede haber UN destructor por clase.

La sintáxis de declaración del destructor es la siguiente
~nombre_de_clase();
El destructor se llama en alguno de los siguientes casos:

- Se termina el programa.

Destructor

El destructor es una función miembro especial, que se ejecuta cuando un objeto sale del alcance o cuando se llama a **delete** con un apuntador de la clase.

- *El propósito del destructor es liberar los recursos requeridos en el tiempo de vida de la clase.*
- Existe un destructor definido implicitamente (que no hace nada). Siempre es público e inline.
- El destructor NUNCA lleva parámetros.
- Solo puede haber UN destructor por clase.

La sintáxis de declaración del destructor es la siguiente

~nombre_de_clase();

El destructor se llama en alguno de los siguientes casos:

- Se termina el programa.
- Se termina un hilo que tiene datos asociados al mismo.

Destructor

El destructor es una función miembro especial, que se ejecuta cuando un objeto sale del alcance o cuando se llama a **delete** con un apuntador de la clase.

- *El propósito del destructor es liberar los recursos requeridos en el tiempo de vida de la clase.*
- Existe un destructor definido implicitamente (que no hace nada). Siempre es público e inline.
- El destructor NUNCA lleva parámetros.
- Solo puede haber UN destructor por clase.

La sintáxis de declaración del destructor es la siguiente

~nombre_de_clase();

El destructor se llama en alguno de los siguientes casos:

- Se termina el programa.
- Se termina un hilo que tiene datos asociados al mismo.
- Se termina el alcance de la clase (cuando la clase está en el stack).

Destructor

El destructor es una función miembro especial, que se ejecuta cuando un objeto sale del alcance o cuando se llama a **delete** con un apuntador de la clase.

- *El propósito del destructor es liberar los recursos requeridos en el tiempo de vida de la clase.*
- Existe un destructor definido implicitamente (que no hace nada). Siempre es público e inline.
- El destructor NUNCA lleva parámetros.
- Solo puede haber UN destructor por clase.

La sintáxis de declaración del destructor es la siguiente
~nombre_de_clase();
El destructor se llama en alguno de los siguientes casos:

- Se termina el programa.
- Se termina un hilo que tiene datos asociados al mismo.
- Se termina el alcance de la clase (cuando la clase está en el stack).
- Se llama a **delete** con un apuntador a la clase de la memoria dinámica de la clase.

Destructor


```
1  #include <iostream>

3  using namespace std;
   class MClase
5  {
     public:
7      MClase() {cout<< "Hola"<<endl;} //Constructor
   };
9
   int main()
11 {
     MClase Objeto; //Se llamo al constructor
13   return 0; //Se llamo al destructor por default;
   }
```

Destructor


```
#include <iostream>

using namespace std;
 class MClase
 {
   public:
     MClase() {cout<< "Hola"<<endl;} //Constructor
     ~MClase() {cout << "Se murio"<<endl;}
 };

 int main()
 {
   MClase Objeto; //Se llamo al constructor
   return 0; //Llama al destructor de usuario
 }
```

Destructor


```
#include <iostream>
using namespace std;
class MClase{
 double *x;
 int n;
 int m;
 public:
  MClase() {cout<< "Default"<<endl; n=-1; m=-1;}
  MClase(int _n) {n=_n; x=new double[n];
          cout<< "Pide: "<<n<< endl;}
  MClase(int _n,int _m){n=_n; m=_m; x=new double[n*m];
         cout<< "Pide "<<n<< " y "<< m<< endl;}
  ~MClase();
};
MClase::~MClase() {
  if (n>0) {  delete []x;
    if (m>0) cout << "Devuelve:" << n*m << endl;
    else cout << "Devuelve:" << n << endl;
  }
  else  cout << "No devuelve"<< endl;
}
```

Destructor

Nota: Ver el orden en que devuelve la memoria.


```
int main()
{
  MClase Obj1, Obj2(5), Obj3(3,4); //Se llamo al
   constructor
  return 0;
}
```

```
Default
Pide: 5
Pide 3 y 4
Devuelve:12
Devuelve:5
No devuelve
```

Memoria dinámica para objetos

new y delete

new y **delete** son similares para objetos que para datos nativos. Solo que **new** llama al constructor y **delete** llama al destructor.


```
#include <iostream>
using namespace std;
 class MClase {
    double *x;
    public:
      MClase() {x=NULL; cout<< "construye 1"<<endl;} //
     Constructor
      MClase(int n){x= new double[n]; cout<< "construye
     2"<<endl;}
      ~MClase(){if (x) delete[]x; cout<< "destruye"<<
     endl;}
 };
int main(){
    MClase *Obj1, *Obj2; //Solo apuntadores
    Obj1= new MClase;
    Obj2= new MClase(10);
    delete Obj1;
    delete Obj2;
    return 0;
 }
```

new y delete

Si la memoria para el objeto se requiere dinámicamente, es decir con **new** el destructor TIENE que ser llamado forzosamente para liberarla.

```
1 int main(){
    MClase *Obj1, *Obj2; //Solo apuntadores
3   Obj1= new MClase;
    Obj2= new MClase(10);
5   return 0;
  }
```

```
==6873== HEAP SUMMARY:
==6873==     in use at exit: 96 bytes in 3 blocks
==6873==   total heap usage: 3 allocs, 0 frees, 96 by
==6873== LEAK SUMMARY:
==6873==    definitely lost: 16 bytes in 2 blocks
==6873==    indirectly lost: 80 bytes in 1 blocks
==6873==      possibly lost: 0 bytes in 0 blocks
==6873==    still reachable: 0 bytes in 0 blocks
==6873==         suppressed: 0 bytes in 0 blocks
==6873== Rerun with --leak-check=full to see details
```

new y delete

NOTE: Si se llama new, el destructor NO se llama automáticamente, pero si se pude memoria estática entonces SI se llama automaticamente, aunque indirectamente (en el constructor) se pida memoria dinámica.

```
int main(){
2   MClase *Obj1; //Solo apuntador
    MClase Obj2(10); //Pide vector de tamanio 10
4   Obj1= new MClase;
    delete Obj1;
6   return 0;//Se llama destructor para Obj2
 }
```

```
==7158== HEAP SUMMARY:
==7158==     in use at exit: 0 bytes in 0 blocks
==7158==   total heap usage: 2 allocs, 2 frees, 88 by
```

new y delete

Llamar a new para un bloque de objetos hace **n** llamadas al constructor, y delete realiza **n** llamadas al destructor.


```
1  #include <iostream>
   using namespace std;
3  class MClase{
    double *x;
5   public:
     MClase() {x=NULL; cout<< "construye 1"<<endl;}
7    MClase(int n){ cout<< "construye 2"<<endl;}
     ~MClase(){if (x) delete[]x; cout<< "destruye"<<endl;}
9  };
   int main(){
11  MClase *Obj1= new MClase[3];
    delete []Obj1;
13   return 0;
   }
```

```
construye 1
construye 1
construye 1
destruye
destruye
destruye
```

new y delete

Para acceder a miembros de la clase, cuando se requiere dinámicamente memoria hay que usar el operador $->$. Solo recuerde que en caso de arreglos, al igual que con las estructuras los corchetes [] requieren el contenido del apuntador. Ej

```
int main(){
 MClase *Obj1= new MClase[3]; //Arreglo
 MClase *Obj2=new MClase; //Un objeto
 Obj1[0].miembro=5; //Accede a miembro
 Obj2->miembro=5; //Accede a miembro
 delete []Obj1;
 delete Obj2;
  return 0;
}
```

Constructor de copia

Constructor de copia

Se usa en 3 casos:

- CUando se inicializa un objeto a partir de otro. Ej. **MClase ObjetoC=Objeto;**

Constructor de copia

Se usa en 3 casos:

- CUando se inicializa un objeto a partir de otro. Ej. **MClase ObjetoC=Objeto;**
- Cuando se pasa por copia un objeto a una función. Ej. **int funcion(MClase Copia);**

Constructor de copia

Se usa en 3 casos:

- CUando se inicializa un objeto a partir de otro. Ej. **MClase ObjetoC=Objeto;**
- Cuando se pasa por copia un objeto a una función. Ej. **int funcion(MClase Copia);**
- Cuando se regresa un objeto de una función. Ej. **MClase funcion(){ MClase Original; return Original;} MClase Copia=funcion();**

Constructor de copia


```
#include <iostream>
using namespace std;
class MClase{
    int n;  double *x;
  public:
   MClase() {x=NULL; n=-1;} //Constructor
   MClase(int _n){n=_n;    x= new double[n]; }
   MClase(MClase &Copia){
     cout<< "Copia" << endl;
     if (Copia.n>0){
      n=Copia.n;
      x= new double[n];
      for (int i=0; i<n; i++) x[i]=Copia.x[i];
     }
  }
  ~MClase(){if (x) delete[]x;}
};
int main(){
   MClase Obj1(10);
   MClase Obj2=Obj1;
   return 0;
}
```

Función friend

Función friend

Una función **friend** se define fuera de la clase, pero puede acceder a todos los miembros protegidos y privados de la clase.

Un **friend** puede ser una funcion, un template de función, un miembro de una función, una clase, o template de clase.

Friend


```
#include <iostream>
using namespace std;
class MClase{
    int n;  double *x;
  public:
   MClase() {x=NULL; n=-1;} //Constructor
   MClase(int _n){n=_n;    x= new double[n]; }
  ~MClase(){if (x) delete[]x;}
  friend double Suma(MClase);
};
double Suma(MClase obj){
    double sum=0.0;
    cout << "n= " << obj.n<< endl;
    for (int i=0; i< obj.n; i++)
        sum+=obj.x[i];
}
int main(){
    MClase Obj1(10);
    cout<< "Suma "<< Suma(Obj1)<< endl;
    return 0;
}
```

Friend, sin error, constructor de copia


```
#include <iostream>
using namespace std;
class MClase{
    int n;  double *x;  int copia;
  public:
   MClase() {x=NULL; n=-1; copia=-1;} //Constructor
   MClase(int _n){n=_n;   x= new double[n]; copia=-1;}
   MClase(MClase &Original){ n=Original.n; x=Original.x
    ; copia=1; }
  ~MClase(){if (x && copia!=1) delete[]x;}
  friend double Suma(MClase);
};
double Suma(MClase obj){
   double sum=0.0;
   cout << "n= " << obj.n<< endl;
   for (int i=0; i< obj.n; i++)
       sum+=obj.x[i];
}
int main(){
   MClase Obj1(10);
   cout<< "Suma "<< Suma(Obj1)<< endl;
   return 0;
}
```

inline

Inline Function

Una función inline se copia en el lugar donde es llamada en tiempo de compilación. Esto puede aumentar la velocidad de ejecución del código.

- Las funciones definidas en la declaración de la clase son **inline** por default.
- Si una función inline se redefine, los clientes de la función (donde se llame) se deben de recompilar.
- La palabra **inline** provee una sugerencia al compilador, no es forzoso que la función sea inline.

```
#include <iostream>
using namespace std;
class MClase{
  int n;  double *x;  int copia;
  public:
  MClase() {x=NULL; n=-1; copia=-1;}
  double suma();
};
inline double MClase::suma(){
      double sum=0.0;
      for (int i=0; i<n; i++)
        sum+=x[i];
}
```

tarea

Tarea

Tarea 11. Escriba un programa que realice un simulación Montecarlo de variables reales definidas en cierto dominio. Debe considerar lo siguiente:

- Defina una clase que tendrá un constructor por defecto que inicializa todas las variables en 0 y en NULL los apuntadores.
- Un constructor que recibe como parámetros el número de dimensiones de los datos. Y fija el número de datos simulados en 1000 por defecto. Y los límites de simulación de las variables.
- Un constructor que recibe el número de datos y el número de datos simulados.
- Una friend función que es proporcional a la función que se va a simular, ej. $f(x) = exp(-g(x)/T)$.
- Una función miembro para fijar y una para obtener (set y get) el número de variables, si ya se tenía memoria para los límites de las variables se debe devolver esa memoria y pedir núeva.
- Una función para fijar y una para obtener los límites de las variables.
- Una función para fijar y una para obtener el número datos a simular.
- Una función miembro histograma.
- Una función miembro que calcule el promedio de las simulaciones (estimador de la media).
- Una función que devuelva un dato generado por la simulación. Este dato será generado cada que se cambien los límites, o número de variables de la clase. O si algún valor de esa instancia ya fue entregado, es decir supongamos que mi función se llama **x**, y mi objeto se llama **y**. Si llamo **y.x(1);** me regresa el valor de mi instancia de Montecarlo para la variable 1, si llamo otra vez **y.x(1)** me regresa una instancia diferente, pero si llamo **y.x(1); y.x(2);** las instancia para la variable 1 y para la variable 2, pertenecen al mismo vector instanciado.

Todos los datos son privados (se accede a ellos por funciones), todas las funciones no requeridas en la tarea son privadas o protegidas.

El apuntador this

El apuntador this

Cada objeto en C++ tiene acceso a su propia dirección a través de un apuntador que se llama **this**. Está decalrado de forma explícita en todas las funciones miembro.
Las funciones friend no pueden obtener esta dirección ya que no son miembros de la clase.

this

```
1  #include <iostream>
   using namespace std;
3  class MClase{
     public:
5     int n; double *x;
      MClase(int _n){n=_n;    x= new double[n]; }
7    ~MClase(){if (x) delete[]x;}
     void printDirs();
9  };
   void MClase::printDirs(){
11  cout<< "Mi apuntador this=" <<this<< " x="<<this->x<<"
        n= "<<this->n<<endl;
   }
13 int main(){
     MClase Obj1(10);  Obj1.printDirs();
15   cout<< "Corroboro this=" << &Obj1 << " x= "<< Obj1.x
       << " n= "<<Obj1.n<<endl;
     return 0;
17 }
```

```
Mi apuntador this=0x7fff90259d40 x= 0xd39010 n= 10
Corroboro this=0x7fff90259d40 x= 0xd39010 n= 10
```

this, uso práctico


```
1  #include <iostream>
   using namespace std;
3  class MClase{
       int n;  double *x;  int copia;
5    public:
     MClase() {x=NULL; n=-1; copia=-1;} //Constructor
7    MClase(int _n){n=_n;   x= new double[n]; copia=-1;}
     ~MClase(){if (x && copia!=1) delete[]x;}
9    double getSuma();
     friend double Suma(MClase);
11 };
   double Suma(MClase obj){
13   double sum=0.0;
     for (int i=0; i< obj.n; i++)
15       sum+=obj.x[i];
   }
17 double MClase::getSuma() {
     return Suma(*this);
19 }
   int main(){
21   MClase Obj1(10);
     cout<< "Suma "<< Obj1.getSuma()<< endl;
23   return 0;
   }
```