

Programación Orientada a Objetos

Práctica 0: Clases, objetos y excepciones
Implementación de clases de utilidad para la gestión de la librería

Curso 2023–2024

Índice

1. Clase Fecha	2
2. Clase Cadena	5
3. Organización	8
4. Makefile y pruebas a realizar	9
4.1. Makefile	9
4.2. Pruebas a realizar	10

1. Clase Fecha

Se trata de hacer una clase para trabajar con fechas, en español. Esta clase se llamará *Fecha* y sus atributos serán 3 enteros que representarán, por este orden, día, mes y año.

Una *Fecha* se podrá construir:

1. Con 3 parámetros, que serán, por este orden: el día, el mes y el año.

```
Fecha a(18, 7, 1936);    // 18 de julio de 1936
```

2. Con 2 parámetros; que serán, por este orden: el día y el mes, siendo el año el de la fecha del sistema.

```
Fecha b(17, 7);          // 17 de julio del año en curso
```

3. Con un parámetro, el día, tomándose el mes y el año de la fecha del sistema. Pero no se permitirá la conversión implícita de un entero a una *Fecha*.

```
Fecha c(1);              // Primer día del mes en curso del año en curso
Fecha d = 1;              // ERROR, conversión int -> Fecha no permitida
void fuu(const Fecha& f);
fuu(5);                   // ERROR, conversión int -> Fecha no permitida
fuu(Fecha(5));            // OK, conversión explícita de 5 a Fecha(5)
```

4. Sin parámetros, tomando los valores de la fecha del sistema.

```
Fecha hoy;               // La fecha del sistema, de hoy
```

5. A partir de otra *Fecha*.

```
Fecha d(c);              // 1 del mes y año en curso
Fecha e = b;             // Igual, e vale 17 de julio de este año
```

6. A partir de una cadena de caracteres de bajo nivel en el formato "*dd/mm/aaaa*", siendo *dd* el día expresado con 1 o 2 dígitos, *mm* el mes expresado con 1 o 2 dígitos y *aaaa* el año expresado con 4 dígitos (todos los dígitos en base 10). Se permiten conversiones de una cadena de caracteres en este formato a una *Fecha*. Si hay más caracteres no numéricos después del último número, simplemente se descartarán. Los ceros a la izquierda se aceptan y descartan: "00003/002/02020" es lo mismo que "3/2/2020".

```
Fecha f("11/9/2001");    // 11 de septiembre de 2001
Fecha g("1/12/1998");     // 1 de diciembre de 1998
Fecha h("1/01/2000");     // 1 de enero de 2000
Fecha i("11/12/1973");    // 11 de diciembre de 1973
Fecha j("22/03");         // MAL: falta el /año
Fecha k("22-03-1965");    // MAL: - en vez de /
Fecha j("02/2/2002)&");   // OK, 2 de febrero de 2002, se descartan )&
```

En todos los casos anteriores, un valor 0 para día, mes o año no será considerado incorrecto, sino que en ese caso se tomará el valor correspondiente de la fecha del sistema. Ejemplos:

```
Fecha l(3, 0, 2012); // 3 del mes en curso de 2012
Fecha m(0, 0, 0);    // Como 'Fecha m;': la fecha del sistema de hoy
Fecha n("00/0/000"); // Ídem, "hoy". Da igual el n.º de ceros
Fecha o("0/0/2011"); // Día y mes del sistema del año 2011
Fecha p(0, 1);       // Como (0, 1, 0): este día de enero de este año
```

Los constructores deben comprobar que las fechas que se van a construir sean correctas, es decir:

1. Que el día esté comprendido entre 1 y el número de días del mes, y
2. que el mes esté comprendido entre 1 y 12, y
3. que el año esté comprendido entre dos constantes que se definirán con los nombres *Fecha::AnnoMinimo* y *Fecha::AnnoMaximo* con valores respectivos 1902 y 2037. Estas dos constantes serán públicas.

```
Fecha q(31, 4);      // MAL: abril solo tiene 30 días
Fecha r(31, 13);     // MAL: solo hay 12 meses
Fecha s(12, 12, 2048); // MAL: 2048 > Fecha::AnnoMaximo
Fecha t("2/5/1808"); // MAL: 1808 < Fecha::AnnoMinimo
```

En caso de que esto no suceda, el constructor correspondiente elevará una excepción del tipo *Fecha::Invalida*, que llevará información de por qué ha ocurrido el fallo en forma de una cadena de caracteres de bajo nivel que se le pasará como parámetro al construirla, y que devolverá con un método público llamado *por_que*.

```
try { Fecha t("02/05/1808"); }
catch(Fecha::Invalida& e) { std::cerr << e.por_que() << std::endl; }
```

Una *Fecha* podrá incrementarse o decrementarse en 1 día mediante los operadores de incremento o decremento prefijos y sufijos, con la semántica habitual de dichos operadores.

```
Fecha q = ++hoy; // q vale: "mañana"; y hoy es mañana
Fecha r = q--;   // r vale mañana, pero q vuelve a hoy
```

A una *Fecha* podrá sumársele o restársele un número cualquiera de días mediante los operadores de suma y resta de *Fecha* y entero. Estos operadores devolverán otra *Fecha* que será el resultado de la original más o menos el número de días especificado por el operando entero.

```
Fecha s = hoy + 7; // s es 1 semana a partir de hoy
Fecha t = hoy - 30; // t es 1 mes (30 días) antes de hoy
```

Una *Fecha* podrá incrementarse o decrementarse un número cualquiera de días mediante los operadores suma o resta de *Fecha* y entero con asignación.

```
Fecha u = s += 7;      // u y s son 2 semanas a partir de hoy
fuu(t -= 30);          // t es ahora 60 días antes de hoy,
                        // y esa es la Fecha que recibe fuu
```

En los operadores antedichos, habrá que comprobar que la fecha resultante de la operación sea válida, ya que podría sobrepasarse el rango de años. Por ejemplo:

```
try {
    Fecha armagedon(31, 12, Fecha::AnnoMaximo),
        big_bang(1, 1, Fecha::AnnoMinimo);
    armagedon++;      // Error, desbordamiento superior en armagedon
    big_bang - 3;      // Error, desbordamiento inferior en la Fecha devuelta
} catch(Fecha::Invalida& e) { /* ... */ }
```

Una *Fecha* podrá asignarse a otra.

```
o = p;    // Ahora o vale el día actual de enero del año en curso
```

Una *Fecha* poseerá métodos observadores que devolverán los atributos. Estos métodos se llamarán *dia*, *mes* y *anno*.

```
int dia = hoy.dia();
int mes = hoy.mes();
int año = hoy.anno();
```

Una *Fecha* (constante o no) podrá convertirse implícitamente a una cadena de caracteres en el formato "*DIASEM DD de MES de AAAA*", donde *DIASEM* es el nombre del día de la semana en español, *DD* es el día del mes con 1 o 2 dígitos, *MES* es el nombre del mes en español y *AAAA* es el año expresado con 4 dígitos. Ejemplo: *miércoles 12 de septiembre de 2001*. Obsérvese que en español los nombres de los días de la semana y de los meses se escriben en minúscula, debe colocarse la tilde donde corresponda y los números de los años no llevan separador de miles, de forma que el ejemplo anterior estaría mal expresado como *Miercoles 12 de Setiembre de 2.001*. En el caso de septiembre, debe usarse esta forma y no la vulgar «setiembre». Esto es importante para que se pasen las pruebas automáticas. En el siguiente ejemplo, la *Fecha f* se convierte automáticamente a **const char***, y de esa forma ya puede imprimirse en la salida estándar *martes 11 de septiembre de 2001*.

```
std::cout << f << std::endl; // Imprime en la salida estándar
```

La conversión repetida de una misma fecha en cadena de caracteres conlleva un coste en tiempo que podemos reducir si mantenemos una copia de la cadena obtenida en la primera conversión, que se devolverá directamente en las sucesivas conversiones solicitadas, a menos

que la fecha haya cambiado, en cuyo caso se actualizará esta copia en la primera petición de conversión que ocurra tras el cambio. Así pues, se añadirán dos atributos a la clase *Fecha*: uno llamado *crep* para guardar la fecha en forma de cadena cuando sea convertida por primera vez, de tipo vector de **char** y longitud fija y suficiente para cualquier fecha; y otro booleano de nombre *actual*, que indicará si la cadena está actualizada. La posible actualización de la cadena ha de ser transparente para el usuario de la clase *Fecha*, de modo que aunque la fecha no debe ser alterada desde el punto vista lógico (para el usuario será la misma antes y después de la conversión), la cadena almacenada sí que ha de poder ser modificada internamente. La incorporación de los dos nuevos atributos implica cambios en algunos de los métodos ya descritos de la clase *Fecha*, por lo que deben ser revisados para realizar las modificaciones pertinentes.

Dos fechas podrán compararse mediante los operadores habituales de igualdad, desigualdad, mayor, menor, mayor o igual, y menor o igual, que devolverán un valor booleano con el significado lógico de «menor = antes» y «mayor = después».

```
a < b;           // Verdad, 1936 ya pasó hace mucho, es menor que este año
a <= c;          // Verdad, menor
hoy == Fecha(); // Verdad
s > t;           // Verdad
s >= u;          // Verdad
o != p;          // Falso
```

2. Clase Cadena

Se trata de hacer una clase general para trabajar con cadenas de caracteres (**char**), como una paupérrima imitación de *std::string* de la biblioteca estándar. Esta clase se llamará *Cadena* y sus atributos serán un puntero a caracteres de tipo **char** y un entero sin signo de tipo *size_t* que representará el tamaño de la cadena o número de caracteres en cada momento. El puntero antedicho apuntará a una cadena de caracteres acabada en el carácter terminador NUL o '\0', como las cadenas de C, lo que hace la implementación mucho más fácil. El atributo de tamaño no contará este terminador; dicho de otra forma, la *Cadena* formada por los caracteres 'h', 'o', 'l' y 'a', tiene de tamaño 4 y ocupa 5 *bytes* de memoria, 1 más que el tamaño por el terminador.

Los caracteres de una *Cadena* no vacía, acabados en el carácter '\0', se almacenarán en memoria dinámica asignada con **new**. Si una *Cadena* está vacía (o sea, su tamaño es 0), no tendrá asignada memoria dinámica y el puntero a los caracteres apuntará a la misma dirección que un tercer atributo de tipo **char**[1] cuyo único carácter será el terminador. Ya que el valor de este atributo va a ser común a todos los objetos, se deberá definir como atributo de clase.

No debería ser obligatorio, pero para que funcionen las pruebas automáticas y las comprobaciones de código, los atributos antedichos deben llamarse *vacía*, *tam_* y *s_* y ser declarados en este mismo orden (primero la cadena formada sólo por el terminador, después el entero y por último el puntero).

Una *Cadena* se construirá:

1. Con 2 parámetros, que serán por este orden: un tamaño inicial y un carácter de relleno.

```
Cadena a(3, 'X');           // tamaño y relleno: "XXX"
```

2. Con 1 parámetro, que será un tamaño inicial; en este caso la cadena se rellenará con espacios. No se permitirá la conversión implícita de un entero a una *Cadena*.

```
Cadena b(5);                // tamaño y espacios: "     " (5 espacios)
Cadena bb = 5;              // ERROR, conversión no permitida
void foo(const Cadena& c);
foo(7);                     // ERROR, conversión no permitida
foo(Cadena(7));             // OK, conversión explícita
```

3. Sin parámetros: se creará una *Cadena* vacía, de tamaño 0.

```
Cadena c;                   // Cadena vacía, tamaño 0: ""
```

4. Por copia de otra *Cadena*.

```
Cadena d(a);                // copia de Cadena: "XXX"
```

5. A partir de una cadena de caracteres de bajo nivel, permitiéndose las conversiones no explícitas desde **const char*** a *Cadena*.

```
Cadena f("Hola");          // copia de cadena de C: "Hola"
```

Una *Cadena* podrá asignarse a otra. Una cadena de bajo nivel también podrá asignarse directamente a una *Cadena*. La original se destruye. No obstante, si no hay memoria suficiente para realizar la copia, estas operaciones no tendrán efecto, es decir, el operando izquierdo conservará su contenido.

```
a = f;                      // Ya no existe a = "XXX", ahora a = "Hola"; si no hay memoria,
                             // a mantiene el valor "XXX"
a = "C++11";                // Ahora a = "C++11" (y f sigue valiendo "Hola"), pero, si falta
                             // memoria, a continúa con el valor que tuviera
```

Una *Cadena* podrá convertirse a una cadena de bajo nivel (**const char***). Se prohibirán las conversiones implícitas para evitar posibles conversiones indeseadas y problemas de ambigüedad. En el ejemplo siguiente, *puts()* es una función de la biblioteca estándar de C (y C++) que recibe un **const char*** e imprime los caracteres a los que apunta hasta su terminador en la salida estándar, seguido de un salto de línea.

```
std::puts((const char*)f); // Imprime "Hola\n" en la salida estándar
```

La función observadora *length()* devolverá el número de caracteres de una *Cadena*.

```
size_t longitud_de_saludo = f.length(); // longitud_de_saludo = 4
```

A una *Cadena* podrá concatenársele otra, añadiéndose esta al final, mediante el operador de suma con asignación. Si no se puede obtener memoria para llevar a cabo la concatenación, la cadena no se debe modificar.

```
a += f;    // a = "C++11Hola", f sigue igual: "Hola"; pero si no hay
           // memoria disponible, a también sigue igual: "C++11"
```

Dos objetos *Cadena* podrán concatenarse mediante el operador de suma, resultando una nueva *Cadena* que será la concatenación de ambas.

```
c = a + f; // asignación a c de la Cadena "C++11Hola" y "Hola"
           // c = "C++11HolaHola"
```

Dos objetos *Cadena* podrán compararse con los operadores lógicos habituales: igualdad, desigualdad, mayor que, menor que, mayor o igual y menor o igual. El resultado será un valor lógico (**bool**). Que una *Cadena* sea menor que otra significa que está antes en el sistema de ordenación alfabético según los códigos de caracteres. Si son iguales, es que tienen los mismos caracteres en el mismo orden y son de igual longitud. Observe que uno cualquiera de los 2 operandos podría ser una cadena de bajo nivel que se convirtiera automáticamente a *Cadena* con el constructor de conversión que se ha pedido anteriormente.

```
a = "XXX";    // b = "      " (5 espacios)
a < b;         // falso, 'X' > ' '
a > "xxx";     // falso, 'X' < 'x'
"XXX" == a;    // verdad
Cadena v;
v != Cadena();// falso, son 2 Cadena vacías
a <= "XXXX";   // verdad, es menor, porque 0 ('\0') < 'X'
a >= "XX";     // verdad, es mayor, porque 'X' > 0
```

Podrá obtenerse un carácter determinado de una *Cadena* mediante su índice en ella, para lo que se redefinirá o sobrecargará el operador de índice (corchetes) y una función *at()*. La diferencia es que el operador índice no comprobará si el número que se le pasa como operando está dentro del rango del tamaño de la *Cadena*, y la función *at()* sí lo hará. En este caso, si el parámetro de *at()* no está dentro del rango $0..length() - 1$, lanzará la excepción estándar *std::out_of_range*. Estas funciones de índice podrán aplicarse a una *Cadena* definida **const** para observación, y tanto para asignación como para observación si no es **const**. Es decir, por ejemplo:

```
const Cadena cc("hola");
Cadena c("ola");
cc[0] = ' ';    // ERROR, cc es const
char h = cc[0]; // OK, h = 'h'
c[0] = 'a';     // OK, c = "ala"
h = c[1];       // OK, h = 'l'
cc.at(0) = 'z'; // ERROR, cc es const
```

```

h = cc.at(0);    // OK, h = 'h'
c.at(0) = 'o';   // OK, c = "ola" de nuevo
h = c.at(1);     // OK, h = 'l'
cc[7];           // ERROR FATAL NO CONTROLADO DURANTE LA EJECUCIÓN
try {
    cc.at(7);     // Error controlado, se lanza std::out_of_range
} catch(std::out_of_range e) {
    std::cerr << e.what() << std::endl;
}

```

Cuando una *Cadena* salga fuera de ámbito o se destruya, deberá liberarse la memoria dinámica que tenga reservada, si es que tiene.

La función miembro *substr()* recibirá dos parámetros enteros sin signo: un índice y un tamaño, y devolverá una *Cadena* formada por tantos caracteres como indique el tamaño a partir del índice. Por ejemplo:

```

Cadena grande("NIHIL NOVVM SVB SOLEM"); // «Nada nuevo bajo el Sol»
Cadena nuevo = grande.substr(6, 5);     // nuevo = "NOVVM"

```

La función *substr* deberá lanzar una excepción *std::out_of_range* cuando se proporcione una posición inicial después del último carácter, o cuando la subcadena pedida se salga de los límites de la cadena. Por ejemplo:

```

Cadena s("hola hola");
s.substr(9, 1); // lanza std::out_of_range (9 está fuera del rango 0..8)
s.substr(6, 10); // ídem: después del índice 6 no hay 10 caracteres

```

Obviamente, ya que se está haciendo una imitación de *std::string*, está prohibido usar dicha clase, aunque pueden usarse otras funciones de la biblioteca estándar de C++ (que incluye la de C) que se estimen necesarias o convenientes. Tampoco debe usarse *std::string* en *Fecha*, sino cadenas de caracteres de bajo nivel, como en C. En las prácticas sucesivas, donde hubiera que emplear *string* se usará *Cadena* en su lugar.

3. Organización

Las prácticas de esta asignatura, *Programación orientada a objetos* o, abreviadamente, POO, deben guardarse en directorios con la siguiente disposición:

- El directorio raíz de las prácticas de POO tendrá el nombre del alumno en el formato *Apellido1_Apellido2_Nombre*; sin tildes y sin eñes, que se cambiarán por doble ene, y juntando nombres compuestos. Por ejemplo, el hipotético alumno Álvaro José Muñoz de la Minglanilla crearía el directorio *Munnoz_delaMinglanilla_AlvaroJose*.
- Bajo el directorio raíz anterior se creará un directorio para cada práctica llamado *Pn*, siendo *n* el número de práctica; es decir, los directorios P0, P1, P2, P3 y P4.

- Esta es la práctica 0, por lo que en P0 tendrá que escribir y tener los ficheros fuentes de la práctica; algunos se le darán a través normalmente del campus virtual, y otros, obviamente, los tiene que hacer el alumno. Para los ficheros fuente seguirá los convenios:
 - Los nombres de los ficheros estarán en minúsculas.
 - Los ficheros de cabecera de C++ tendrán de extensión `hpp`.
 - Los ficheros fuente en C++ tendrán de extensión `cpp`.
 - El fichero de descripciones para `make` se llamará `Makefile`.
- Bajo el directorio raíz se ubicarán también los siguientes directorios que se suministran en el campus virtual y que son válidos para todas las prácticas:
 - `Tests-auto`: contiene el código para realizar las pruebas unitarias automáticas.
 - `dsl-comprobaciones`: sirve para realizar las comprobaciones *check* (se explicarán en la siguiente sección). Incluye la biblioteca *libcac.a* (y también los fuentes y su *Makefile* para regenerar la biblioteca en caso necesario).

Es **muy recomendable** leer y estudiar atentamente la documentación proporcionada en el campus virtual. Sobre todo el PDF *consejos* (renombrado irónicamente a *NO_LEER_NUNCA*), donde se da información general sobre buenas prácticas al escribir programas en C++, y particular sobre cómo hacer diversas cosas de las que se piden en las clases *Fecha* y *Cadena*: saber si un año es bisiesto, saber la fecha actual, implementar los diferentes operadores, etc.

4. Makefile y pruebas a realizar

4.1. Makefile

Deberá añadir el *Makefile* que encontrará en la carpeta “Materiales de la P0” dentro de su carpeta P0. Este incluye las reglas necesarias para la compilación de los ficheros de *Fecha* y *Cadena*. Igualmente, este *Makefile* permitirá la compilación de los ficheros de pruebas proporcionados (que se listan en la siguiente subsección) y su enlazado con los módulos objeto *fecha.o* y *cadena.o*.

El *Makefile* proporcionado incluye las variables CXX para cambiar el compilador de C++ y la variable CXXFLAGS con las siguientes opciones:

- g Necesaria para poder depurar el programa.
- Wall y -Wextra Avisos extra; por si hay algo que puede dar problemas a la hora de la ejecución.
- std=ESTÁNDAR El estándar a seguir. *ESTÁNDAR* debe ser `c++17`.
- pedantic Para que el lenguaje se ajuste fielmente al estándar escogido, de forma que lo que sea incompatible con él produzca un error.

Puede usar otras opciones del compilador, consulte su manual. Para el preprocesador de C++, está definida la variable `CPPFLAGS` con las opciones siguientes:

- DPO En cada práctica, para que funcionen los tests, hay que definir una macro del preprocesador llamada como la práctica, en este caso, P0. Esto puede y debe hacerse al preprocesar, mediante la opción `-D`, que sirve para definir una macro del preprocesador que se pone a continuación (opcionalmente con un valor, que en este caso no hace falta).
- I../Tests-auto -I. Para que el preprocesador encuentre las cabeceras de los programas de prueba automáticos, hay que usar la opción `-I`, que le indica en qué directorio buscar.

Puede usar otras variables en el *Makefile*, a su gusto y conveniencia, así como otras reglas (siempre, claro está, que no rompa las reglas ya proporcionadas).

Sepa que, la última regla, cuyo objetivo se llamará *clean*, borrará del directorio todos los ficheros sobrantes o que puedan ser regenerados: módulos objeto, ejecutables, etc.

4.2. Pruebas a realizar

A continuación, se listan los tipos de prueba a realizar sobre el código desarrollado:

1. Test de consola Se trata de un programa que pone en juego las clases desarrolladas y que permite, a través de la consola, interactuar con el mismo y observar si la secuencia de acciones planteadas en el código ofrece el resultado esperable (a comprobar visualmente).

¿Qué es necesario? Incluir el fichero *test-fechacadena-consola.cpp* dentro de la carpeta P0.

¿Cómo lo ejecuto? Ejecutando `make test-consola`

Nota: Conviene estudiar el código de estas pruebas para comprender mejor lo que se pide en este enunciado.

2. Tests automáticos Este programa plantea y ejecuta una serie de escenarios de prueba, los cuales comprueban, de forma automática, si el estado del programa es el correcto tras la ejecución de esos escenarios.

¿Qué es necesario? La carpeta `Tests-auto` (ver Sección 3).

¿Cómo lo ejecuto? Ejecutando `make test-auto`

Nota: Sepa también que la primera regla del *Makefile*, cuyo objetivo se llama *all*, construye los dos programas de prueba anteriores, pero no los ejecuta.

3. Valgrind Se trata de una utilidad que analiza el uso de la memoria de un programa y que, por tanto, permite detectar problemas en la gestión de la misma.

¿Qué es necesario? Tener instalada la utilidad *valgrind* en el sistema.

¿Cómo lo ejecuto? Ejecutando `make valgrind` desde la carpeta de la práctica. Esta regla ejecutará la utilidad `valgrind` sobre los dos programas de prueba anteriores para analizar su uso de memoria.

4. Comprobaciones *check* Este tipo de pruebas permite analizar el código fuente de sus ficheros con el fin de detectar posibles desviaciones respecto a lo indicado en el enunciado, así como indicar sugerencias adicionales.

¿Qué es necesario?

- Incluir el fichero `fecha_cadena_check.cpp` y `Make_check.mk` dentro de la carpeta P0.
- La carpeta `dsl-comprobaciones` (ver Sección 3).

¿Cómo lo ejecuto? Ejecutando `make check` desde la carpeta de la práctica. Esta regla derivará a `Make_check.mk` la generación de un ejecutable de nombre `fecha_cadena_check`, y seguidamente lo ejecutará sobre sus ficheros `fecha.cpp` y `cadena.cpp`.