Tp 0

Generado por Doxygen 1.8.6

Miércoles, 13 de Abril de 2016 23:30:38

Índice general

Capítulo 1

Índice de clases

1.1. Lista de clases

Lista de las clases, estructuras, uniones e interfaces con una breve descripción:

Capítulo 2

Documentación de las clases

2.1. Referencia de la Clase complejo

Clase representativa de un número complejo.

`#include <complejo.h>`

Métodos públicos

- complejo ()

  Constructor sin parámetros.
- complejo (double)

  Constructor.
- complejo (double, double)

  Constructor.
- complejo (const complejo &)

  Constructor por copia.
- complejo const & operator= (complejo const &)

  Sobrecarga operador asignación.
- ∼complejo ()

  Destructor.
- double re () const

  Parte real de un complejo.
- double im () const

  Parte imaginaria de un complejo.
- double abs () const

  Módulo de un número complejo.
- double phase () const

  Fase de un número complejo.

Métodos públicos estáticos

- static complejo fromPolarToRectangular (double, double)

  Conversor de complejo de forma polar a forma cartesiana.

Amigas

- complejo const operator+ (complejo const &, complejo const &)
  Sobrecarga operador suma.
- complejo const operator- (complejo const &, complejo const &)
  Sobrecarga operador resta.
- complejo const operator∗ (complejo const &, complejo const &)
  Sobrecarga operador multiplicación.
- complejo const operator/ (complejo const &, double)
  Sobrecarga operador división.
- complejo const operator^ (complejo const &, int)
  Sobrecarga operador potenciación.
- bool operator== (complejo const &, double)
  Sobrecarga operador igual.
- bool operator== (complejo const &, complejo const &)
  Sobrecarga operador igual.
- std::ostream & operator<< (std::ostream &, const complejo &)
  Sobrecarga operador escritura.
- std::istream & operator>> (std::istream &, complejo &)
  Sobrecarga operador lectura.

2.1.1. Descripción detallada

Clase representativa de un número complejo.

Posee algunas de las operaciones más comunes que se pueden llegar a necesitar para operar con numeros complejos.

2.1.2. Documentación del constructor y destructor

2.1.2.1. complejo::complejo ( )

Constructor sin parámetros.

Este constructor permite inicializar un complejo en (0,0).

```
8 : re_(0), im_(0) {}
```

2.1.2.2. complejo::complejo (double *r*)

Constructor.

Este constructor permite crear un complejo a partir de un número real. La parte imaginaria queda inicializada en cero.

```
10 : re_(r), im_(0){}
```

2.1.2.3. complejo::complejo (double *r*, double *i*)

Constructor.

Este constructor permite crear un complejo a partir de dos números reales. El primero corresponde a la parte real y el segundo a la parte imaginaria.

```
12 : re_(r), im_(i){}
```

2.1.2.4. complejo::complejo (const complejo & *c*)

Constructor por copia.

Construye un complejo a partir de la copiar de otro.

```
14 : re_(c.re_), im_(c.im_){}
```

2.1.2.5. complejo::~complejo ( )

Destructor.

Destructor para un numero complejo.

```
22 {}
```

2.1.3. Documentación de las funciones miembro

2.1.3.1. double complejo::abs ( ) const

Módulo de un número complejo.

Este método devuelve el modulo de un complejo.

```
32                          {
33     return std::sqrt(re_ * re_ + im_ * im_);
34 }
```

2.1.3.2. complejo complejo::fromPolarToRectangular (double *mod*, double *phase*) [static]

Conversor de complejo de forma polar a forma cartesiana.

Este método recibe por parámetro el modulo y la fase de un complejo y retorna un número complejo en su forma cartesiana.

```
42                                                          {
43     double re = mod*cos(phase);
44     double im = mod*sin(phase);
45     return complejo(re,im);
46 }
```

2.1.3.3. double complejo::im ( ) const

Parte imaginaria de un complejo.

Este método devuelve la parte imaginaria de un número complejo.

```
28                          {
29     return im_;
30 }
```

2.1.3.4. complejo const & complejo::operator= (complejo const & *c*)

Sobrecarga operador asignación.

Asigna parte real a parte real y parte imaginaría a parte imaginaría.

```
16                                                  {
17     re_ = c.re_;
18     im_ = c.im_;
19     return *this;
20 }
```

2.1.3.5. double complejo::phase ( ) const

Fase de un número complejo.

Este método devuelve la fase de un número complejo.

```
37                              {
38     return atan(this->im_/this->re_);
39 }
```

2.1.3.6. double complejo::re ( ) const

Parte real de un complejo.

Este método devuelve la parte real de un número complejo

```
24                            {
25     return re_;
26 }
```

2.1.4. Documentación de las funciones relacionadas y clases amigas

2.1.4.1. complejo const operator∗ (complejo const & *x*, complejo const & *y*) [friend]

Sobrecarga operador multiplicación.

Este método multiplica dos números complejos.

```
58                                                                  {
59     complejo r(x.re_ * y.re_ - x.im_ * y.im_,
60                x.re_ * y.im_ + x.im_ * y.re_);
61     return r;
62 }
```

2.1.4.2. complejo const operator+ (complejo const & *x*, complejo const & *y*) [friend]

Sobrecarga operador suma.

Este método suma dos números complejos.

```
48                                                                  {
49     complejo z(x.re_ + y.re_, x.im_ + y.im_);
50     return z;
51 }
```

2.1.4.3. complejo const operator- (complejo const & *x*, complejo const & *y*) [friend]

Sobrecarga operador resta.

Este método resta dos números complejos.

```
53                                                                  {
54     complejo r(x.re_ - y.re_, x.im_ - y.im_);
55     return r;
56 }
```

2.1.4.4. complejo const operator/ (complejo const & *c*, double *f*) [friend]

Sobrecarga operador división.

Este método divide un complejo con un número real.

```
66 {
67     return complejo(c.re_ / f, c.im_ / f);
68 }
```

2.1.4.5. std::ostream& operator<< (std::ostream & , const complejo &) [friend]

Sobrecarga operador escritura.

Este operador escribe un complejo en formato (Re,Img) al flujo de salida.

2.1.4.6. bool operator== (complejo const & *c*, double *f*) [friend]

Sobrecarga operador igual.

Este método compara la parte real de un un complejo con un numero real. Además verifica que la parte imaginaria del complejo sea cero.

```
95                                                  {
96     bool b = (c.im_ != 0 || c.re_ != f) ? false : true;
97     return b;
98 }
```

2.1.4.7. bool operator== (complejo const & *x*, complejo const & *y*) [friend]

Sobrecarga operador igual.

Este método compara dos números complejos. Parte real con parte real y parte imaginaria con parte imaginaria.

```
100                                                         {
101     bool b = (x.re_ != y.re_ || x.im_ != y.im_) ? false : true;
102     return b;
103 }
```

2.1.4.8. std::istream& operator>> (std::istream & , complejo &) [friend]

Sobrecarga operador lectura.

Este operador lee del flujo de entrada un número complejo. Acepta números reales individuales a los cuales se los toma como la parte real del numero complejo que retorna. En el caso de que reciba al numero complejo como par ordenado devuelve un complejo con la parte real y la parte imaginaria obtenida del flujo de entrada.

2.1.4.9. complejo const operator^ (complejo const & *c*, int *power*) [friend]

Sobrecarga operador potenciación.

Este método potencia un número complejo con un numero entero.

```
70                                                      {
71
72     if(power == 0){
73         return complejo(1,0);
74     }
75
76     if(power == 1){
```

```
77          return c;
78      }
79
80      double module = c.abs();
81      double phase = c.phase();
82      if(power < 0){
83          module = 1/module;
84      }
85
86      for(int i = 0; i < power - 1 ; i++){
87          module = module*module;
88      }
89      phase = phase*power;
90
91      return complejo::fromPolarToRectangular(module,phase);
92  }
```

La documentación para esta clase fue generada a partir de los siguientes ficheros:

- complejo.h
- complejo.cc

2.2. Referencia de la Clase DFTcalculator

Clase DFTcalculator.

`#include <DFTcalculator.h>`

Métodos públicos estáticos

- static void **calculateDFT** (const vector< complejo > &data, vector< complejo > &result)
- static void **calculateIDFT** (const vector< complejo > &data, vector< complejo > &result)
- static void calculateDFT (const vector< complejo > &data, vector< complejo > &result)

  Método el cual permite calcular la transformada discreta de fourier.
- static void calculateIDFT (const vector< complejo > &data, vector< complejo > &result)

  Método el cual permite calcular la anti-transformada discreta de fourier.

2.2.1. Descripción detallada

Clase DFTcalculator.

Esta clase contiene una serie de metodos que permiten calcular la transformada discreta de fourier y la transformada inversa de fourier.

2.2.2. Documentación de las funciones miembro

2.2.2.1. static void DFTcalculator::calculateDFT (const vector< complejo > & *data,* vector< complejo > & *result*) [inline], [static]

Método el cual permite calcular la transformada discreta de fourier.

Recibe por parámetro dos vectores uno con la información y otro donde escribirá el resultado.

```
66      {
67          calculate(data , result , "dft");
68      }
```

2.2.2.2. static void DFTcalculator::calculateIDFT (const vector< complejo > & *data,* vector< complejo > & *result*) [inline], [static]

Método el cual permite calcular la anti-transformada discreta de fourier.

Recibe dos parámetros uno con la información y otro donde la escribirá.

```
75     {
76         calculate(data , result , "idft");
77     }
```

La documentación para esta clase fue generada a partir de los siguientes ficheros:

- DFTcalculator.cc
- DFTcalculator.h

2.3. Referencia de la plantilla de la Clase vector< T >

Clase vector.

`#include <vector.h>`

Métodos públicos

- vector ()

  Constructor sin parámetros.
- vector (int size_)

  Constructor.
- vector (const vector< T > &cv)

  Constructor copia.
- ~vector ()

  Destructor de vector.
- int length () const

  Largo del vector.
- void pushBack (T &elem)

  Inserta un elemento al vector.
- vector< T > & operator= (const vector< T > &rigth)

  Operador asignación.
- bool operator== (const vector< T > &rigth) const

  Operador comparación.
- const T & operator[] (int index) const

  Operador indexación constante.
- T & operator[] (int index)

  Operador indexación.

Amigas

- std::istream & operator>> (std::istream &is, vector< T > &vector)

  Operador lectura.
- std::ostream & operator<< (std::ostream &os, const vector< T > &vector)

  Operador escritura.

2.3.1. Descripción detallada

template<class T>class vector< T >

Clase vector.

Permite almacenar un arreglo de datos. Contiene una serie de de métodos que permiten agregar, quitar y leer los distintos elementos almacenados en el vector. Además la clase esta templetizada, permitiendo así crear vectores de cualquier tipo.

2.3.2. Documentación del constructor y destructor

2.3.2.1. template<class T> vector< T >::vector ( ) [inline]

Constructor sin parámetros.

Inicializa un vector en cero. Además no crea la memoria para el vector.

```
41                 {
42             this->pv = NULL;
43             this->size = 0;
44             this->capacity = 0;
45         }
```

2.3.2.2. template<class T> vector< T >::vector (int *size_*) [inline]

Constructor.

Inicializa un vector vacío. A diferencia del constructor sin parámetros, este crea la memoria para el vector de acuerdo al parámetro size_.

```
52                         {
53             this->pv = new T[size_];
54             this->size = size_;
55             this->capacity = size_;
56         }
```

2.3.2.3. template<class T> vector< T >::vector (const vector< T > & *cv*) [inline]

Constructor copia.

Inicializa un vector a partir de otro pasado por parámetro.

```
62                                 {
63             this->size = cv.size ;
64             this->capacity = cv.capacity;
65             T* cp = new T[ size ];
66             for ( int i = 0; i < this->size; i++ ){
67                 cp[ i ] = cv.pv[ i ];
68             }
69             if(this->pv){
70                 delete[] this->pv;
71             }
72             this->pv = cp;
73         }
```

2.3.2.4. template<class T> vector< T >::~vector ( ) [inline]

Destructor de vector.

borra la memoria creada para el vector.

```
78                  {
79              if(this->pv){
80                  delete [] this->pv;
81              }
82          }
```

2.3.3. Documentación de las funciones miembro

2.3.3.1. template<class T> int vector< T >::length ( ) const [inline]

Largo del vector.

Este método devuelve el largo del vector.

```
87                          {
88              return this->size;
89          }
```

2.3.3.2. template<class T> vector<T>& vector< T >::operator= (const vector< T > & *rigth*) [inline]

Operador asignación.

Este operador copia los elementos del vector pasado por parámetro al objeto sobre el cual se ejecuto.

```
121                                                 {
122              if (&rigth != this)
123              {
124                  if (this->size != rigth.size) {
125                      T * aux;
126                      aux = new T[ rigth.size ];
127                      delete [] this->pv;
128                      this->size = rigth.size;
129                      this->pv = aux;
130                      for (int i = 0; i < size; i++){
131                          this->pv[i] = rigth.pv[i];
132                      }
133                      return *this;
134                  }
135                  else
136                  {
137                      for (int i = 0; i < this->size; i++){
138                          this->pv[i] = rigth.pv[i];
139                      }
140                      return *this;
141                  }
142              }
143              return *this;
144          }
```

2.3.3.3. template<class T> bool vector< T >::operator== (const vector< T > & *rigth*) const [inline]

Operador comparación.

Compara el contenido del vector pasado por parámetro con el vector sobre el cual se ejecuto el operador. En el caso que todos los elementos coincidan devuelve true.

```
152                                                 {
153              if (this->size != rigth.size)
154                  return false; // Vectores de diferentes tamaños
155              else
156                  for (int i = 0; i < this->size; i++)
157                      if (this->pv[ i ] != rigth.pv[ i ])
158                          return false;
159              return true;
160          }
```

2.3.3.4. template<class T> const T& vector< T >::operator[] (int *index*) const [inline]

Operador indexación constante.

Permite obtener el objeto almacenado en una determinada posición del vector.

```
166                                                 {
167             if(index >= this->size){
168                 cerr « "Indice incorrecto en const operator[]" « endl;
169                 abort();
170             }
171             return this->pv[index];
172         }
```

2.3.3.5. template<class T> T& vector< T >::operator[] (int *index*) [inline]

Operador indexación.

Permite asignarle un valor a una determinada posición del vector.

```
178                                       {
179             if(index >= this->size){
180                 cout « "Indice incorrecto en operator[]" « endl;
181                 abort();
182             }
183             return this->pv[index];
184         }
```

2.3.3.6. template<class T> void vector< T >::pushBack (T & *elem*) [inline]

Inserta un elemento al vector.

Este método inserta al final del vector un elemento. Para agregar el elemento primero verifica que tenga memoria, en caso de no tener crea memoria con capacidad igual al doble de la que tenía.

```
97                                      {
98              if(this->size == 0){
99                  this->pv = new T[2];
100                 this->capacity = 2;
101             }
102             else{
103                 if(this->capacity == this->size){
104                     T* aux = this->pv;
105                     this->pv = new T[this->capacity*2];
106                     this->capacity = this->capacity*2;
107                     for(int i = 0 ; i < this->size ; i++){
108                         this->pv[i] = aux[i];
109                     }
110                     delete[] aux;
111                 }
112             }
113             this->pv[this->size] = elem;
114             this->size++;
115         }
```

2.3.4. Documentación de las funciones relacionadas y clases amigas

2.3.4.1. template<class T> std::ostream& operator<< (std::ostream & *os,* const vector< T > & *vector*) [friend]

Operador escritura.

Permite escribir en un stream de salida el vector del cual se llamo.

```
202                                                                                 {
203             for(int i = 0 ; i < vector.size ; i++){
204                 os « vector[i] « endl;
205             }
206             return os;
207         }
```

2.3.4.2. template<class T> std::istream& operator>> (std::istream & *is,* vector< T > & *vector*) [friend]

Operador lectura.

Permite leer de un stream de entrada un conjunto de objetos del tipo T, almacenándolos en un vector.

```
190                                                                     {
191             T aux;
192             for(int i = 0 ; is » aux ; i++){
193                 vector.pushBack(aux);
194             }
195             return is;
196         }
```

La documentación para esta clase fue generada a partir del siguiente fichero:

- vector.h

Índice alfabético