

Algoritmos y Programación II

Trabajo Práctico 0

Carlos Germán Carreño Romano, Sebastián Sampayo, Rodrigo Bourbon

April 23, 2015

Contents

1	Introducción	1
1.1	Radio definida por software (SDR)	1
1.2	Transmisión de TV por cable	2
1.3	Aplicación del Trabajo Práctico	2
2	Desarrollo	3
2.1	Estándar de estilo	4
3	Compilación	4
3.1	Opciones del programa	5
4	Ejecución	5
4.1	Casos de prueba	6
4.2	Caso 1	6
4.3	Caso 2	6
4.4	Caso 3	6
4.5	Caso 4	7
4.6	Caso 5	7
5	Códigos	7
5.1	Estructura de archivos	7
5.2	Clase Complex	12
5.3	Clase cmdline	20
6	Enunciado	30

1 Introducción

1.1 Radio definida por software (SDR)

El concepto de Radio Definida por Software se le atribuye a Joseph Mitola, 1990. Se refiere a un dispositivo que permite reducir al mínimo el hardware necesario para la recepción de señales de radio. Dicho equipo captura la señal analógica (ya sea mediante un cable o una antena), la digitaliza (mediante un conversor A/D) para luego realizar por software toda la etapa de procesamiento de



Figure 1: Sintonizador de radio digital.

señal requerido en la decodificación. Esto ha logrado que la recepción de cierto rango de telecomunicaciones sea mucho más accesible en términos económicos y prácticos (ya que el mismo dispositivo físico se puede utilizar para distintos fines con solo re-programar el software). Un ejemplo de este dispositivo se puede ver a continuación:

1.2 Transmisión de TV por cable

En telecomunicaciones, la televisión analógica se transmite mediante el método de la Multiplexión por División en Frecuencia (FDM). Esta técnica consiste en transmitir varias señales simultáneamente modulando cada una con una portadora diferente, en el rango de VHF/UHF, de forma tal que los anchos de banda de cada señal no se superpongan significativamente. El canal destinado para la transmisión de una emisora tiene un ancho de banda de aproximadamente 6 Mhz, donde los 5.45 Mhz más bajos corresponden al espectro de la señal de video y los últimos 0.55Mhz (aproximadamente) se reservan para el espectro de la señal de audio. Este modelo de comunicación se puede ver representado en la figura 2:

1.3 Aplicación del Trabajo Práctico

Sabiendo que el audio de la televisión está modulado en frecuencia (FM), si se toma la porción del canal adecuada es posible demodular dicha señal y escuchar algún canal de televisión.

En este caso particular, el SDR se utilizó para capturar un ancho de banda de 2.4Mhz y centrado en 181.238 Mhz. A través del aplicativo desarrollado se pudo escuchar efectivamente el programa emitido.

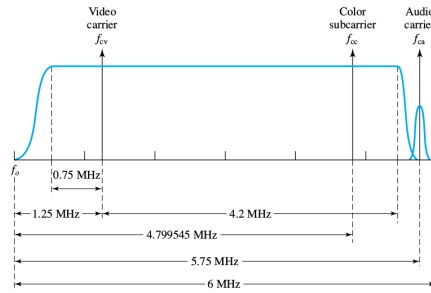


Figure 2: Señal de TV transmitida.

2 Desarrollo

Para implementar el proceso propuesto en la especificación del presente trabajo, se descubrió que la cascada de los bloques "Promediador móvil de N elementos" seguido de "Decimador de N elementos", es totalmente equivalente a un único bloque en el cual se tomen N elementos y cuya salida sea el promedio de dichos N elementos. A continuación se toman los siguientes N elementos y la nueva salida es el promedio de estos últimos N elementos, y así sucesivamente.

Consecuentemente, se optó por implementar este último algoritmo a fin de simplificar el código del trabajo. En concreto se utilizaron sentencias básicas de control de flujo (while; if; continue). Esto se puede ver en el siguiente extracto del main:

```
(...)
//Mientras haya complejos en la entrada
while (*iss >> input_complex)
{
    // ( -- Promediar 11 elementos -- )
    buffer1 += input_complex;
    if (i < DECIMATOR1_SIZE-1)
    {
        i++;
        continue;
    }
    x = buffer1/DECIMATOR1_SIZE;
    buffer1 = 0;
    i = 0;
}
(...)
```

La idea es ir acumulando en un búffer la suma de los complejos a la entrada (`input_complex`), hasta que hayan pasado 11 elementos (parametrizado por `DECIMATOR1_SIZE`). En dicho momento la condición del if se cumple y entonces se divide al valor del búffer por 11 para obtener la salida del bloque. Luego se vacía el buffer y se reinicializa la cuenta.

2.1 Estándar de estilo

Adoptamos la convención de code styling de Google para C++, salvando las siguientes excepciones:

- streams: utilizamos flujos de entrada y salida
- sobrecarga de operadores
- Para notación de archivos, variables, clases, tipos de datos y variables globales nos seguimos de la regla detallada en:

<https://google-styleguide.googlecode.com/svn/trunk/cppguide.html#Naming>

3 Compilación

Para compilar los códigos fuentes utilizamos el compilador g++, de la Free Software Foundation ¹, con la opción -Wall, que activa cualquier tipo de advertencia además de los errores de compilación.

El proceso de compilación se realiza con el comando

`make`

que ejecuta el archivo Makefile, donde se detalla el árbol de archivos del programa, la compilación de cada código objeto por separado, y algunos comentarios. El archivo Makefile se detalla a continuación:

```
CC      = g++
CCFLAGS = -g -Wall -pedantic
LDFLAGS =
OBJS = main.o complex.o cmdline.o
PROGRAM_NAME = tp0

all: tp0
# @/bin/true

tp0: $(OBJS)
$(CC) $(LDFLAGS) $(OBJS) -o $(PROGRAM_NAME)

main.o: main.cc complex.h cmdline.h
$(CC) $(CCFLAGS) -c main.cc

complex.o: complex.cc complex.h
$(CC) $(CCFLAGS) -c complex.cc

cmdline.o: cmdline.cc cmdline.h
$(CC) $(CCFLAGS) -c cmdline.cc
```

¹www.fsf.org

```
clean:
rm *.o
```

3.1 Opciones del programa

Los argumentos posibles para pasarle a la línea de comandos del programa son los siguientes (el orden en que aparezcan no tiene importancia):

- Ingreso de datos: es un argumento obligatorio y se indica en su forma corta mediante "-i" y en su forma larga mediante "--input". Se le debe pasar siempre una opción que debe ser la ruta de un archivo del cual queramos leer o bien la opción por defecto "-" que utiliza el flujo de entrada estándar.
- salida de datos: es un argumento obligatorio y se indica en su forma corta mediante "-o" y en su forma larga mediante "--output". Se le debe pasar siempre una opción que debe ser la ruta de un archivo en el cual queramos imprimir o bien la opción por defecto "-" que utiliza el flujo de salida estándar.
- Formato de ingreso/salida de datos: es un argumento opcional, siendo la opción por defecto la de modo texto y se indica en su forma corta mediante "-f" y en su forma larga mediante "--format". Las opciones que recibe este argumento son:
 - Formato texto: "txt".
 - Formato U8: "U8".

Ambas opciones de formato analizan la entrada y producen una salida de acuerdo a lo establecido en el enunciado del trabajo. Para caracterizarlas, en el programa se definió un tipo enumerativo con las opciones y un atributo privado de este tipo en la clase "Complex" para al momento de crear un objeto de la clase pasarle como argumento el formato. Este atributo es útil al momento de ejecutar la lectura de la entrada. Esto se realizó sobrescribiendo los flujos de entrada y salida ("»" y "«" respectivamente) y declarando un método privado particular para cada formato (se definieron como privados porque son relativos a la implementación). Los métodos de los flujos llaman al método correspondiente mediante una selección con la sentencia "switch", que evalúa el atributo de formato del complejo en el que se está leyendo. La salida resultante es un valor que también depende del formato elegido, por lo tanto se codificó una función para cada caso y se selecciona la que corresponde mediante un puntero de arreglos a función. Esta estructura del programa nos proporciona escalabilidad y mantenibilidad, ya que se codificaron módulos dedicados para cada operación y si se agrega una nueva opción solo se debe codificar otro módulo, o si falla alguno la reparación es específica de ese módulo. Para la compilación

4 Ejecución

En los siguientes casos de prueba requeridos en el enunciado, se ejecutó el programa desarrollado en una consola linux. Se muestra a continuación las imágenes

con las corridas del programa para los distintos casos.

4.1 Casos de prueba

4.2 Caso 1

```
developer@DeveloperVM: ~/Code
developer@DeveloperVM:~/Code$ cat 1.in
developer@DeveloperVM:~/Code$ ./tp0 -i 1.in -o 1.out
developer@DeveloperVM:~/Code$ cat 1.out
developer@DeveloperVM:~/Code$ ls -l 1.*
-rwxrwxrwx 1 root root 0 abr 23 12:22 1.in
-rwxrwxrwx 1 root root 0 abr 23 15:00 1.out
developer@DeveloperVM:~/Code$
```

4.3 Caso 2

```
developer@DeveloperVM: ~/Code
developer@DeveloperVM:~/Code$ perl -e 'print "(0,0)\n" x44;' > 2.in
developer@DeveloperVM:~/Code$ ./tp0 -i 2.in -o -
0
developer@DeveloperVM:~/Code$ ls 2.*
2.in 2.out
developer@DeveloperVM:~/Code$
```

4.4 Caso 3

```
developer@DeveloperVM: ~/Code
developer@DeveloperVM:~/Code$ cat test3.pl
$pi = abs(atan2(0, -1));
for (1 .. 4) {
  for (1 .. 4) {
    $phi += $pi / 10;
    $re = cos($phi);
    $im = sin($phi);
    $sign *= -1.0;
    print "($re, $im)\n" x 11;
  }
}
developer@DeveloperVM:~/Code$ perl ./test3.pl | ./tp0 -i - -o -
0.075
0.1
0.1
0.1
developer@DeveloperVM:~/Code$
```

4.5 Caso 4

```
developer@DeveloperVM: ~/Code
developer@DeveloperVM:~/Code$ cat test4.pl
$pi = abs(atan2(0, -1));
$sign = 1.0;

while (1) {
    $phi += $pi / 10 * $sign;
    $re = cos($phi);
    $im = sin($phi);
    $sign *= -1.0;

    print "($re, $im)\n" x 44;
}
developer@DeveloperVM:~/Code$ perl ./test4.pl | ./tp0 -i - -o - | head
0
-0.025
0.025
-0.025
0.025
-0.025
0.025
-0.025
0.025
-0.025
0.025
developer@DeveloperVM:~/Code$
```

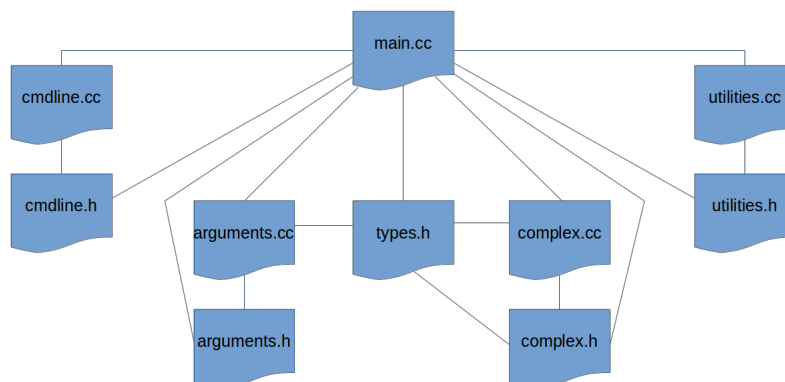
4.6 Caso 5

```
developer@DeveloperVM: ~/Code
developer@DeveloperVM:~/Code$ ./tp0 -f U8 -i sample-2-2.4Mss -o - | aplay -
-f U8 -r 54000
Playing raw data 'stdin' : Unsigned 8 bit, Rate 54000 Hz, Mono
```

5 Códigos

A continuación se adjuntan los códigos fuente de todos los archivos utilizados en este proyecto. Para tener una referencia, el árbol de archivos descrito en el makefile tiene una referencia gráfica en la siguiente sección.

5.1 Estructura de archivos



main

```
1  //
2  //
3  // Facultad de Ingenieria de la Universidad de Buenos
4  // Aires
5  // Algoritmos y Programacion II
6  // 1er Cuatrimestre de 2015
7  // Trabajo Practico 0: Programacion en C++
8  // Demodulacion de senal FM
9  //
10 //
11 //
12 #include <iostream>
13 #include <fstream>
14 #include <sstream>
15 #include <cstdlib>
16 #include <cstring>
17
18 #include "main.h"
19 #include "complex.h"
20 #include "cmdline.h"
21 #include "arguments.h"
22 #include "utilities.h"
23 #include "types.h"
24
25 using namespace std;
26
27
28 // Coleccion de funciones para imprimir en formatos
29 // distintos
30 void (*print_phase[]) (double) = {
31     print_phase_text ,
32     print_phase_U8
33 };
34
35
36 // Opciones de argumentos de invocacion
37 static option_t options[] = {
38     {1, "i", "input", "-", opt_input, OPT_SEEN},
39     {1, "o", "output", "-", opt_output, OPT_SEEN},
40     {1, "f", "format", "txt", opt_format, OPT_SEEN},
41     {0, },
42 };
```



```

43
44 extern istream *iss;
45 extern format_option_t format_option;
46
47 int
48 main(int argc, char * const argv[])
49 {
50     size_t i = 0;
51     size_t j = 0;
52     Complex buffer1 = 0;
53     double buffer2 = 0;
54     Complex x, x_prev;
55     Complex aux;
56     double output_phase;
57
58     // Parsear argumentos de invocacion
59     cmdline cmdl(options);
60     cmdl.parse(argc, argv);
61
62     // Inicializar el complejo con el fomato especificado
63     Complex input_complex(format_option);
64     // cout<<int(input_complex.format_option())<<endl;
65
66     // ( — Condiciones Iniciales Nulas — )
67     x_prev = 0;
68
69     // Mientras haya complejos en la entrada
70     while (*iss >> input_complex)
71     {
72         // ( — Promediar 11 elementos — )
73         buffer1 += input_complex;
74         if (i < DECIMATOR1.SIZE-1)
75         {
76             i++;
77             continue;
78         }
79         x = buffer1/DECIMATOR1.SIZE;
80         buffer1 = 0;
81         i = 0;
82
83         // ( — Obtener la derivada de la fase — )
84         aux = x * x_prev.conjugated();
85         output_phase = aux.phase();
86
87         // ( — Avanzar una muestra — )
88         x_prev = x;
89
90         // ( — Promediar 4 elementos — )
91         buffer2 += output_phase;
92         if (j < DECIMATOR2.SIZE-1)

```

```

93     {
94         j++;
95         continue;
96     }
97     output_phase = buffer2/DECIMATOR2.SIZE;
98     buffer2 = 0;
99     j = 0;
100
101     // ( — Imprimir en el formato especificado — )
102     (print_phase[format_option])(output_phase);
103 }
104 return 0;
105 } // main

```

main.h

```

1 //


---


    //
2 // Facultad de Ingenieria de la Universidad de Buenos
   Aires
3 // Algoritmos y Programacion II
4 // 1er Cuatrimestre de 2015
5 // Trabajo Practico 0: Programacion en C++
6 // Demodulacion de senal FM
7 //
8 // main.h
9 // Parametros principales de la aplicacion
10 // - Tamano de los decimadores
11 //


---


    //
12
13 #ifndef MAIN_H_INCLUDED
14 #define MAIN_H_INCLUDED
15
16 #define DECIMATOR1.SIZE 11
17 #define DECIMATOR2.SIZE 4
18
19 #endif

```

utilities.h

```

1 //


---


    //
2 // Facultad de Ingenieria de la Universidad de Buenos
   Aires
3 // Algoritmos y Programacion II
4 // 1er Cuatrimestre de 2015
5 // Trabajo Practico 0: Programacion en C++

```

```

6 // Demodulacion de senal FM
7 //
8 // utilities.h
9 // Funciones utilitarias para el proposito de la
  aplicacion
10 // - Imprimir en formato texto
11 // - Imprimir en formato binario U8
12 //

```

```

13 //
14
15 #ifndef _UTILITIES_H_INCLUDED_
16 #define _UTILITIES_H_INCLUDED_
17
18 #include <iostream>
19
20 #define PI 3.14159265359
21
22 // Imprime en el rango [-1, 1] en formato texto (ASCII)
23 void print_phase_text(double);
24 // Imprime en el rango [0, 255] en formato U8 (binario)
25 void print_phase_U8(double);
26
27 #endif

```

```

1 //
2 //
3 // Facultad de Ingenieria de la Universidad de Buenos
  Aires
4 // Algoritmos y Programacion II
5 // 1er Cuatrimestre de 2015
6 // Trabajo Practico 0: Programacion en C++
7 // Demodulacion de senal FM
8 //
9 // utilities.cc
10 // Funciones utilitarias para el proposito de la
  aplicacion
11 // - Imprimir en formato texto
12 // - Imprimir en formato binario U8
13 //

```

```

14 //
15
16 #include <iostream>
17 #include <fstream>
18 #include <sstream>
19
20 #include "utilities.h"

```

```

19
20 using namespace std;
21
22
23
24 extern ostream *oss;
25
26 // Imprime en el rango [-1, 1] en formato texto (ASCII)
27 void
28 print_phase_text(double output_phase)
29 {
30
31     *oss << output_phase/PI << endl;
32
33     return;
34 }
35
36
37 // Imprime en el rango [0, 255] en formato U8 (binario)
38 void
39 print_phase_U8(double output_phase)
40 {
41
42     *oss << (unsigned char)((output_phase + PI) * 255 / (2
43         * PI));
44
45     return;
46 }

```

5.2 Clase Complex

complex.h

```

1 #ifndef _COMPLEX_H_INCLUDED_
2 #define _COMPLEX_H_INCLUDED_
3
4 #include <iostream>
5
6 #include "types.h"
7
8 class Complex
9 {
10 public:
11     Complex();
12     Complex(double);
13     Complex(format_option_t);
14     Complex(double, double);
15     Complex(double r, double i, format_option_t f_o);
16     Complex(const Complex &);
17     Complex const &operator=(Complex const &);

```

```

18 Complex const &operator*=(Complex const &);
19 Complex const &operator+=(Complex const &);
20 Complex const &operator-=(Complex const &);
21 ~Complex();
22
23 double real() const;
24 double imag() const;
25 format_option_t format_option() const;
26 double abs() const;
27 double abs2() const;
28 double phase() const;
29 Complex const &conjugate();
30 Complex const conjugated() const;
31 bool iszero() const;
32
33 friend Complex const operator+(Complex const &,
34                               Complex const &);
35 friend Complex const operator-(Complex const &,
36                               Complex const &);
37 friend Complex const operator*(Complex const &,
38                               Complex const &);
39 friend Complex const operator/(Complex const &,
40                               Complex const &);
41 friend Complex const operator/(Complex const &,
42                               double);
43
44 friend bool operator==(Complex const &, double);
45 friend bool operator==(Complex const &, Complex const
46                        &);
47
48 friend std::ostream &operator<<(std::ostream &, const
49                               Complex &);
50 friend std::istream &operator>>(std::istream &,
51                               Complex &);
52
53 private:
54     double real_, imag_;
55     format_option_t format_option_;
56
57 // Lee en formato texto "(Real, Imaginario)"
58 friend std::istream &read_format_text(std::istream &,
59                                       Complex &);
60 // Lee en formato binario Real8bits seguido de
61   Imaginario8bits
62 friend std::istream &read_format_U8(std::istream &,
63                                     Complex &);
64 // Escribe en formato texto "(Real, Imaginario)"
65 friend std::ostream &write_format_text(std::ostream &,
66                                         const Complex &);

```

```

55 // Escribe en fomato binario Real8bits seguido de
    Imaginario8bits
56 friend std::ostream &write_format_U8(std::ostream &,
    const Complex &);
57
58 }; // class Complex
59
60 #endif // _COMPLEX_H_INCLUDED_

complex.cc

1 #include <iostream>
2 #include <cmath>
3
4 #include "complex.h"
5 #include "types.h"
6
7 using namespace std;
8
9
10
11 Complex::Complex() : real_(0), imag_(0), format_option_(
    FORMAT_OPTION_TEXT)
12 {
13 }
14
15 Complex::Complex(format_option_t f_o) : real_(0), imag_(
    0), format_option_(f_o)
16 {
17 }
18
19 Complex::Complex(double r) : real_(r), imag_(0),
    format_option_(FORMAT_OPTION_TEXT)
20 {
21 }
22
23 Complex::Complex(double r, double i) : real_(r), imag_(i)
    , format_option_(FORMAT_OPTION_TEXT)
24 {
25 }
26
27 Complex::Complex(double r, double i, format_option_t f_o)
    : real_(r), imag_(i), format_option_(f_o)
28 {
29 }
30
31 Complex::Complex(Complex const &c) : real_(c.real_),
    imag_(c.imag_), format_option_(c.format_option_)
32 {
33 }

```

```

34
35 Complex const &
36 Complex::operator=(Complex const &c)
37 {
38     real_ = c.real_;
39     imag_ = c.imag_;
40     format_option_ = c.format_option_;
41     return *this;
42 }
43
44 Complex const &
45 Complex::operator*=(Complex const &c)
46 {
47     double re = real_ * c.real_
48               - imag_ * c.imag_;
49     double im = real_ * c.imag_
50               + imag_ * c.real_;
51     real_ = re, imag_ = im;
52     return *this;
53 }
54
55 Complex const &
56 Complex::operator+=(Complex const &c)
57 {
58     double re = real_ + c.real_;
59     double im = imag_ + c.imag_;
60     real_ = re, imag_ = im;
61     return *this;
62 }
63
64 Complex const &
65 Complex::operator-=(Complex const &c)
66 {
67     double re = real_ - c.real_;
68     double im = imag_ - c.imag_;
69     real_ = re, imag_ = im;
70     return *this;
71 }
72
73 Complex::~~Complex()
74 {
75 }
76
77 double
78 Complex::real() const
79 {
80     return real_;
81 }
82
83 double Complex::imag() const

```

```

84 {
85     return imag_;
86 }
87
88 format_option_t Complex::format_option() const
89 {
90     return format_option_;
91 }
92
93 }
94
95 double
96 Complex::abs() const
97 {
98     return std::sqrt(real_ * real_ + imag_ * imag_);
99 }
100
101 double
102 Complex::abs2() const
103 {
104     return real_ * real_ + imag_ * imag_;
105 }
106
107 double
108 Complex::phase() const
109 {
110     return atan2(imag_, real_);
111 }
112
113 Complex const &
114 Complex::conjugate()
115 {
116     imag_ *= -1;
117     return *this;
118 }
119
120 Complex const
121 Complex::conjugated() const
122 {
123     return Complex(real_, -imag_);
124 }
125
126 bool
127 Complex::iszero() const
128 {
129     #define ZERO(x) ((x) == +0.0 && (x) == -0.0)
130     return ZERO(real_) && ZERO(imag_) ? true : false;
131 }
132
133 Complex const

```



```

134 operator+(Complex const &x, Complex const &y)
135 {
136     Complex z(x.real_ + y.real_, x.imag_ + y.imag_);
137     return z;
138 }
139
140 Complex const
141 operator-(Complex const &x, Complex const &y)
142 {
143     Complex r(x.real_ - y.real_, x.imag_ - y.imag_);
144     return r;
145 }
146
147 Complex const
148 operator*(Complex const &x, Complex const &y)
149 {
150     Complex r(x.real_ * y.real_ - x.imag_ * y.imag_,
151               x.real_ * y.imag_ + x.imag_ * y.real_);
152     return r;
153 }
154
155 Complex const
156 operator/(Complex const &x, Complex const &y)
157 {
158     return x * y.conjugated() / y.abs2();
159 }
160
161 Complex const
162 operator/(Complex const &c, double f)
163 {
164     return Complex(c.real_ / f, c.imag_ / f);
165 }
166
167 bool
168 operator==(Complex const &c, double f)
169 {
170     bool b = (c.imag_ != 0 || c.real_ != f) ? false : true;
171     return b;
172 }
173
174 bool
175 operator==(Complex const &x, Complex const &y)
176 {
177     bool b = (x.real_ != y.real_ || x.imag_ != y.imag_) ?
178               false : true;
179     return b;
180 }
181
182 istream &
183 operator>>(istream &is, Complex &c)

```

```

183 {
184     switch (c.format_option_) {
185
186         case FORMAT_OPTION_TEXT:
187             return read_format_text(is, c);
188
189         case FORMAT_OPTION_U8:
190             return read_format_U8(is, c);
191
192     }
193
194     return is;
195 }
196
197 ostream &
198 operator<<(ostream &os, const Complex &c)
199 {
200     switch (c.format_option_) {
201
202         case FORMAT_OPTION_TEXT:
203             return write_format_text(os, c);
204
205         case FORMAT_OPTION_U8:
206             return write_format_U8(os, c);
207
208     }
209
210     return os;
211 }
212
213 // Lee en formato texto "(Real, Imaginario)"
214 istream &
215 read_format_text(istream &is, Complex &c)
216 {
217
218     int good = false;
219     int bad  = false;
220     double re = 0;
221     double im = 0;
222     char ch = 0;
223
224     if (is >> ch
225         && ch == '(') {
226         if (is >> re
227             && is >> ch
228             && ch == ', '
229             && is >> im
230             && is >> ch
231             && ch == ')')
232             good = true;

```

```

233         else
234             bad = true;
235     } else if (is.good()) {
236         is.putback(ch);
237         if (is >> re)
238             good = true;
239         else
240             bad = true;
241     }
242
243     if (good)
244         c.real_ = re, c.imag_ = im;
245     if (bad)
246         is.clear(ios::badbit);
247
248     return is;
249 }
250
251 // Lee en formato binario Real8bits seguido de
252 // Imaginario8bits
253 istream &
254 read_format_U8(istream &is, Complex &c)
255 {
256     int good = false;
257     int bad = false;
258     unsigned char re = 0;
259     unsigned char im = 0;
260
261     if (is >> re && is >> im)
262         good = true;
263     else
264         bad = true;
265
266     if (good)
267         c.real_ = re - 128;
268         c.imag_ = im - 128;
269     if (bad)
270         is.clear(ios::badbit);
271
272     return is;
273 }
274
275 // Escribe en formato texto "(Real, Imaginario)"
276 ostream &
277 write_format_text(ostream &os, const Complex &c)
278 {
279
280
281

```

```

282     return os << "("
283           << c.real()
284           << ", "
285           << c.imag()
286           << ")";
287
288 }
289
290 // Escribe en fomato binario Real8bits seguido de
291     Imaginario8bits
292 ostream &
293 write_format_U8(ostream &os, const Complex &c)
294 {
295     return os << (char)c.real()
296           << (char)c.imag();
297
298 }

```

5.3 Clase cmdline

cmdline.h

```

1  #ifndef _CMDLINE_H_INCLUDED_
2  #define _CMDLINE_H_INCLUDED_
3
4  #include <string>
5  #include <iostream>
6
7  #define OPT_DEFAULT    0
8  #define OPT_SEEN      1
9  #define OPT_MANDATORY 2
10
11 struct option_t {
12     int has_arg;
13     const char *short_name;
14     const char *long_name;
15     const char *def_value;
16     void (*parse)(std::string const &); // Puntero a
17         funcion de opciones
18     int flags;
19 };
20
21 class cmdline
22 {
23 public:
24     cmdline(option_t *);
25     void parse(int, char * const []);
26
27 private:
28     // Este atributo apunta a la tabla que describe todas

```

```

28 // las opciones a procesar. Por el momento, solo
    puede
29 // ser modificado mediante constructor, y debe
    finalizar
30 // con un elemento nulo.
31 //
32 option_t *option_table;
33
34 // El constructor por defecto cmdline::cmdline(), es
35 // privado, para evitar construir "parsers" (
    analizador
36 // sintactico, recibe una palabra y lo interpreta en
37 // una accion dependiendo su significado para el
    programa)
38 // sin opciones. Es decir, objetos de esta clase sin
    opciones.
39 //
40
41 cmdline();
42 int do_long_opt(const char *, const char *);
43 int do_short_opt(const char *, const char *);
44 }; // class cmdline
45
46 #endif

```

cmdline.cc

```

1 // cmdline - procesamiento de opciones en la linea de
    comando.
2 //
3 // $Date: 2012/09/14 13:08:33 $
4 //
5 #include <string>
6 #include <cstdlib>
7 #include <iostream>
8
9 #include "cmdline.h"
10
11 using namespace std;
12
13
14
15 cmdline::cmdline()
16 {
17 }
18
19 cmdline::cmdline(option_t *table) : option_table(table)
20 {
21     /*
22     - Lo mismo que hacer:

```

```

23     option_table = table;
24
25     Siendo "option_table" un atributo de la clase cmdline
26     y table un puntero a objeto o struct de "option_t".
27
28     Se estaria contruyendo una instancia de la clase
29     cmdline
30     cargandole los datos que se hayan en table (la table
31     con
32     las opciones, ver el codigo en main.cc)
33     */
34 }
35
36 void
37 cmdline::parse(int argc, char * const argv[])
38 {
39     #define END_OF_OPTIONS(p) \
40         ((p)->short_name == 0 \
41          && (p)->long_name == 0 \
42          && (p)->parse == 0)
43
44     // Primer pasada por la secuencia de opciones: marcamos
45     // todas las opciones, como no procesadas. Ver codigo
46     // de
47     // abajo.
48     for (option_t *op = option_table; !END_OF_OPTIONS(op);
49          ++op)
50         op->flags &= ~OPT_SEEN;
51
52     // Recorremos el arreglo argv. En cada paso, vemos
53     // si se trata de una opcion corta, o larga. Luego,
54     // llamamos a la funcion de parseo correspondiente.
55     //
56     for (int i = 1; i < argc; ++i) {
57         // Todos los parametros de este programa deben
58         // pasarse en forma de opciones. Encontrar un
59         // parametro no-opcion es un error.
60         //
61         if (argv[i][0] != '-') {
62             cerr << "Invalid non-option argument: "
63                  << argv[i]
64                  << endl;
65             exit(1);
66         }
67
68         // Usamos "--" para marcar el fin de las
69         // opciones; todo los argumentos que puedan

```

```

69     // estar a continuacion no son interpretados
70     // como opciones.
71     //
72     if (argv[i][1] == '-'
73         && argv[i][2] == 0)
74         break;
75
76     // Finalmente, vemos si se trata o no de una
77     // opcion larga; y llamamos al metodo que se
78     // encarga de cada caso.
79     //
80     if (argv[i][1] == '-')
81         i += do_long_opt(&argv[i][2], argv[i + 1]);
82     else
83         i += do_short_opt(&argv[i][1], argv[i + 1]);
84 }
85
86 // Segunda pasada: procesamos aquellas opciones que,
87 // (1) no hayan figurado explicitamente en la linea
88 // de comandos, y (2) tengan valor por defecto.
89 //
90 for (option_t *op = option_table; !END_OF_OPTIONS(op);
91      ++op) {
92 #define OPTION_NAME(op) \
93     (op->short_name ? op->short_name : op->long_name)
94     if (op->flags & OPT_SEEN)
95         continue;
96     if (op->flags & OPT_MANDATORY) {
97         cerr << "Option "
98             << "-"
99             << OPTION_NAME(op)
100             << " is mandatory."
101             << "\n";
102         exit(1);
103     }
104     if (op->def_value == 0)
105         continue;
106     op->parse(string(op->def_value));
107 }
108
109 int
110 cmdline::do_long_opt(const char *opt, const char *arg)
111 {
112     // Recorremos la tabla de opciones, y buscamos la
113     // entrada larga que se corresponda con la opcion de
114     // linea de comandos. De no encontrarse, indicamos el
115     // error.
116     //

```

```

117     for (option_t *op = option_table; op->long_name != 0;
118         ++op) {
119         if (string(opt) == string(op->long_name)) {
120             // Marcamos esta opcion como usada en
121             // forma explicita, para evitar tener
122             // que inicializarla con el valor por
123             // defecto.
124             op->flags |= OPT_SEEN;
125
126             if (op->has_arg) {
127                 // Como se trata de una opcion
128                 // con argumento, verificamos que
129                 // el mismo haya sido provisto.
130                 //
131                 if (arg == 0) {
132                     cerr << "Option requires argument: "
133                         << "___"
134                         << opt
135                         << "\n";
136                     exit(1);
137                 }
138                 op->parse(string(arg));
139                 return 1;
140             } else {
141                 // Opcion sin argumento.
142                 //
143                 op->parse(string(""));
144                 return 0;
145             }
146         }
147     }
148
149     // Error: opcion no reconocida. Imprimimos un mensaje
150     // de error, y finalizamos la ejecucion del programa.
151     //
152     cerr << "Unknown option: "
153         << "___"
154         << opt
155         << "."
156         << endl;
157     exit(1);
158
159     // Algunos compiladores se quejan con funciones que
160     // logicamente no pueden terminar, y que no devuelven
161     // un valor en esta ultima parte.
162     //
163     return -1;
164 }
165

```



```

166 int
167 cmdline::do_short_opt(const char *opt, const char *arg)
168 {
169     option_t *op;
170
171     // Recorremos la tabla de opciones, y buscamos la
172     // entrada corta que se corresponda con la opcion de
173     // linea de comandos. De no encontrarse, indicamos el
174     // error.
175     //
176     for (op = option_table; op->short_name != 0; ++op) {
177         if (string(opt) == string(op->short_name)) {
178             // Marcamos esta opcion como usada en
179             // forma explicita, para evitar tener
180             // que inicializarla con el valor por
181             // defecto.
182             //
183             op->flags |= OPT_SEEN;
184
185             if (op->has_arg) {
186                 // Como se trata de una opcion
187                 // con argumento, verificamos que
188                 // el mismo haya sido provisto.
189                 //
190                 if (arg == 0) {
191                     cerr << "Option requires argument: "
192                         << "_"
193                         << opt
194                         << "\n";
195                     exit(1);
196                 }
197                 op->parse(string(arg));
198                 return 1;
199             } else {
200                 // Opcion sin argumento.
201                 //
202                 op->parse(string(""));
203                 return 0;
204             }
205         }
206     }
207
208     // Error: opcion no reconocida. Imprimimos un mensaje
209     // de error, y finalizamos la ejecucion del programa.
210     //
211     cerr << "Unknown option: "
212         << "_"
213         << opt
214         << ". "
215         << endl;

```

```

216     exit(1);
217
218     // Algunos compiladores se quejan con funciones que
219     // logicamente no pueden terminar, y que no devuelven
220     // un valor en esta ultima parte.
221     //
222     return -1;
223 }

```

arguments.h

```

1  //


---


    //
2  // Facultad de Ingenieria de la Universidad de Buenos
    Aires
3  // Algoritmos y Programacion II
4  // 1er Cuatrimestre de 2015
5  // Trabajo Practico 0: Programacion en C++
6  // Demodulacion de senal FM
7  //
8  // arguments.h
9  // Funciones a llamar para cada opcion posible de la
    aplicacion
10 // Nombres de los argumentos de la opcion "--format"
11 //


---


    //
12
13 #ifndef _ARGUMENTS_HINCLUDED_
14 #define _ARGUMENTS_HINCLUDED_
15
16 #include <iostream>
17
18 #define FORMAT_OPTIONS 2
19 #define FORMAT_TEXT "txt"
20 #define FORMAT_U8 "U8"
21
22 void opt_input(std::string const &);
23 void opt_output(std::string const &);
24 void opt_format(std::string const &);
25
26 #endif

```

arguments.cc

```

1  //


---


    //
2  // Facultad de Ingenieria de la Universidad de Buenos
    Aires

```

```

3 // Algoritmos y Programacion II
4 // 1er Cuatrimestre de 2015
5 // Trabajo Practico 0: Programacion en C++
6 // Demodulacion de senal FM
7 //
8 // arguments.cc
9 // Funciones a llamar para cada opcion posible de la
   aplicacion
10 //

```

```

11 //
12 #include <iostream>
13 #include <fstream>
14 #include <sstream>
15 #include <cstdlib>
16 #include <cstring>
17
18 #include "arguments.h"
19 #include "types.h"
20
21 using namespace std;
22
23
24
25 istream *iss;
26 ostream *oss;
27 fstream ifs;
28 fstream ofs;
29 format_option_t format_option;
30
31 // Nombres de los argumentos de la opcion "--format"
32 string description_format_option[] = {
33
34     FORMAT_TEXT,
35     FORMAT_U8
36
37 };
38
39 void
40 opt_input(string const &arg)
41 {
42     // Si el nombre del archivos es "-", usaremos la
       entrada
43     // estandar. De lo contrario, abrimos un archivo en
       modo
44     // de lectura.
45     //
46     if (arg == "-") {

```

```

47     iss = &cin;    // Establezco la entrada estandar cin
                        como flujo de entrada
48 }
49 else {
50     ifs.open(arg.c_str(), ios::in); // c_str(): Returns a
                        pointer to an array that contains a null-
                        terminated
51                                // sequence of characters (i.e., a C-
                        string) representing
52                                // the current value of the string
                        object.

53     iss = &ifs;
54 }
55
56 // Verificamos que el stream este OK.
57 //
58 if (!iss->good()) {
59     cerr << "Cannot open "
60           << arg
61           << ". "
62           << endl;
63     exit(1);
64 }
65 }
66
67 void
68 opt_output(string const &arg)
69 {
70     // Si el nombre del archivos es "-", usaremos la salida
71     // estandar. De lo contrario, abrimos un archivo en
72     // de escritura.
73     //
74     if (arg == "-") {
75         oss = &cout; // Establezco la salida estandar cout
                        como flujo de salida
76     } else {
77         ofs.open(arg.c_str(), ios::out);
78         oss = &ofs;
79     }
80
81     // Verificamos que el stream este OK.
82     //
83     if (!oss->good()) {
84         cerr << "Cannot open "
85               << arg
86               << ". "
87               << endl;
88         exit(1); // EXIT: Terminacion del programa en su
                        totalidad

```

```

89     }
90 }
91
92 void
93 opt_format(string const &arg)
94 {
95     size_t i;
96     // Recorremos diccionario de argumentos hasta encontrar
97     // uno que coincida
98     for(i=0; i < FORMAT_OPTIONS; i++) {
99         if(arg == description_format_option[i]) {
100             format_option = (format_option_t)i; // Casteo
101             break;
102         }
103     }
104     // Si recorrio todo el diccionario, el argumento no
105     // esta implementado
106     if (i == FORMAT_OPTIONS) {
107         cerr << "Unknown format" << endl;
108         exit(1);
109     }
110 }

```

6 Enunciado