Clase 21. Miembros estáticos

static

Miembros static

Cuando un miembro de la clase se declara **static**, no importa cuantos objetos se instancien, solo habra una copia única de ese miembro y conservará su valor.

- Un objeto **static** es compartido por todos los objetos de la misma clase y sus clases derivadas.
- Todos los miembros **static** se inicializan a cero.
- Los miembros static están en otra parte de la memoria diferente a donde estánlos objetos y existen fuera de estos, por lo cual se pueden inicializar y utilizar sin necesidad de instanciar un objeto.
- Sin embargo, tienen las etiquetas de tipo de acceso (publico o privado) como cualquier otro miembro de la clase.
- Los miembros **static** pueden ser datos o funciones.

Miembros static

La declaración de la variable no implica su definición. La variable se define arriba del main y tiene alcance global. Sino se usa no hay problema al compilar.

```
1  #include <iostream>
   using namespace std;
3  class MClase{
       public:
5        static int var;
   };
7  int MClase::var=5;
   int main(){
9    cout<< MClase::var <<endl;
     return 0;
11 }
```

Miembros static

Las funciones **static** tambien pueden ser llamadas sin tener ninguna instancia de la clase, solo tienen acceso a miembros static y otras funciones static.

```
#include <iostream>
using namespace std;
class MClase{
  static int var;
   public:
     static int getVar(){
       return var;
     }
};
int MClase::var=5;
int main()
{
  cout<< MClase::getVar() <<endl;
  return 0;
}
```

Herencia

Herencia

La herencia permite definir una clase (la clase derivada) a partir de otra clase (la clase base):

- Sirve para reutilización de código, para estándarización de datos y m'etodos en diferentes clases similares, y acelerar la codificación.
- Cuando se escribe una nueva clase, en lugar de definir nuevas variables y métodos el programador podría utilizar los métodos y variables definidos enase, y solo extender la funcionalidad o los datos. A la clase existente se le denomina clase **base** y a la nueva clase **derivada**.

La sintaxis básica para declarar una clase derivada:

```
1 class clase_derivada : especificador_de_acceso
      clase_base
{
3 };
```

Herencia


```
1  class Figura{
       int nvertices;
3      double *coordX, *coordY;
       public:
5      Figura(){}
       Figura(int _nvertices){
7        nvertices=nvertices;
         coordX=new double[nvertices];
9        coordY=new double[nvertices];
         }
11     ~Figura(){
         delete []coordX;
13       delete []coordY;
     }
15 };
   class Rectangulo: public Figura{
17     public:
         double Area();
19 };
```

Herencia, especificadores de acceso

public Todos los miembros heredados conservan su especificación de acceso. Los publicos siguen siendo publicos, etc.


```
1  class Figura{
   int nvert;
3  double *coordX, *coordY;
   public:
5  double area;
   Figura(){
7   nvert=-1;}
   Figura(int _nvert){
9   nvert=nvert;
    coordX=new double[nvert];
11  coordY=new double[nvert];
   }
13 ~Figura(){
     if (nvert!=-1){
15    delete []coordX;
      delete []coordY;
17   }
    }
19 };
```

```
1  class Rectangulo:public
       Figura
   {
3   public:
      double Area();
5  };
   double Rectangulo::Area(){
7    return 0;
   }
9
   int main(){
11   Rectangulo Obj;
     Obj.area=5;
13   return 0;
   }
```

¿Que no es heredado?

Las clases heredadas heredan todo excepto:

- El constructor y el destructor.
- Los operadores de asignación (miembros de la clase).
- Los **friend**.
- Los datos privados.

No quiere decir que no se puedan utilizar los anteriores, simplemente: el constructor de la clase heredada, llama al constructor por default de la clase base (a menos que se diga otra cosa), pero uno no sustituye al otro. Los datos privados no son accesibles desde métodos de la clase heredada, y las funciones friend no lo son para la clase heredada, pero aun todos ellos existen y pueden ser invocados.

Herencia, especificadores de acceso

- La clase heredada manda llamar al constructor sin argumentos de la clase base, pero también llama a su propio constructor.
- Note que el constructor de la clase base accede a un miembro privado de la clase base.
- También se llama a ambos destructores.


```
class Figura{
int nvert;
double *coordX, *coordY;
public:
double area;
Figura(){ nvert=-1;
cout<< "construye base" << endl; } };
class Rectangulo:public Figura {
 public:
   double Area();
};
double Rectangulo::Area(){
  return 0;
}
int main(){
  Rectangulo Obj;
  return 0;
}
```

Herencia, constructor y destructor


```
#include <iostream>
using namespace std;
class Figura{
   public:
   Figura(){cout <<"Construye base"<<endl; }
  ~Figura(){ cout<<"Destruye base"<< endl;  }
};
class Rectangulo: public Figura{
    public:
     Rectangulo(){ cout<< "Construye derivada"<<endl; }
     ~Rectangulo(){ cout<< "Destruye derivada"<<endl; }
};
int main(){
  Rectangulo Obj;
  return 0;
}
```

```
Construye base
Construye derivada
Destruye derivada
Destruye base
```

Herencia, constructor y destructor


```
#include <iostream>
using namespace std;
class Figura{
   public:
   Figura(){cout <<"Construye base"<<endl; }
  ~Figura(){ cout<<"Destruye base"<< endl;  }
};
class Rectangulo: public Figura{
    public:
     Rectangulo(){ cout<< "Construye derivada"<<endl; }
     ~Rectangulo(){ cout<< "Destruye derivada"<<endl; }
};
int main(){
  Rectangulo Obj;
  return 0;
}
```

```
Construye base
Construye derivada
Destruye derivada
Destruye base
```

Herencia, private, recordatorio

Los miembros **privados** no pueden ser accedidos por las clases derivadas.


```
#include <iostream>
using namespace std;
class Figura{
    int nv;
    public:
    Figura(){}
    ~Figura(){  }
};
class Rectangulo: public Figura{
     public:
       Rectangulo(){nv=4;  }
};
int main()
{
  Rectangulo Obj;
  return 0;
}
```

```
ej03.cpp: In constructor 'Rectangulo::Rectangulo()':
ej03.cpp:5:8: error: 'int Figura::nv' is private
```

Herencia, protected, recordatorio

Los miembros **protected** son privados para todos excepto para las clases base y heredadas.


```
1  #include <iostream>
   using namespace std;
3  class Figura{
     protected:
5      int nv;
       public:
7      Figura(){}
       ~Figura(){  }
9  };
   class Rectangulo: public Figura{
11       public:
           Rectangulo(){
13 nv=4;
           }
15 };
   int main()
17 {
     Rectangulo Obj;
19   return 0;
   }
```

Herencia, private

Si la clase base se declara **private** al momento de derivar una nueva clase, todos los miembros de la clase base son privados en la nueva clase. Lo mismo sucede con **protected**

```
#include <iostream>
using namespace std;
class Figura{
    public:
    int nv;
};
class Rectangulo: private Figura{
     public:
        Rectangulo(){
nv=4;
        }
};
int main()
{
   Rectangulo Obj;
   Obj.nv=5;
   return 0;
}
```

Herencia, constructor

La llamada a un constructor con parámetros de la clase heredada se puede especificar en el constructor de la clase base.


```
#include <iostream>
using namespace std;
class Figura{
    public:
    int nv;
    Figura(){ }
    Figura(int _nv){
      nv=_nv;
   }
};
class Rectangulo: private Figura{
   public:
    Rectangulo() : Figura(4){}
    Rectangulo(int _nv, int _mv) : Figura(_nv){}
};
int main()
{
   Rectangulo Obj(5,4);
   return 0;
}
```

Tarea 11

11.2 Realice un programa que genere muestras pseudo-aleatorias de una distribución normal multivariada.

- Defina una clase matriz, que pide memoria dinámica en el constructor. Utilice la clase matriz para cualquier matriz de las clases subsecuentes. Verifique que toda la memoria sea requerida, utilizada y liberada adecuadamente.
- Defina una clase vector, que pide memoria dinámica en el constructor. Utilice la clase vector para cualquier vector de las clases subsecuentes. Verifique que toda la memoria sea requerida, utilizada y liberada adecuadamente.
- La clase base servirá para tomar muestras de una normal multivariada pero de variables independientes. Esto es si son N variables cada variable se genera de forma independiente (con el algoritmo que ya programaron). Para esto se requerirá un vector de medias μ y una matríz diagonal (que puede ser almacenada en un vector) de varianzas o desviaciones estándar Σ.
- Una clase heredada de la anterior, puede considerar dependencias entre las variables. Si Σ es la matr'iz de covarianza y suponiendo que siempre es positiva definida (lo que ocurre casi con certeza). La descomposición Cholesky de Σ es $AA^T = \Sigma$. Si z es un vector muestra simulado de normales estándar independientes, entonces:

  $$x = \mu + Az$$

  Se distribuye normal con media μ y matriz de covarianza Σ.
- Verifiquen que las muestras generadas por su algoritmo en realidad se distribuyen así, recalculando la media y matríz de covarianza de 10000 muestras.

Sobrecarga de funciones

Sobrecarga de funciones

C++ permite especificar mas una definición de funciones u operadores con el mismo alcance. A esto se le llama **sobrecarga de funciones u operadores**. La función es utilizada de acuerdo al tipo de parámetros que recibe (como ya lo hemos visto con los constructores).

```
#include <iostream>
using namespace std;
double funcion(){
    cout<<"funcion"<< endl;
return 0.0;
}
double funcion(double x){
    cout<<"funcion 2"<< endl;
return x;
}
int main(){
  int i;
  double x;
  x=funcion(5.0);
  cout << x << endl;
  return 0;
}
```

Tarea 11

11.3

En el programa anterior, sobrecargue en la clase derivada la función que realiza el muestreo, si recibe dos vectores (μ y Σ) considera las variables independientes, si recibe dos matrices considera las variables dependientes.