Tema 4

Clases y objetos en C++

4.1. Introducción

A lo largo del curso nos hemos encontrado con varias situaciones en las que era necesario trabajar con datos para los que no existía un tipo predefinido adecuado. Por ejemplo, programas que debían procesar números racionales, naipes en un juego de cartas, fichas de clientes, listas de nombres, etc. La solución que hemos adoptado hasta el momento es definir un nuevo tipo, normalmente una estructura (`struct`), y definir funciones y procedimientos que procesaran tales estructuras. Esta solución presenta varios inconvenientes que ilustraremos a continuación.

Supongamos que vamos a escribir un programa C++ en el que necesitamos procesar fechas. C++ carece de un tipo predefinido adecuado para representar las fechas, de manera que decidimos definir un tipo `TFecha` de la siguiente manera:

```
struct TFecha {
    int dia;    // 1..31
    int mes;    // 1..12
    int anyo;   // 2000...2999
};
```

Además de definir la estructura `TFecha`, debemos definir funciones y procedimientos que soporten operaciones básicas sobre este tipo. Por ejemplo, podemos incluir en nuestro programa las siguientes declaraciones:

```
// operaciones basicas para el tipo TFecha
bool laborable(TFecha f);
bool festivo(TFecha f);
void manyana(TFecha& f);
```

```
void ayer(TFecha& f);
int dias_entre(TFecha f1, TFecha f2);
```

Sería deseable que una vez que hemos definido el tipo `TFecha` y sus operaciones básicas, este nuevo tipo se pudiera emplear como si fuera un tipo predefinido de C++. Por desgracia, esto no es así. La solución adoptada presenta una serie de inconvenientes.

En primer lugar, no hay forma de prohibir a otros programadores el acceso a los componentes de la estructura que implementa el tipo `TFecha`. Cualquier programador puede acceder de forma directa a cualquier campo y modificar su valor. Esto puede hacer los programas más difíciles de depurar, pues es posible que estos accesos directos a la estructura no preserven la consistencia de los datos. Por ejemplo, un programador puede escribir una función como la siguiente:

```
void pasado_manyana(TFecha& f)
{
    f.dia= f.dia+2;
}
```

Es fácil ver que la función `pasado_manyana` puede dar lugar a fechas inconsistentes como el 30 de febrero de 2002. El programador ha olvidado que "pasado mañana" puede ser "el mes que viene" o incluso "el año que viene". Si todos los accesos directos a los campos de `TFecha` los realiza el programador que definió el tipo y nos encontramos con una fecha inconsistente, el error debe estar necesariamente localizado en alguna de las operaciones básicas del tipo.

Otro problema que se deriva de permitir el acceso directo a la estructura es que los programas se vuelven más difíciles de modificar. Supongamos que decidimos alterar la estructura interna del tipo `TFecha` modificando el tipo del campo `mes`, añadiendo un tipo enumerado para los meses:

```
enum TMes {enero, febrero,..., noviembre, diciembre};

struct TFecha {
    int  dia;    // 1..31
    TMes mes;    // enero...diciembre
    int  anyo;   // 2000...2999
};
```

Si otro programador había escrito una función como la siguiente:

```
void mes_que_viene(TFecha& f)
{
    f.mes= (f.mes % 12 ) + 1;
}
```

ésta dejará de compilar. Si todos los accesos directos a `TFecha` se han realizado en las operaciones básicas, sólo éstas necesitan ser modificadas.

Finalmente, otro inconveniente de definir un nuevo tipo mediante una estructura y una serie de operaciones básicas es la falta de cohesión. No hay forma de ver el tipo `TFecha` como un todo, como un conjunto de valores y una serie de operaciones básicas asociadas. En concreto, no hay forma de establecer explícitamente la relación entre el tipo `TFecha` y sus operaciones básicas. Suponemos que la función `festivo` es una operación básica del tipo `TFecha` simplemente porque tiene un argumento de este tipo. Pero, ¿cómo sabemos si `pasado_manyana` es o no una operación básica? Y si definimos una función que toma argumentos de diferentes tipos... ¿a cuál de esos tipos pertenece la función?, ¿de cuál de ellos es una operación básica?

El propósito de las clases en C++ es facilitar al programador una herramienta que le permita definir un nuevo tipo que se pueda usar como un tipo predefinido de C++. En particular, las clases de C++ facilitan un mecanismo que permite prohibir los accesos directos a la representación interna de un tipo, así como indicar claramente cuáles son las operaciones básicas definidas para el tipo.

4.2. Revisión de conceptos básicos

4.2.1. Interfaz vs. Implementación

Al definir una clase deben separarse claramente por una parte los detalles del funcionamiento interno de la clase, y por otra la forma en que se usa la clase. Esto lo hemos hecho en pseudo-código distinguiendo entre el interfaz y la implementación de la clase:

```
INTERFAZ CLASE NombreClase
       METODOS
             ...
FIN NombreClase

IMPLEMENTACION CLASE NombreClase
       ATRIBUTOS
```

```
            ...
        METODOS
            ...
FIN NombreClase
```

El interfaz puede entenderse como las instrucciones de uso de la clase, mientras que la implementación contiene (y oculta) los detalles de funcionamiento.

4.2.2. Implementador vs. Usuario

Es muy importante recordar que un programador puede desempeñar dos papeles diferentes respecto a una clase: implementador y usuario. El programador *implementador* de una clase se encarga de definir su interfaz (cabecera de los métodos) y de desarrollar los detalles internos de su implementación (atributos y cuerpo de los métodos). El implementador de una clase tiene acceso total a los objetos de esa clase.

Por otro lado, el programador *usuario* sólo puede utilizar los objetos de una clase aplicándoles los métodos definidos en su interfaz. El usuario no tiene acceso directo a los detalles internos de la implementación.

En las siguientes secciones, veremos cómo definir e implementar clases en C++ (punto de vista del implementador) y cómo usar una clase C++ (punto de vista del usuario).

4.3. Definición de clases en C++

Desgraciadamente, la división entre interfaz e implementación no es tan limpia en C++ como en el pseudo-código. Las clases se definen en C++ mediante una construcción `class` dividida en dos partes: una parte privada (`private`) que contiene *algunos* detalles de la implementación, y una parte pública (`public`) que contiene todo el interfaz.

```
class NombreClase {
    private:
            // implementacion de la clase
            // solamente los atributos
    public:
            // interfaz de la clase
};
```

En la parte privada de la construcción `class` aparecen sólo los atributos de la clase y algunos tipos intermedios que puedan ser necesarios. En C++, la implementación de los

métodos de la clase se facilita aparte. En la parte pública, suelen aparecer solamente las declaraciones (cabeceras) de los métodos de la clase. Por ejemplo, la siguiente es una definición de la clase `CComplejo` que representa números complejos:

```
class CComplejo {
    private:
        // atributos
        double real, imag;
        // los metodos se implementan aparte
    public:
        void asigna_real(double r);
        void asigna_imag(double i);
        double parte_real();
        double parte_imag();
        void suma(const CComplejo& a, const CComplejo& b);
};
```

Los campos `real` e `imag` son los atributos de la clase y codifican el *estado* de un objeto de la clase `CComplejo`. Puesto que los atributos están declarados en la parte privada de la clase, forman parte de la implementación y no es posible acceder a ellos desde fuera de la clase. Su acceso está restringido: sólo se puede acceder a ellos en la implementación de los métodos de la clase.

Los métodos que aparecen en la parte pública forman el interfaz de la clase y describen su *comportamiento*; es decir, las operaciones que podemos aplicar a un objeto del tipo `CComplejo`. En particular, con estos métodos podemos asignar valores a las partes real e imaginaria, leer las partes real e imaginaria, y sumar dos numeros complejos.

4.4. Implementación de métodos en C++

Como comentamos anteriormente, la implementación de los métodos de una clase en C++ se realiza fuera de la construcción `class {...}`. La sintaxis de la definición de un método es similar a la de la definición de una función (o procedimiento), excepto que el nombre del método debe estar precedido por el nombre de la clase de la que forma parte:

```
void CComplejo::asigna_real(double r)
{
  // cuerpo del metodo...
}
```

Como puede apreciarse, el método `asignar_real` no recibe ningún argumento de tipo `CComplejo`. ¿Cómo es posible entonces que este método sepa qué número complejo tiene que modificar? La respuesta es que todos los métodos de la clase `CComplejo` reciben como argumento de entrada/salida implícito el complejo al que se va a aplicar el método. Surge entonces la siguiente pregunta: si este argumento es implícito y no le hemos dado ningún nombre, ¿cómo accedemos a sus atributos? La respuesta en este caso es que podemos referirnos a los atributos de este parámetro implícito simplemente escribiendo los nombres de los atributos, sin referirnos a qué objeto pertenecen. C++ sobreentiende que nos referimos a los atributos del argumento implícito. Así, el método `asigna_real` se implementa como sigue:

```
void CComplejo::asigna_real(double r)
{
   real= r;
}
```

donde el atributo `real` que aparece a la izquierda de la asignación es el atributo del argumento implícito. Incluso un método como `parte_imaginaria`, que aparentemente no tiene argumentos, recibe este argumento implícito que representa el objeto al que se aplica el método:

```
double CComplejo::parte_imaginaria()
{
   return imag;  // atributo imag del argumento implicito
}
```

Por otro lado, un método puede recibir argumentos explícitos de la clase a la que pertenece. Por ejemplo, el método `suma` recibe dos argumentos explícitos de tipo `CComplejo`. Al definir el método `suma`, podemos acceder a los atributos de los argumentos explícitos utilizando la notación punto usual, como si se tratara de una estructura:

```
void CComplejo::suma(const CComplejo& a, const CComplejo& b)
{
    real= a.real + b.real;
    imag= a.imag + b.imag;
}
```

Obsérvese que desde el cuerpo del método `suma` podemos realizar accesos directos a los atributos de `a` y `b`. Esto es posible porque este método forma parte de la implementación

de la clase y, por lo tanto, conoce cómo está implementada. Cuando escribimos el código de un método, adoptamos el papel de implementadores y por lo tanto tenemos acceso a todos los detalles de implementación de la clase.

4.5. Uso de clases en C++

Una vez que se ha definido e implementado una clase, es posible declarar objetos de esta clase y usarlos desde fuera de la clase como si se tratara de tipos predefinidos[1]. En concreto, una variable de un tipo clase se declara como cualquier otra variable:

```
CComplejo a, b, s;
CComplejo v[10];   // array de 10 objetos CComplejo
```

Sobre un objeto de una clase sólo pueden aplicarse las siguientes operaciones:

1. aquéllas definidas en el interfaz de la clase
2. la asignación
3. el paso de parámetros, por valor o por referencia. A lo largo del curso **siempre pasaremos los objetos por referencia**. Si no queremos que el parámetro sea modificado, le añadiremos `const` a la definición del parámetro, tal y como aparece en la definición del método `suma`

Es muy importante recordar que como usuario de una clase es imposible acceder de forma directa a los atributos privados de un objeto de tal clase. Por ejemplo, no podemos inicializar el complejo `a` de la siguiente forma:

```
a.real= 2;   // ERROR: acceso no permitido al usuario
a.imag= 5;   // ERROR: acceso no permitido al usuario
```

puesto que los atributos `real` e `imag` son privados. Si queremos inicializar el complejo `a`, debemos hacerlo aplicando los métodos `asigna_real` y `asigna_imag`.

La aplicación de métodos a un objeto se realiza mediante paso de mensajes. La sintaxis de un mensaje es similar a la de la llamada a una función, excepto que el nombre del método va precedido por el nombre del objeto al que se aplica el método. Por ejemplo, para asignar `7` a la parte real del complejo `a`, basta aplicar el método `asigna_real` al objeto `a` componiendo el mensaje:

[1]En realidad, para poder emplear una clase C++ como un auténtico tipo predefinido es necesario tener en cuenta unos detalles que están más allá del objetivo de este curso

```
a.asigna_real(7);
```

Aparte de la asignación y del paso de parámetros **por referencia**, toda la manipulación de los objetos por parte del usuario debe hacerse a través de paso de mensajes a los mismos. El siguiente código muestra un ejemplo de uso de la clase `CComplejo`:

```
CComplejo a, b, s;

a.asigna_real(1);
a.asigna_imag(3);
b.asigna_real(2);
b.asigna_imag(7);
s.suma(a,b);
cout << s.parte_real() << ", " << s.parte_imag() << "i" << endl;
```

4.6. Constructores y destructores

Como todas las variables de C++, los objetos no están inicializados por defecto[2]. Si el usuario declara objetos `CComplejo` y opera con ellos sin asignarles un valor inicial, el resultado de la operación no estará definido:

```
CComplejo a,b,s;  // no inicializados

a.suma(a,b);
cout << s.parte_real() << ", "  // imprime basura
     << s.parte_imag() << "i"
     << endl;
```

Los valores mostrados en pantalla por el ejemplo anterior son del todo imprevisibles. Una solución es inicializar los complejos explícitamente, como hicimos en el ejemplo de la sección anterior mediante los métodos `asigna_real` y `asigna_imag`:

```
CComplejo a,b,s;

a.asigna_real(1.0);
a.asigna_imag(2.0);
```

[2]En realidad, las variables y objetos *globales* están inicializados por defecto a "cero"

```
b.asigna_real(-1.5);
b.asigna_imag(3.5);

s.suma(a,b);
cout << s.parte_real() << ", "
     << s.parte_imag() << "i"
     << endl;
```

Esta inicialización explícita, además de ser engorrosa, conlleva sus propios problemas. Por un lado, es posible que el programador olvide invocar todos los métodos necesarios para inicializar cada objeto que aparece en su programa; o bien que inicialice algún objeto más de una vez. Por otro lado, no siempre tendremos un valor adecuado para inicializar los objetos, de forma que su inicialización explícita resulta un tanto forzada.

Para paliar este problema, C++ permite definir un método especial llamado el *constructor* de la clase, cuyo cometido es precisamente inicializar por defecto los objetos de la clase. Para que nuestros números complejos estén inicializados, basta añadir un constructor a la clase `CComplejo` de la siguiente manera:

```
class CComplejo {
    private:
        double real, imag;

    public:
        CComplejo();   // constructor
        void asigna_real(double r);
        void asigna_imag(double i);
        double parte_real();
        double parte_imag();
        void suma(const CComplejo& a, const CComplejo& b);
};
```

La implementación del constructor se hace fuera de la construcción `class`. En nuestro ejemplo, podríamos optar por inicializar los complejos a $1 + 0i$:

```
CComplejo::CComplejo()
{
   real= 1;
   imag= 0;
}
```

Un constructor es un método muy diferente de todos los demás. En primer lugar, su nombre coincide siempre con el de la clase a la que pertenece (en nuestro caso, `CComplejo`). Además, un constructor no es ni un procedimiento ni una función, y por lo tanto no tiene asociado ningún tipo de retorno (ni siquiera `void`). Por último, el usuario nunca invoca un constructor de manera explícita. Esto no tendría sentido, pues de lo que se trata es de que los objetos sean inicializados de manera implícita por C++, sin intervención alguna por parte del usuario. Por ello, el constructor de una clase es invocado automáticamente justo después de cada declaración un objeto de esa clase. Siguiendo con nuestro ejemplo, en el código:

```
CComplejo a,b,s;  // inicializados a 1+0i por el constructor

a.suma(a,b);
cout << s.parte_real() << ", "  // imprime 2, 0i
     << s.parte_imag() << "i"
     << endl;
```

El constructor `CComplejo::CComplejo()` se invoca automáticamente 3 veces, para inicializar los objetos `a`, `b` y `s`, respectivamente.

El constructor que hemos descrito anteriormente es el constructor por *defecto*. Se llama así porque los objetos son todos inicializados a un valor por defecto. Además del constructor por defecto, es posible asociar a una clase un constructor *extendido* en que se indiquen mediante argumentos los valores a los que se debe inicializar un objeto de la clase. Podemos añadir un constructor extendido (con argumentos) a la clase `CComplejo` como sigue:

```
class CComplejo {
    private:
        double real, imag;

    public:
        CComplejo();   // constructor por defecto
        CComplejo(double r, double i); // constructor extendido
        void asigna_real(double r);
        void asigna_imag(double i);
        double parte_real();
        double parte_imag();
        void suma(const CComplejo& a, const CComplejo& b);
```

```
};
```

La implementación del constructor extendido es inmediata, basta emplear los argumentos para inicializar los atributos:

```
CComplejo::CComplejo(double r, double i)
{
   real= r;
   imag= i;
}
```

Para que C++ ejecute de forma automática el constructor extendido, basta añadir a la declaración de un objeto los valores a los que desea que se inicialice:

```
CComplejo a;       // inicializado por defecto
CComplejo b(1,5);  // inicializado a 1+5i
CComplejo c(2);    // ERROR: o  dos argumentos o ninguno...
```

De la misma manera que C++ permite definir constructores para los objetos de una clase, también es posible definir un *destructor* que se encargue de destruir los objetos automáticamente, liberando los recursos que pudieran tener asignados:

```
class CComplejo {
    private:
        double real, imag;

    public:
        CComplejo();
        CComplejo(double r, double i);
        ~CComplejo();                    // destructor
        void asigna_real(double r);
        void asigna_imag(double i);
        double parte_real();
        double parte_imag();
        void suma(const CComplejo& a, const CComplejo& b);
};
```

El nombre del destructor es siempre el de la clase a la que pertenece antecedido por el símbolo `~` (en nuestro ejemplo, `~CComplejo()`). Al igual que los constructores, los destructores se definen fuera de la construcción `class {...}` y no tienen tipo de retorno alguno:

```
CComplejo::~CComplejo()
{
   // destruir el numero complejo...
}
```

Es importante recordar que sólo puede definirse un destructor para cada clase, y que éste nunca toma argumentos. Además, el usuario nunca debe ejecutar un destructor de forma explícita. Los destructores son invocados automáticamente por C++ cada vez que un objeto deja de existir. Por ejemplo:

```
void ejemplo()
{
   CComplejo a;       // inicializado por defecto
   CComplejo b(1,5);  // inicializado a 1+5i

   //...

}
```

Al terminar de ejecutarse la función `ejemplo`, el destructor `CComplejo::~CComplejo()` es invocado automáticamente 2 veces para destruir los objetos locales `a` y `b`.

Los destructores se emplean típicamente en clases cuyos objetos tienen asociados recursos que se deben devolver al sistema. Durante este curso, los emplearemos sobre todo para liberar la memoria dinámica asignada a un objeto implementado con punteros.

4.7. Relación de uso o clientela

Los atributos de una clase C++ pueden ser de cualquier tipo. En particular, puede definirse una clase $\mathcal{A}$ que tenga como atributos objetos de alguna otra clase $\mathcal{B}$. Cuando esto ocurre, se dice que la clase $\mathcal{A}$ guarda una relación de uso o clientela con la clase $\mathcal{B}$. La clase $\mathcal{A}$ es la clase usuaria o *cliente* y la clase $\mathcal{B}$ es la clase usada o *proveedora*. Por ejemplo, la siguiente es una clase que define un vector de 3 números complejos:

```
class CVectorComp {
   private:
      CComplejo v[3];  // usa objetos de CComplejo
   public:
      CVectorComp();
```

```
            ~CVectorComp();
            void asignar(int i, const TComplejo& c);
            CComplejo acceder(int i);
            void suma(const CVectorComp& a, const CVectorComp& b);
};
```

En este caso, la clase `CVectorComp` es cliente de la clase `CComplejo`.

Por supuesto, una clase puede ser cliente de varias clases al mismo tiempo; es decir, tener atributos de varias clases. En general, la relación de uso entre clases no plantea ninguna dificultad. Sin embargo, hay unos pocos de detalles de los que conviene estar prevenido:

1. el *implementador* de la clase cliente es *usuario* de las clases proveedoras, y por lo tanto debe manipular los objetos de éstas a través del interfaz

2. antes de invocar al constructor de la clase cliente, los constructores de las clases proveedoras son invocados una vez por cada objeto usado por la clase cliente

3. después de invocar al destructor de la clase cliente, los destructores de las clases proveedoras son invocados una vez por cada objeto usado por la clase cliente

Aplicando lo anterior a nuestro ejemplo, el implementador de la clase `CVectorComp` es un *usuario* de la clase `CComplejo`, y por lo tanto debe manipular los números complejos a través su interfaz exclusivamente. Además, en el siguiente código:

```
void ejemplo_vector()
{
   CVectorComp a;

   //...
}
```

se generan las siguientes invocaciones automáticas de constructores y destructores:

1. se invoca 3 veces a `CComplejo()`, una por cada elemento del array `v`

2. se invoca 1 vez a `CVectorComp`, para construir `a`

3. se invoca 1 vez a `~CVectorComp()`, para destruir `a`

4. se invoca 3 veces a `~CComplejo()`, una por cada elemento del array `v`

La lección a recordar aquí es que C++ se ocupa automáticamente de la construcción y destrucción de objetos compuestos, sin que intervenga el programador. En general, no se debe llamar explícitamente ni a los constructores ni a los destructores.

4.8. Métodos constantes

Por defecto, los métodos de una clase tienen derecho a modificar los atributos del objeto receptor, ya que lo reciben como un parámetro implícito de entrada/salida. A veces nos interesaría indicar que un método no modifica ningún atributo del objeto receptor. Para ello, basta añadir la palabra clave `const` al final de la declaración del método (tanto en el interfaz como en la implementación.)

Por ejemplo, los métodos `parte_real` y `parte_imag` de la clase `CComplejo` no modifican el estado del objeto receptor. Esto puede indicarse explícitamente en el interfaz de la clase:

```
class CComplejo {
    private:
        double real, imag;

    public:
        CComplejo();
        CComplejo(double r, double i);
        ~CComplejo();
        void asigna_real(double r);
        void asigna_imag(double i);
        double parte_real() const;      // constante
        double parte_imag() const;      // constante
        void suma(const CComplejo& a, const CComplejo& b);
};
```

Y también debe indicarse en la implementación de los métodos:

```
double parte_real() const
{
   return real;
}
```

4.9. Punteros a objetos

Los punteros a objetos se declaran como cualquier otro tipo de puntero:

```
CComplejo* pc;
```

Puesto que `pc` es un puntero y no un objeto, su declaración no lleva implícita la ejecución del constructor, y por lo tanto no está inicializado: `pc` apunta a cualquier posición de memoria justo después de haber sido declarado.

Un puntero a objeto se manipula como cualquier otro tipo de puntero. En particular, para crear la variable dinámica apuntada por el puntero `pc`, debe emplearse el operador `new`; para enviar mensajes al objeto apuntado por `pc`, debe emplearse el operador `->`; y finalmente, para liberar el objeto apuntado por `pc` debe emplearse el operador `delete`:

```
CComplejo* pc;

pc= new CComplejo;
pc->asigna_real(3);
pc->asigna_imag(6);
//...
delete pc;
```

Aparentemente, el puntero `pc` se comporta como cualquier otro puntero. Sin embargo, los punteros a objetos guardan una sutil diferencia con los punteros a otros tipos. La variable dinámica `*pc` es un objeto que debe ser inicializado y destruido. Por lo tanto, al crear un objeto dinámico `*pc` a través de `new`, éste es inicializado automáticamente por el constructor `CComplejo()`. De la misma manera, al liberar un objeto dinámico `*pc` mediante `delete`, éste es destruido automáticamente por del destructor `~CComplejo()`. Así, en el código anterior se generan las siguientes invocaciones automáticas:

```
CComplejo* pc;

pc= new CComplejo;   // constructor por defecto sobre *pc
pc->asigna_real(3);
pc->asigna_imag(6);
//...
delete pc;          // destructor sobre *pc
```

En el momento de crear la variable dinámica (`new`), es posible indicar los argumentos del constructor extendido:

```
CComplejo* pc;

pc= new CComplejo(1,3);   // constructor extendido sobre *pc
pc->asigna_real(3);
pc->asigna_imag(6);
//...
delete pc;               // destructor sobre *pc
```

4.10. Ejercicios

1. Implementa la siguiente clase para representar complejos en forma binómica:

```
class CComplejo {

    private:
        double real, imag;

    public:
        CComplejo();
        CComplejo(double r, double i);
        void asigna_real(double r);
        void asigna_imag(double i);
        double parte_real() const;
        double parte_imag() const;
        double modulo() const;
        double argumento() const;
        void suma(const CComplejo& a, const CComplejo& b);
        void resta(const CComplejo& a, const CComplejo& b);
};
```

   y escribe una función `calculadora(a,b)` que reciba dos complejos `a` y `b` (por referencia) y calcule e imprima su suma y su resta.

2. Repite el ejercicio anterior pero representando esta vez los complejos en forma polar:

```
class CComplejo {
```

```
    private:
        double mdl,        // modulo
               argumento;  // argumento

    public:
        CComplejo();
        CComplejo(double r, double i);
        void asigna_real(double r);
        void asigna_imag(double i);
        double parte_real() const;
        double parte_imag() const;
        double modulo() const;
        double argumento() const;
        void suma(const CComplejo& a, const CComplejo& b);
        void resta(const CComplejo& a, const CComplejo& b);
};
```

Para realizar la conversión entre forma binómica y polar, puedes emplear las siguientes funciones de la biblioteca `math.h`:

```
double cos(double x);
double sin(double x);
double sqrt(double x);
double atan2(double y, double x);
```

La función `atan2` calcula el arcotangente de `y/x`. Observa que el interfaz de la clase `CComplejo` no ha cambiado, por lo que el programa de prueba y la función `calculadora(a,b)` deben seguir funcionando sin realizar modificación alguna. En concreto, el constructor `CComplejo(r,i)` sigue recibiendo las partes real e imaginaria del complejo.

3. Implementa la siguiente clase para representar un reloj digital:

```
class CReloj {

    private:
        int horas, minutos, segundos;
        int bateria;
```

```
            // cada tic-tac consume una unidad de energia

    public:
        CReloj(int h, int m, int s, int b);
        void tic_tac();
        void avanza(int h, int m, int s);
        void atrasa(int h, int m, int s);
        bool esta_parado();
        void recarga_bateria(int b);
        void escribe_en_12h();
        void escribe_en_24h();
        void sincronizar(CReloj& r);
            // ambos relojes se ajustan al promedio
};
```

y escribe un menú que permita manipular un reloj digital.

4. Implementa la siguiente clase para representar cuentas bancarias:

```
class CCuenta {

    private:
        double saldo;  // en euros
        double interes;

    public:
        CCuenta(double saldo_inicial, double inter);
        void ingresar(double i);
        void retirar(double r);
        double saldo();
        double intereses();
        bool numeros_rojos();
};
```

y escribe un menú que permita manipular una cuenta.

5. Implementa la siguiente clase para representar una serpiente multicolor:

```
class CSerpiente {

    private:

        enum TColores {rojo, azul, verde, amarillo};

        static const int MAX_SERPIENTE= 32;
        TColores cuerpo[MAX_SERPIENTE];
        int longitud;

    public:
        Serpiente();
        ~Serpiente();
        void crecer();
        void menguar();
        void mudar_piel();
        void pintar();
};
```

Una serpiente se representa mediante el array de `TColores cuerpo`, del que se emplearán sólo las posiciones desde `0` hasta `longitud-1`. Cada vez que la serpiente crece se añade una nueva anilla al final de su cuerpo de un color elegido al azar (utiliza la función `int rand()` de `stdlib.h`). Cada vez que la serpiente mengua, se elimina la última anilla de su cuerpo. Si la serpiente muda la piel, se cambian aleatoriamente todos sus colores, pero su longitud no se altera.

Escribe un programa que simule la vida de una serpiente y vaya mostrando por pantalla como cambia su aspecto. Cuando la serpiente muera, se debe mostrar un mensaje por pantalla que anuncie tan trágico suceso.