Clase 23. Abstracción, encapsulamiento, clases puramente abstractas, templates. Lectura y escritura de archivos.

Abstracción y encapsulamiento

Abstracción y encapsulamiento

La abstracción es una técnica de diseño de software que se basa en esconder toda la información no relevante para el cliente. La principal característica es separar la interfaz de la implementación.


```
1  class Solver{
   private:
3    Matrix A,L,U;
     Vector index, X, B,
5    double MaxError, MaxIter;
     string ErrorMsg, ExitMsg;
7    void LU();
     void Cholesky();
9    void GC();
   public:
11   Solver();
     Solver(string FileName);
13   Solver(Matrix A, Vector B);
     Vector Solve();
15   ~Solver();
     string getMessage();
17   setMaxIte(int maxiter);
     setMaxError(double Error);
19 };
```

- El método Solve() abstrae la forma en que funciona.
- El cliente sabe que resuelve un sistema de ecuaciones, pero no sabe como.
- Se pueden agregar mas métodos de solución, pero eso no cambiará la interfaz a los métodos que es Solve().
- Solve() es la interfaz, LU, Cholesky, etc. son implementación.

Abstracción y encapsulamiento

El encapsulamiento consiste en poner juntos los datos y métodos que realizan una función

```
1  class Solver{
   private:
3    Matrix A,L,U;
     Vector index, X, B,
5    double MaxError, MaxIter;
     string ErrorMsg, ExitMsg;
7    void LU();
     void Cholesky();
9    void GC();
   public:
11   Solver();
     Solver(string FileName);
13   Solver(Matrix A, Vector B);
     Vector Solve();
15   ~Solver();
     string getMessage();
17   setMaxIte(int maxiter);
     setMaxError(double Error);
19 };
```

- La clase mantiene la matríz, el vector, el término independiente, etc. en el objeto que ocupa todo.
- Las etiquetas ayudan tanto la abstracción como el encapsulamiento.

Ventajas

- La abstracción y encapsulamiento evitan que el cliente provoque errores en los objetos o inserte corrupciones de memoria. Ej. Cambio la dimensón de una matriz y automaticamente se modifica la memoria, se puede asegurar que esto siempre ocurra en mi objeto Matriz y que no se intente acceder a memoria que no se ha requerido. Se devuelve la memoria que se pide siempre con llamadas automaticas al destructor. Se puede evitar que se intente resolver un sistema que no se ha inicializado, etc.

Ventajas

- La abstracción y encapsulamiento evitan que el cliente provoque errores en los objetos o inserte corrupciones de memoria. Ej. Cambio la dimensón de una matriz y automaticamente se modifica la memoria, se puede asegurar que esto siempre ocurra en mi objeto Matriz y que no se intente acceder a memoria que no se ha requerido. Se devuelve la memoria que se pide siempre con llamadas automaticas al destructor. Se puede evitar que se intente resolver un sistema que no se ha inicializado, etc.
- Facilita la evolución del código. Ej. puedo agregar métodos de solución, eficientarlos, paralelizarlos, etc. Sin que el cliente tenga que modificar nada de su código.

Ventajas

- La abstracción y encapsulamiento evitan que el cliente provoque errores en los objetos o inserte corrupciones de memoria. Ej. Cambio la dimensón de una matriz y automaticamente se modifica la memoria, se puede asegurar que esto siempre ocurra en mi objeto Matriz y que no se intente acceder a memoria que no se ha requerido. Se devuelve la memoria que se pide siempre con llamadas automaticas al destructor. Se puede evitar que se intente resolver un sistema que no se ha inicializado, etc.
- Facilita la evolución del código. Ej. puedo agregar métodos de solución, eficientarlos, paralelizarlos, etc. Sin que el cliente tenga que modificar nada de su código.
- Mantenimiento y reutilización.

Desventajas

- Se pierde la noción de si se está haciendo uso eficiente de la memoria, de la cantidad de memoria que se utiliaz, de si la memoria es continua o está fragmentada, de si se realizan solo la operaciones necesarias, de si se utiliza adecuadamente el cache, etc.

Desventajas

- Se pierde la noción de si se está haciendo uso eficiente de la memoria, de la cantidad de memoria que se utiliaz, de si la memoria es continua o está fragmentada, de si se realizan solo la operaciones necesarias, de si se utiliza adecuadamente el cache, etc.
- Se pierde la noción de la memoria requerida y devuelta (muchas veces se deja que los objetos se construyan y se destruyan sin que yo este pensando en eso), lo cual hace pensar al programador/cliente que la memoria se maneja de forma automática y correcta (no siempre es verdad).

Desventajas

- Se pierde la noción de si se está haciendo uso eficiente de la memoria, de la cantidad de memoria que se utiliaz, de si la memoria es continua o está fragmentada, de si se realizan solo la operaciones necesarias, de si se utiliza adecuadamente el cache, etc.
- Se pierde la noción de la memoria requerida y devuelta (muchas veces se deja que los objetos se construyan y se destruyan sin que yo este pensando en eso), lo cual hace pensar al programador/cliente que la memoria se maneja de forma automática y correcta (no siempre es verdad).
- TODAS las aplicaciones eficientes hacen uso eficiente de la memoria y el cache, el manejo de la memoria es, posiblemente, lo que mas impacta el desempeño de una aplicación.

Clases (totalmente) abstractas (interfaces)

Las clases puramente abstractas solo tienen métodos puramente abstractos (funciones puramente virtuales), sin incluir el constructor, los cuales no pueden ser declarados puramente virtuales. **Las clases puramente abstractas sirven para interfaces, y no pueden ser instanciadas.(El ejemplo de abajo NO compila)**


```
1  class Matrix{
     protected:
3      int nrow,ncol;
     public:
5     virtual void set_nrow()=0;
      virtual int  get_nrow()=0;
7     virtual void set_ncol()=0;
      virtual int get_ncol()=0;
9     virtual int Fill(const char *filename)=0;
      virtual void set_value(int i, int j)=0;
11    virtual double get_value(int i, int j)=0;
};
13 int main(){
    Matrix Obj;
15  return 0;
}
```

Clases (totalmente) abstractas (interfaces)


```
class Matrix{
 protected:
  int nrow,ncol;
 public:
  virtual void setNR(int nr)=0;
  virtual int  getNR()=0;
};
class MatrixS:public Matrix{
private:
 typedef struct{
  double xd; int i;
 }SPARSED;
 SPARSED *rows;
public:
 void setNR(int nr){nrow=nr;}
 int getNR(){return nrow;}
};
int main(){
 MatrixS Obj;
 Matrix* Obj2= new MatrixS;
 delete Obj2;
 return 0;
}
```

- Se declara un objeto de la clase derivada, la clase puramente abstracta sirve para especificar cuales funciones **deben** de estar presentes en la clase derivada y que el cliente utilice las funciones especificadas en la clase puramente abstracta,

Clases (totalmente) abstractas (interfaces)


```
1  class Matrix{
   protected:
3    int nrow,ncol;
   public:
5    virtual void setNR(int nr)=0;
     virtual int  getNR()=0;
7  };
   class MatrixS:public Matrix{
9  private:
    typedef struct{
11    double xd; int i;
    }SPARSED;
13  SPARSED *rows;
   public:
15  void setNR(int nr){nrow=nr;}
    int getNR(){return nrow;}
17 };
   int main(){
19  MatrixS Obj;
    Matrix* Obj2= new MatrixS;
21  delete Obj2;
    return 0;
23 }
```

- Se declara un objeto de la clase derivada, la clase puramente abstracta sirve para especificar cuales funciones **deben** de estar presentes en la clase derivada y que el cliente utilice las funciones especificadas en la clase puramente abstracta,
- Se declara un apuntador a la clase abstracta pero se hace la instancia de la derivada.

Clases (totalmente) abstractas (interfaces)

typedef, estructuras y uniones dentro de la clase. La definiciones solo tienen alcance dentro de la clase, o la estructura. Si se hacen publicas se puede utilizar el especificador de resolusión de ámbito o alcance.


```
1  class Matrix{
   protected:
3    int nrow,ncol;
   public:
5    virtual void setNR(int nr)=0;
     virtual int  getNR()=0;
7  };
   class MatrixS:public Matrix{
9  private:
    typedef struct{
11   enum{INT,DOU,CHAR}type;
     union{double xd;int xi; char
       xc;};
13   int i;
    }SPARSED;
15  SPARSED *rows;
   public:
17  void setNR(int nr){nrow=nr;}
    int getNR(){return nrow;}
```

```
1  int main(){
    MatrixS Obj;
3   Matrix* Obj2= new
      MatrixS;
    delete Obj2;
5   return 0;
   }
```

Tarea

Tarea 12.2

- Defina una clase MatrizA puramente abstracta, y clases derivadas de matrices simétricas y asimétricas.
- Defina otra clase Matriz, que será la interfaz para MatrizA. E instanciará una matriz simétricas o asimétrica dependiendo de los datos.
- Los operadores de MatrizA están sobrecargados para realizar las operaciones de suma, resta, multiplicación y división entre matrices simétrica/simétrica, simétrica/asimétrica, asimétrica/asim´etrica, etc. Con el menor número de operaciones y uso de memoria posible.

- Agregue ejemplos de uso en su programa.

Templates

Templates en funciones

Los **templates** son utilizados para escribir funciones genericas para diferentes tipos. Ej. requiero ordenar una lista de objetos. Una vez conocida una función u operador que compara esos objetos, no importa que objetos sean el algoritmo para ordenarlos es el mismo. Los templates me permiten definir ese algoritmo sin necesidad de hacerlo para un tipo específico. Se pueden definir templates de funciones y templates de clases. La sintaxis es la siguiente:

```
template <typename TIPO>
Tipo_Retorno funcion(TipoParam Param,... ){
}
template <class TIPO>
Tipo_Retorno funcion(TipoParam Param,... ){
}
```

Templates en funciones

Los **templates** son utilizados para escribir funciones genericas para diferentes tipos. Ej. requiero ordenar una lista de objetos. Una vez conocida una función u operador que compara esos objetos, no importa que objetos sean el algoritmo para ordenarlos es el mismo. Los templates me permiten definir ese algoritmo sin necesidad de hacerlo para un tipo específico. Se pueden definir templates de funciones y templates de clases. La sintaxis es la siguiente:

```
template <typename TIPO>
Tipo_Retorno funcion(TipoParam Param,... ){
}
template <class TIPO>
Tipo_Retorno funcion(TipoParam Param,... ){
}
```

- Los templates son reemplazados en tiempo de compilación (son una macro básicamente), NO son reemplazados en tiempo de linkeo ahi ya debe de saber el compilador que tipo recibe.

Templates en funciones

Los **templates** son utilizados para escribir funciones genericas para diferentes tipos. Ej. requiero ordenar una lista de objetos. Una vez conocida una función u operador que compara esos objetos, no importa que objetos sean el algoritmo para ordenarlos es el mismo. Los templates me permiten definir ese algoritmo sin necesidad de hacerlo para un tipo específico. Se pueden definir templates de funciones y templates de clases. La sintaxis es la siguiente:

```
template <typename TIPO>
Tipo_Retorno funcion(TipoParam Param,... ){
}
template <class TIPO>
Tipo_Retorno funcion(TipoParam Param,... ){
}
```

- Los templates son reemplazados en tiempo de compilación (son una macro básicamente), NO son reemplazados en tiempo de linkeo ahi ya debe de saber el compilador que tipo recibe.
- Básicamente para cada llamada a la función con un tipo de diferente se genera el código para ese tipo específico.

Template, ejemplo

TIPO es reemplazado por **double** y por **char** en tiempo de compilación.


```
template <class TIPO>
TIPO square(TIPO x)
{
  TIPO result=x*x;
  return result;
}

int main(){
  std::cout << square(9.8) << std::endl;
  std::cout << square('x') << std::endl;
 return 0;
}
```

Template, ejemplo

YA que el reemplazo se realiza en tiempo de compilación, el siguiente código NO compila:


```
#include <iostream>
template <class TIPO>
TIPO square(TIPO x);

TIPO square(TIPO x){
  TIPO result=x*x;
  return result;
}
int main(){
  std::cout << square(9.8) << std::endl;
  std::cout << square('x') << std::endl;
 return 0;
}
```

Debido a que el compilador CONOCE el tipo de la cabecera, pero desconoce el tipo en la definición lo mismo pasa si se usan archivos .h y .cpp.

Template, ejemplo

Este ejemplo si compila.

```
1  #include <iostream>
   template <class TIPO>
3  TIPO square(TIPO x);

   template <class TIPO>
5  TIPO square(TIPO x){
     TIPO result=x*x;
7    return result;
   }
9

   int main(){
11   std::cout << square(9.8) << std::endl;
     std::cout << square('x') << std::endl;
13  return 0;
   }
15
```

Template, ejemplo

Infier el tipo por el argumento.

```
#include <iostream>

template <class TIPO>
double square(TIPO x)
{
  TIPO result=x*x;
  return result;
}
int main(){
  std::cout << square(9.8) << std::endl;
  std::cout << square('x') << std::endl;
 return 0;
}
```

Template, ejemplo

No puede saberse el tipo.

```
1  #include <iostream>
   template <class TIPO>
3  double square(double x)
   {
5    TIPO result=x*x;
     return result;
7  }
   int main(){
9    std::cout << square(9.8) << std::endl;
     std::cout << square('x') << std::endl;
11   return 0;
   }
```

```
.cpp:4:8: note: template<class TIPO> double square(double)
.cpp:4:8: note:   template argument deduction/substitution failed:
.cpp:10:26: note:   couldn't deduce template parameter 'TIPO'
```

Template, ejemplo

Se indica explícitamente el tipo.

```
1  #include <iostream>
   template <class TIPO>
3  TIPO square(double x)
   {
5    TIPO result=x*x;
     return result;
7  }
   int main(){
9    std::cout <<  square<double>(9.8) << std::endl;
     std::cout <<  square<int>('x') << std::endl;
11   return 0;
   }
```

Template en clases

El template en un clase, permite definir una clase para cada tipo de dato.

```
#include <iostream>

template <class TIPO>
class DATUM{
        TIPO x;
    public:
      TIPO get_datum(){return x;}
      void set_datum(TIPO _x){x=_x;}
};
int main(){
  DATUM<int>X;
  DATUM<double>Y;
  X.set_datum(5.3);
  Y.set_datum(5.3);
  std::cout << X.get_datum() <<" "<<  Y.get_datum()<<
    std::endl;
 return 0;
}
```

Tarea

Tarea 12.3

- Defina una clase template ARREGLO para arreglos de cualquier tipo.
- Defina una función template SORT para ordenar por el método de la burbuja cualquier tipo de dato.
- Sobrecargue los operadores de comparación $<, >, ==$ para la clase arreglo, un arreglo es mayor que otro si tiene mas datos, menor si tiene menos, etc.
- Provea ejemplos de uso para al menos 3 tipos diferentes.

Lectura y escritura de archivos

Para leer y escribir a archivo en C++ se utiliza la libreria estándar **fstream**. Que define los siguientes tipos (clases):

- **ofstream** Para escribir a archivo.
- **ifstream** Para leer a archivo.
- **fstream** Puede leer y escribir a archivo.

Apertura de archivos

- Las funciones **open y close** se puede utilizar para abrir y cerrar un archivo, está función es miembro de la clase.
- Los operadores $<<, >>$ pueden ser utilizados para leer y escribir similar a como se hace con cin y cout,
- Las función **open** tiene el modo de apertura por default para ofstream y ifstream.

```
1 void open(const char *filename, ios::openmode mode);
  void close();
```

Apertura de archivos, ejemplo

Crear y escribir sobre un archivo.

```
#include <fstream>
using namespace std;
int main(){
  ofstream OutFile;
  OutFile.open("archivo.txt");
  OutFile<< "Hola Mundo"<< endl;
  OutFile.close();
return 0;
}
```

Apertura de archivos, ejemplo

Leer un archivo.

```
1  #include <fstream>
   #include <iostream>
3  using namespace std;
   int main(){
5    ifstream InFile;
     char Data[128];
7    InFile.open("archivo.txt");
     InFile >> Data;
9    InFile.close();
     cout<< Data<<endl;
11 return 0;
   }
```

```
Hola
```

Apertura de archivos, ejemplo

Lectura y escritura de archivo.

```
#include <fstream>
#include <iostream>
using namespace std;
int main(){
  ifstream InFile;
  char Data[128];
  InFile.open("archivo.txt");
  InFile >> Data;
  InFile.close();
  cout<< Data<<endl;
return 0;
}
```

Consola:

```
Hola
```

Contenido de archivo despues de imprimir:

```
HolaHolado
```

Modos de apertura

ios es un **namespace**. app, trunc, ate función como modificadores con el operador OR (ejemplo abajo).

```
ios::app   Anexa datos
ios::ate   Abre el archivo y mueve el control
           al final del archivo (append to end).
ios::in    Abre para lectura
ios::out   Abre para escritura
ios::trunc Si el archivo existe remueve el
           contenido.
```

Modos de apertura

En este ejemplo se muestra: como abrir el archivo desde el constructor, como usar el operador OR para abrirlo en varios modos, como verificar si se abrió correctamente, y como verificar si una operación se realizó.

```
#include <fstream>
#include <iostream>
using namespace std;
int main(){
  fstream File("archivo.txt",ios::trunc | ios::out);
  char Data[128]="Aqui viendo que hace esto ";
  if(!File.eof()){
    cout<<"Pude abrir el archivo!"<< File << endl;
  File << Data<< endl;
  File>>Data ;
  if(File.fail())
    cout << "No pude escribir!"<<endl;
  File.close();
  }
return 0;
}
```

Tarea

Continuación de 12.2 y 12.3.
Probar estás tareas leyendo y escribiendo desde y hacia archivo.