Programación Orientada a Objetos

Práctica 0: Consejos, trucos, avisos, buenas prácticas

Versión 2.0

1. Generalidades

- Procura familiarizarte con un buen editor y entorno: un poco de práctica te ahorrará muchísimo tiempo después. Nosotros te aconsejamos el editor Emacs, pero hay otros buenos, como Atom (libre) o Sublime Text (privativo y de pago). Puedes ver una comparativa en
  https://en.wikipedia.org/wiki/Comparison_of_text_editors#Programming_features

- Acostúmbrate a poner en cada fichero de cabecera las *guardas de inclusión múltiple*. Esto es: al principio del fichero, comentarios o líneas en blanco aparte, las dos líneas:

  ```
  #ifndef FICHERO_HPP_
  #define FICHERO_HPP_
  ```

  donde `FICHERO` es el nombre del fichero de cabecera, pero con todas las letras en mayúsculas; y al final del todo, la directiva del preprocesador

  ```
  #endif
  ```

  Esto hará que los contenidos del fichero de cabecera solo se incluyan una vez, aunque por error o por la complejidad de los ficheros, sea incluido más veces. Esto solo debe ponerse al principio y al final de cada fichero, una sola vez en cada uno, no una vez por cada clase si se define más de una en un fichero de cabecera. Con esto, podrás incluir en cada fichero fuente solo las cabeceras que necesites en él, y no tendrás problemas.

- En los ficheros de cabecera (extensión `hpp`) solo se deben poner, aparte de comentarios, líneas en blanco y espacios de nombres:

  definiciones de tipos: clases, estructuras, uniones, enumeraciones, sinónimos de tipos (*typedef*).

  declaraciones o prototipos de funciones externas.

  definiciones de funciones en línea: *inline*.

  definiciones de plantillas (*template*): clases y funciones

  definiciones de constantes no locales: *const*.

- En cambio, en los ficheros fuente (extensión `cpp`), y no en los de cabecera, se deben poner:

  definiciones de funciones no en línea (*inline*) ni genéricas (*template*).

  definiciones de variables globales no estáticas (de alcance global).

- Siempre que algo no deba ser modificado, aplica el calificativo *const*: punteros, referencias, parámetros de funciones punteros o referencias, funciones observadoras o que no deben modificar los atributos; no obstante, no tiene sentido pasar como parámetro a una función una variable de un tipo simple por valor como *const*. Ejemplos:

```
class Fred {
public:
  void mal(George, const int, char*); // observadora
  void bien(const George&, int, const char*) const; // observadora
  // ...
};
```

  En el primer parámetro de *mal*, de tipo *George* (cierta clase), se está copiando el objeto que se le pase, de esa clase. Para evitar la copia (y eso si está definida o se permite), es mejor pasar una referencia al objeto, y definirla *const* si no ha de modificarse. El segundo parámetro es un simple entero que se pasa por valor, por lo que hacerlo constante no tiene ningún sentido: impediría su modificación dentro de la función, pero es una copia de lo que se le pasa, que ya de por sí no es modificable por no ser visible desde dentro de la función, y la copia además es trivial, por lo que no se gana nada pasando el entero por referencia a constante. El tercer parámetro es un puntero: a menos que la función deba modificar el dato al que apunte (y para eso en C++ tenemos las referencias), ese puntero debería definirse como apuntando a algo no modificable. Por último, si un método no debe modificar los atributos de la clase, debe cualificarse también con *const*.

- Nunca pases un parámetro de un tipo complejo (como una clase) que no deba ser modificado en la función, por valor, sino por referencia *const*: así te ahorras una copia que puede ser costosa. Mira el ejemplo del punto anterior.

- Considera si hacer varios constructores o uno con parámetros predeterminados, si el enunciado lo permite. Normalmente la segunda opción es más corta y elegante.

- En los constructores, usa siempre que puedas la *lista de inicialización*. A veces es la única opción, otras veces es más conveniente, otras es innecesaria, pero nunca perjudica. Ejemplo:

```
Fred::Fred(const string& n, const string& a, const Fecha& f, float al)
  : nombre_(n), apellidos_(a), nacimiento_(f), altura_(al) {
  // ... otro código adicional ...
}

Mal::Mal(const string& n, const string& a, const Fecha& f, float al) {
  nombre_ = n;
```

```
  apellidos_ = a;
  nacimiento_ = f;
  altura_ = al;
  // ... otro código ...
}
```

- Procura siempre *inicializar* cualquier variable u objeto. *Inicializar* significa dar un valor inicial en el momento de la definición, no después, aunque sea inmediatamente después. Ejemplo:

```
char* c = new char[x]; // Bien. Esto es una inicialización de c.
char* c;               // Aquí no se inicializa. ¿A dónde apunta c?
c = new char[x];       // Esto es una asignación, no una inicialización.

string saludo("hola"); // Bien, inicialización.
string despedida;      // Inicialización con ctor. predeterminado, cadena vacía.
// ...
despedida = "agur";    // Asignación.
```

2. La clase *Fecha*

- Para obtener la fecha del sistema («hoy», si está bien establecida), conviene hacer uso de funciones de la biblioteca estándar de C, incorporada en la de C++, por ejemplo así:

```
#include <ctime>
...
std::time_t tiempo_calendario = std::time(nullptr);
std::tm* tiempo_descompuesto = std::localtime(&tiempo_calendario);
int dia  = tiempo_descompuesto->tm_mday;
int mes  = tiempo_descompuesto->tm_mon + 1;    // tm_mon: 0 (ene)..11 (dic)
int anno = tiempo_descompuesto->tm_year + 1900; // tm_year: años desde 1900
...
```

  El *tiempo de calendario* es el número de segundos transcurridos desde cierto momento arbitrario o punto en el tiempo, llamado la Era (en inglés *Epoch*), que en sistemas UNIX suele ser el 1 de enero de 1970 a las 0:00 h. Se debe guardar en una variable de tipo *time_t*, que es un tipo aritmético definido por el sistema. Como este tiempo es inmanejable e incomprensible, hay que transformarlo a algo mejor, y para eso está la función *localtime*, que devuelve ese tiempo descompuesto en sus componentes: horas, minutos, segundos, días, meses, etc., según la hora local; también existe *gmtime*, que usa la hora UTC (tiempo universal coordinado, también conocido como la hora del meridiano de Greenwich). Cada uno de estos componentes es un campo de una estructura llamada *tm*; mira en el Manual de UNIX *localtime*, por ejemplo.

  Otra opción es usar la biblioteca *chrono* de C++11, incluida en la estándar, que es muy completa, pero para lo que queremos hacer aquí es más complicado. Hay que incluir la cabecera `<chrono>`, donde se define, anidado en el espacio de nombres *std* que engloba a toda la biblioteca estándar, el espacio de nombres *chrono*:

```
std::chrono::system_clock::time_point hoy =
                            std::chrono::system_clock::now();
std::time_t tiempo_calendario = std::chrono::system_clock::to_time_t(hoy);
```

Quizá se ve mejor así:

```
using std::chrono::system_clock;
system_clock::time_point hoy = system_clock::now();
std::time_t tiempo_calendario = system_clock::to_time_t(hoy);
```

Se define *hoy* como un punto en el tiempo determinado por el reloj del sistema y se inicializa con una llamada a *now*, que proporciona la hora actual de un reloj determinado, el del sistema en este caso. Después se llama a *to_time_t* que, como su nombre indica, devuelve un tiempo de calendario a partir del punto en el tiempo que se le pasa. A partir de aquí ya se puede usar la función *localtime* como antes.

- Para calcular el día de la semana, es de utilidad otra función de C que viene en el mismo paquete que las anteriores: *mktime*. Su prototipo sería:

  ```
  std::time_t mktime(std::tm*);
  ```

  Se podría pensar en ella como la contraria de *localtime*, pues toma un tiempo descompuesto y devuelve el tiempo de calendario; pero esta no suele ser su característica más usada. Lo más interesante es que rellena los campos de la estructura *tm* relativos al día de la semana (*tm_wday*) y al día del año, y además normaliza la fecha, de forma que un 40 de octubre pasaría a ser 9 de noviembre.

- Un año es bisiesto si es múltiplo de 4, excepto si es el último de un siglo y no es múltiplo de 400. Por lo tanto, para calcular si un año a es bisiesto, esta expresión servirá; devolverá verdad si lo es.

  ```
  (a % 4 == 0 && a % 100 != 0) || a % 400 == 0
  ```

- ¿Dónde definir los operadores? ¿Dentro de la clase, como miembros, o fuera, como funciones externas?

  Hay operadores que no se pueden definir, como el operador condicional (?:); otros solo pueden definirse dentro de las clases, como los de asignación; otros solo fuera, como los de inserción y extracción de flujo, y otros tanto dentro como fuera. En estos casos, nuestro consejo es definir como externos los operadores binarios simétricos; esto es, aquellos cuyos dos operandos son del mismo tipo, como ocurre en esta práctica con los de comparación de fechas o cadenas, o de concatenación de cadenas; sobre todo cuando puede haber conversiones: normalmente querremos que estas se puedan producir implícitamente tanto en el operando izquierdo como en el derecho, lo que solo puede lograrse cuando los operadores se definen externamente. En los otros casos, es preferible definir los operadores como miembros de clase.

- En la clase *Fecha*, para los operadores aritméticos, basta hacer el operador de suma con asignación, `+=`, y los demás en función de él, ya que se dan las equivalencias siguientes (f es la *Fecha*, t una *Fecha* temporal y n un número entero):

```
f -=n => f += -n
f + n => t = f, t += n
f - n => t = f, t += -n
++f   => f += 1
f++   => t = f, f += 1, t
--f   => f += -1
f--   => t = f, f += -1, t
```

Y para el operador `+=` podemos usar la propiedad normalizadora de fechas de *mktime*: tenemos que pasarle la dirección de una estructura *tm* que rellenaremos de forma que el campo del día esté incrementado en n (que podrá ser negativo), y esta función recalculará los campos para darnos una fecha correcta. Mira en el Manual de Unix los campos de la estructura *tm* para saber el orden y nombre de los campos.

En cuanto a los de comparación, basta con el de igualdad y el menor-que. Los demás se pueden hacer en función de estos (y de hecho, así están ya definidos en la Biblioteca Estándar, en la cabecera `<utility>` en el espacio de nombres *std::rel_ops*):

```
a != b => !(a == b)
a >  b => b < a
a <= b => !(b < a)
a >= b => !(a < b)
```

- No definas la copia (constructor de copia y operador de asignación) si no hace falta: si lo que vas a hacer es justamente el comportamiento proporcionado de forma predeterminada (copiar los atributos), no vas a mejorarlo y puedes empeorarlo.
- Para el constructor de *Fecha* que recibe una cadena de caracteres de bajo nivel (constructor de conversión), lo más fácil en esta práctica, donde aún no se ha visto la biblioteca de E/S *IOStream*, es hacer uso de otra función de la biblioteca estándar de C: *sscanf*, cuyo primer parámetro sería esa cadena de caracteres, que es la fuente de donde *sscanf* leerá caracteres. Además, *sscanf* devuelve el número de conversiones que ha hecho, de forma que si no devuelve 3, es que el formato no era el adecuado (número, barra, número, barra, número).
- En el operador de conversión a cadena de caracteres, aparte de la ya mencionada *mktime* para calcular el día de la semana, puede ser de ayuda también la función *strftime* para transformar una fecha codificada en una estructura `tm` a una cadena de caracteres definiendo el formato, es decir, los campos de la fecha que queremos que aparezcan.

3. La clase *Cadena*

- No reinventes la rueda: está prohibido usar la clase *string* ya que estamos precisamente haciendo una (paupérrima) versión de un subconjunto de ella, pero por supuesto se pueden usar las funciones de C de cadenas de bajo nivel, como *strlen*, *strcpy*, etc. Puedes consultar el Manual de Unix para cada una. Para ello, el atributo puntero a caracteres de la clase debe apuntar a una cadena de caracteres que acaben en el carácter terminador `'\0'`, que no cuenta para la longitud.

- Nunca uses las funciones de C de reserva y liberación de memoria **alloc* y *free* en C++; en su lugar emplea los operadores *new* y *delete* (o *new[]* y *delete[]* para ristras). Y piensa que para cada *new* o *new[]* debería haber un *delete* o *delete[]* correspondiente, para que no se desperdicie memoria.