Guía y Normas de Programación de las Prácticas de EDA

Rosa M. Jiménez Conrado Martínez

Contenidos

Introducción

En la primera parte de esta *Guía* se recogen consejos útiles, recomendaciones e información complementaria al material presentado en las sesiones de laboratorio. Conviene por lo tanto tener a mano las transparencias de las sesiones mientras se lee este documento. También es importante consultar la bibliografía básica o algunos de los múltiples tutoriales, FAQs, etc. sobre C++ disponibles en Internet. Debe entenderse que esta *Guía* es tan sólo un complemento y no un manual de programación o del lenguaje de programación C++.

En la segunda parte (Sección 9) de este documento el estudiante encontrará las normas, con indicaciones precisas sobre lo que se debe y lo que no se debe hacer en una práctica de esta asignatura. Por otra parte, la guía da consejos cuyo seguimiento no sólo facilitará la labor del estudiante sino que será valorado positivamente.

En las prácticas de la asignatura se utiliza el paradigma de *programación basada en objetos* (OBP, *Object-Based Programming*) y las recomendaciones contenidas en esta guía deben entenderse en dicho contexto. Aunque resulta cómodo emplear el término *programación orientada a objetos* (OOP, *Object-Oriented Programming*) para referirse también a la OBP, la OBP es sólo la base de la OOP, ya que conceptos primordiales y característicos de la OOP tales como la *herencia* y el *polimorfismo* no se utilizan en la OBP.

1 Implementación de TADs

Un tipo abstracto de datos se implementa en C++ mediante el concepto de *clase*. Una clase consta de elementos públicos y elementos privados. A veces una clase soporta diversas funciones pero no corresponde a un TAD, a una abstracción de datos. Emplearemos el término *módulo* para referirnos a estas clases. Por ejemplo, podemos tener un módulo que ofrezca diversas funciones matemáticas. El lenguaje C++ tiene mecanismos alternativos al de las clases para definir módulos; p.e. los `namespaces`. Éstos se describen brevemente en la subsección 1.1.

Por lo general, los elementos públicos son las operaciones disponibles sobre los objetos de la clase, mientras que los privados incluyen la representación del TAD y las operaciones privadas. Si definimos una clase mediante `class` entonces todos los elementos son privados por defecto, salvo que se diga lo contrario. Por el contrario, si definimos una clase mediante `struct` todos los elementos son públicos por defecto. Las palabras reservadas `public` y `private` se usan para explicitar el tipo de acceso permitido. En general se debe utilizar `class` y no `struct` para definir clases.

Típicamente, la *declaración* de la clase se escribe en un fichero de cabecera o *header*, con extensión `.hpp`. La definición de los métodos de la clase, públicos o privados, se escribe en otro fichero, el de *implementación*, con extensión `.cpp`. También se usan las extensiones `.cc` y `.C` para los ficheros fuente y las extensiones `.hh` y `.h` para los ficheros de cabecera.

Observad que habitualmente la declaración de los métodos en una clase no corresponde a la forma funcional de la signatura abstracta del TAD al que implementa la clase. Así por ejemplo, la operación `Unir` de un TAD Conjunto es una función que dados dos conjuntos devuelve un tercero. Por el contrario, en la clase conjunto que implementa al TAD será típico que se incluya un método `unir` que se aplica a un objeto a y recibe como parámetro otro objeto b. El método modifica el objeto a de tal modo que al finalizar a contenga el resultado de unir (en el sentido abstracto) el conjunto previamente representado por a con el conjunto representado por b. A nivel de TAD, $\mathtt{unir}(A, B) = A \cup B$. Sin embargo, si a y b son objetos de la clase conjunto, a nivel de clase tenemos:

```
// a = A ∧ b = B
a.unir(b);
// a = A ∪ B ∧ b = B
```

La especificación de un TAD se da en términos de operaciones sobre valores del tipo, mientras que una clase ofrece operaciones sobre objetos del tipo en cuestión. Esta diferencia, que puede parecer sutil, es la que justifica la existencia de operaciones de creación, copia y destrucción (véase la sección 3) y, de hecho, es uno de los principios básicos de la OOP.

A fin de implementar una clase X es corriente definir una o más clases auxiliares. Normalmente, la definición se "anida" dentro de la parte privada de X, lo que hace que las clases auxiliares sean privadas para cualquier clase externa a X. Para que X pueda tener acceso a todos los elementos de una clase auxiliar Y, definiremos Y mediante `struct`. Esto es adecuado si Y es relativamente simple. Si la clase auxiliar es "compleja" entonces conviene definirla mediante `class` y dotarla de operaciones que faciliten su manejo (incluyendo constructoras, destructora, asignación, etc.). Además suele ser útil emplear la técnica de las definiciones pospuestas en tales situaciones (véase más abajo).

Por ejemplo, si queremos implementar una pila de enteros mediante una lista enlazada simple convendría definir una clase auxiliar nodo:

```
class pila {
  public:
     pila();
     ~pila();
     void apilar(int x);
     void desapilar(void);
     int  cima() const;
     ...
  private:
     struct nodo {
        int info;
        nodo* siguiente;
     };
     nodo* cim;    // la pila se representa mediante un puntero
```



```
                  // al primer nodo (cima) de la pila y
     int nr_elems; // un contador de elementos
};
```

La definición de una clase anidada se puede posponer, siempre y cuando no se declaren objetos de esa clase antes de que aparezca la definición. Por ejemplo, supongamos que posponemos la definición de la clase nodo. Cuando el compilador procesa la definición de pila sabe cuánto ocupa un puntero a nodo (lo mismo que cualquier otro puntero), pero no cuánto ocupa un nodo, ya que la definición se encuentra en algún otro lugar.

```
// pila.hpp
class pila {
  public: pila();
          ~pila();
          ...
  private: struct nodo; // declaramos la existencia de la clase 'nodo',
                        // pero no lo definimos todavia
           nodo* cim;   // CORRECTO
           nodo x;      // ERROR! todavia no se ha definido 'nodo'
};

// pila.cpp
struct pila::nodo {     // hay que indicar que se quiere definir
   int info;            // la clase 'nodo' anidada en la clase
   nodo* siguiente;     // 'pila' con el operador ::
};
...
```

La posibilidad de posponer definiciones, o equivalentemente *anticipar* declaraciones (*forward declarations*), es interesante, y a veces imprescindible; nos permite ocultar completamente y de manera totalmente efectiva la representación "real" pero, a cambio, ésta habrá de ser siempre accedida a través de un puntero o referencia. Otro inconveniente de esta técnica es que no se puede aplicar, con los compiladores actualmente existentes, para clases genéricas, p.e. una pila de elems.

En general, la técnica de las declaraciones anticipadas sigue el esquema que veíamos con el ejemplo anterior:

```
// mi_clase.hpp
class mi_clase {
  public: ...
  private:
     struct mi_clase_impl;  // no se dice aqui como se implementa
                            // realmente
     mi_clase_impl* repr;   // solo hay un puntero al objeto
                            // propiamente dicho
```

```
};

// mi_clase.cpp
...
struct mi_clase::mi_clase_impl {
  ... // definicion real de la representacion
};
...
```

Para conseguir una cierta independencia entre la parte pública y la parte privada de las clases y garantizar que los *headers* de las clases que tendréis que implementar en vuestras prácticas no se puedan modificar por accidente, se ha optado por utilizar una técnica no estándar pero más sencilla que forzar el uso de las declaraciones anticipadas. Ésta consiste en escribir la parte privada de una clase en un fichero independiente que por convenio tiene el mismo nombre que el *header* pero extensión `.rep` y que se incluye en el *header* mediante la directiva `#include`:

```
// pila.hpp
class pila {
  public: ...
  private:
     #include "pila.rep"
};

// pila.rep
struct nodo { ... }; // puesto que ‘nodo’ se define dentro
nodo* cim;           // de la clase ‘pila’ no hace falta el
...                  // calificador pila::
```

Evidentemente, esta técnica no impide el uso de declaraciones anticipadas, pero tampoco obliga a su uso.

En el fichero de implementación habrá de utilizarse el operador de resolución de ámbito (`::`) para indicar en qué contexto se define un determinado elemento (clase, función, etc.). Una vez desambiguado el contexto, se puede omitir el uso de este operador. Por ejemplo:

```
// pila.cpp
nodo* pila::f(int x) { // ERROR: el compilador no sabe que es ’nodo’
   ...
}

pila::nodo* pila::g(int x) { // OK: nos referimos a la clase ’nodo’
                             // definida dentro de la clase ’pila’
   nodo x;                   // OK: el compilador ya tiene la "pista" de
                             // que trabajamos con ’nodo’ de la clase
   ...                       // ’pila’
}
```



```
int pila::h() {
   pila::nodo y;             // OK: aunque se puede omitir ‘pila::’
                             // ya que el contexto queda "fijado" al
   ...                       // escribir pila::h()
}
```

En aplicaciones reales, no sujetas a las especiales condiciones de la asignatura (un número elevado de equipos resolviendo todos la misma práctica, ejecución de juegos de pruebas automatizada, etc.) la técnica del fichero `.rep` no ofrece ninguna ventaja clara y no se usaría. Se pondría directamente la definición de la representación en el fichero `.hpp` o se usaría la técnica de declaraciones anticipadas.

1.1 *Namespaces*

Un espacio de nombres (*namespace*) agrupa a una colección de elementos (clases, funciones, objetos, . . .) lógicamente relacionados entre sí, al tiempo que permite evitar los conflictos de nombres. Por ejemplo, el espacio de nombres `util` definido por

```
namespace util {
   typedef unsigned char byte;
   int toint(const string& s);
   class Random;
};
```

agrupa el tipo `byte`, la función `toint` y la clase `Random`. Estos elementos difieren de cualesquiera otros que se hayan definido en el espacio de nombres global[1] u otro espacio de nombres aunque coincidan los nombres.

Para referirnos a un elemento definido en un *namespace* `X` se pondrá `X::` delante del nombre. Por ejemplo,

```
util::Random r;
util::byte x = r(256);
string s = util::tostring(static_cast<int>(x));
```

Para evitar la incomodidad de prefijar las funciones, clases, etc. definidas en un *namespace* cada vez que se usan podemos emplear las cláusulas `using`. Por ejemplo, si escribimos `using util::byte;` cualquier referencia posterior a `byte` se refiere al definido en `util`. Con `using namespace util;` todos los identificadores definidos en el *namespace* `util` son accesibles globalmente:

[1]Todo identificador que no se haya definido dentro de un *namespace* se supone definido en el *namespace* global.

```
using namespace util;

Random r;
byte x = r(255);
string s = tostring(static_cast<int>(x));
```

El uso de las cláusulas `using` debe ser limitado, en particular en su segunda forma, pues de lo contrario se pueden producir conflictos de nombres. Si en algún caso es necesario desambiguar, siempre puede usarse la sintaxis *`namespace`*`::`*`identificador`*; para el espacio de nombres global se puede usar `::`*`identificador`*.

```
namespace X {
   int x;
   ...
}

int x;

int f(double& y) {
    y = X::x + ::x;
}
```

2 Apuntadores y gestión de memoria dinámica

Dado un tipo T, T`*` denota el tipo de los apuntadores a objetos del tipo o clase T. Una variable de tipo T`*` almacena la dirección de memoria de un objeto de tipo T. Cualquiera que sea el tipo T, un apuntador de tipo T`*` puede contener el valor especial 0 (`NULL`) para indicar que no apunta a ningún objeto. Se dice entonces que el apuntador es *nulo*.

Si `p` es de tipo T`*` entonces la expresión `*p` denota al objeto apuntado por `p` (salvo que `p` sea nulo). El operador `*` se denomina de *dereferencia*. Dado un objeto `x` de tipo T, `&x` es una expresión de tipo T`*` que denota la dirección de memoria en la que está `x`. El operador `&` se denomina de *referencia*. No obstante, a diferencia de lo que sucede en C, en C++ es relativamente poco frecuente el uso de estos operadores.

Frecuentemente un apuntador `p` apunta a un objeto que pertenece a una clase. Se puede invocar a un método público del objeto apuntado mediante el operador *flecha* (`->`), equivalente al uso del operador de dereferencia (`*`) y a continuación el de selección (`.`). Por ejemplo,

```
#include "pila.hpp"
pila p;         // p = [ ]
pila* q = &p;   // inicializamos q para que apunte a p

p.apilar(3); p.apilar(5);  // p = [ 3, 5 ]
cout << p.cima();          // imprime un 5
```



```
cout << q -> cima();        // imprime un 5
cout << (*q).cima();        // imprime un 5
cout << q.cima();           // ERROR! q *no* es una pila!
cout << q -> cim;           // ERROR! 'cim' es un atributo privado
                            // de la clase pila
```

Para crear y destruir objetos en memoria dinámica se utilizan los operadores `new`, `delete`, `new[]` y `delete[]`. Los dos últimos se emplean para crear y destruir tablas de objetos.

```
int* p = new int;
int* q = new int[10]; // se usa new[] para crear una tabla de 10
                      // enteros; q apunta al inicio de la tabla
```

Si un puntero `q` apunta a una tabla creada con `new[]`, la memoria debe ser liberada con `delete[] q`. Análogamente, si `p` apunta a un objeto creado con `new`, entonces el objeto se destruirá con `delete p`. Es importante tener en cuenta que los objetos creados en memoria dinámica son "anónimos" y por tanto accesibles únicamente a través de apuntadores. Un objeto creado en memoria dinámica existe hasta que no sea destruído explícitamente (o termine la ejecución del programa).

Los operadores `new` y `delete` no se limitan a crear el espacio de memoria o liberarlo. Una vez creado el espacio necesario para el objeto, `new` invoca al constructor apropiado para inicializar al nuevo objeto. Por su parte, `delete` aplica el destructor de la clase a la que pertenece el objeto apuntado y después libera la memoria ocupada por el objeto. Lo mismo sucede con `new[]` y `delete[]`.

Algunos problemas comunes en la gestión de la memoria dinámica incluyen:

- No "emparejar" correctamente los operadores: por ejemplo, intentar destruir con `delete` una tabla creada con `new[]`.

- Liberar un objeto que no ha sido creado con memoria dinámica o ya ha sido liberado (*dangling references*):

  ```
  int x;
  int* p = new int;
  int* q = &x;
  delete q;     // ERROR: q no apunta a un objeto de mem. din.
  q = p;
  delete p;     // OK
  delete q;     // ERROR: el objeto al que apuntaba q ha dejado
                // de existir
  ```

- Perder el acceso a objetos creados con memoria dinámica y por tanto no tener la posibilidad de destruirlos (*memory leaks*).

```
void f() {
   int* p = new int;
   *p = 3;
}
// MAL! cuando finaliza la ejecucion de la funcion f
// la variable local p deja de existir y no tenemos forma
// de acceder al objeto

int f2() {
   int* p = new int;
   int* q = new int;
   p = q;           // MAL! perdemos el acceso al primer objeto
   ...
}

int* g(int x) {
   int* p = new int;
   *p = x;
   return p;
}
// OK: se puede "recoger" el puntero al objeto creado
// dinamicamente en el punto de llamada a la funcion g;
// pero esta forma de trabajar no es aconsejable

void h(nodo* p) {
   nodo* q = new nodo;
   p -> sig = q;
}
// OK: se tiene acceso al nuevo nodo a traves del apuntador 'sig'
// del nodo apuntado por un puntero externo a la funcion h (el
// que se use como parametro actual en la llamada)
```

- Dereferenciar (con `*` o con `->`) un puntero nulo. Este error tiene casi siempre resultados catastróficos (*segmentation fault*, *bus error*, ...). Por ejemplo,

```
bool esta(const lista& L, int x) {
nodo* p = L.primero;
   while (p != NULL && p -> info != x)
      p = p -> sig;
   return (p -> info == x);  // ERROR!! si p == NULL, p -> info es
                             // incorrecto!
}
```

- Usar o implementar incorrectamente constructores por copia o el operador de asignación para clases implementadas mediante memoria dinámica (problema del *aliasing*).


```
char s[] = "abc"; // s[0] = 'a', s[1] = 'b', s[2] = 'c',
                  // s[3] = '\0'
char t[10];

t = s;
cout << t[0]; // imprime 'a'
t[0] = 'b';
cout << s[0]; // imprime 'b'!! t es en realidad un char* y la
              // asignacion t = s hace que t apunte a s[0]

class estudiante {
  public: estudiante(char* nom, int dni);
          char* consulta_nombre();
          int consulta_dni();
          ...
  private: char* _nombre;
           int _dni;
};

estudiante a("pepe", 45218922);

estudiante b = a; // el constructor por copia de oficio no sirve!

a = b; // el operador = de oficio es inadecuado!
```

- Devolver un puntero o referencia a un objeto local de una función. El problema es que al terminar la función, el objeto local se destruye y el puntero o referencia cesa de apuntar a un objeto válido.

```
int& maximo(int A[], int n) {
   int max = 0;
   for (int i = 0; i < n; i++)
      if (A[i] >= A[max]) max = i;
   return A[max]; // OK: se devuelve una referencia a A[max]
}

int& maximo(int A[], int n) {
   int max = 0;
   for (int i = 0; i < n; i++)
      if (A[i] >= max) max = A[i];
   return max; // ERROR: se devuelve una referencia a variable local!
}
```

- Hacer `delete p` con `p == NULL` no es erróneo. No tiene ningún efecto y ocasionalmente su uso ayuda a simplificar el código.

2.1 El módulo mem_din

A fin de poder evitar algunos de los problemas considerados más arriba, facilitar al estudiante la labor de depuración (*debugging*) de sus programas[2] y poder evaluar la correcta gestión de memoria dinámica, en las prácticas de EDA se usará el módulo mem_din para manipular memoria dinámica.

Para ello, basta con emplear la librería libeda al crear el ejecutable (véase la sección 8). En dicha librería se reemplazan los operadores estándar por los operadores new, new[], delete y delete[] de mem_din.

Una diferencia entre los operadores new y new[] estándar y los de mem_din es que los primeros lanzan excepciones (véase la sección 4) del tipo bad_alloc cuando no hay suficiente memoria, mientras que los segundos lanzan objetos de la clase error (definida en <eda/error>).

```
// pila.cpp
...
void apilar(int x) {
   nodo* p = new nodo; // 'new' de 'mem_din'
   p -> info = x;
   p -> sig = cim;
   cim = p;
}

void desapilar() {
  if (cim == NULL)
     throw error(...);
  nodo* p = cim;
  cim = cim -> sig;
  delete p;            // 'delete' de 'mem_din'
}
```

Pero además el módulo mem_din ofrece dos operaciones que nos dan control sobre la memoria dinámica:

- set_parameters
- print_memory_status_report

Ambas se invocan anteponiendo mem_din:: y hay que incluir la cabecera <eda/mem_din> en los ficheros fuente donde se quieren usar. La primera permite cambiar el número máximo de objetos (*chunks*) que pueden estar simultáneamente asignados en memoria dinámica y el número máximo de bytes (en total). Por defecto, al iniciarse el programa y mientras no se cambien mediante set_parameters, estos valores vienen dados por las constantes

[2] Si se usan los operadores new y delete estándar es frecuente que el programa aborte sin dar pistas sobre la causa.

DEFAULT_MAX_CHUNKS y DEFAULT_MAX_SIZE, respectivamente. Estas constantes se definen en el módulo mem_din y son públicas. La operación set_parameters no puede usarse si en ese momento existen objetos asignados en la memoria dinámica. Por lo demás, puede utilizarse en cualquier punto de un programa o en cualquier módulo (siempre que se haya incluído <eda/mem_din>). La operación print_memory_status_report imprime un resumen del estado de la memoria en ese instante. Tiene dos parámetros. El primero es de tipo ostream y es el canal por el cual se imprime el informe. Su valor por defecto es el canal estándar de salida cout. El segundo, de tipo entero, controla el nivel de detalle del informe generado. Su valor por defecto es 1. Si este parámetro vale 0 sólo se imprime el número de objetos dinámicos todavía "vivos" y el número de delete's incorrectos. Cuando un programa que usa mem_din finaliza se imprime en cout automáticamente un informe del estado de memoria (de nivel 0). Los informes de nivel 1 contienen la siguiente información:

- número máximo de objetos asignables
- número máximo de bytes asignables
- número de objetos asignados en ese instante
- número de bytes asignados en ese instante
- número de news y new[]s realizados desde el inicio del programa
- número de veces que se han liberado objetos que no habían sido asignados en memoria dinámica o ya habían sido liberados previamente
- número de veces que se ha liberado un bloque con un operador inadecuado (p.e. new[]-delete)

El uso de estas operaciones es simple. Por ejemplo:

```
#include <eda/mem_din>
...
int main() {
   mem_din::set_parameters(500, 10000);
   ...
   mem_din::print_memory_status_report();
   ...
}

#include <eda/mem_din>
...
int main() {
   mem_din::set_parameters(mem_din::DEFAULT_MAX_CHUNKS / 2, 10000);
   ...
   mem_din::print_memory_status_report(cout, 0);
   ...
}
```

Las operaciones del módulo mem_din lanzan excepciones de la clase error (véase la sección 4). Los operadores `new` y `new[]` lanzan el error `FaltaMemoriaDin` (código: 100) y ningún otro. Los operadores `delete` y `delete[]` no lanzan excepciones. El método `set_parameters` puede lanzar diversas excepciones (p.e., los valores de los parámetros no son correctos o se pide un cambio del tamaño habiendo objetos en uso en la memoria dinámica). En vuestra versión final de la práctica no debéis incluir ninguna llamada a `set_parameters` ni a `print_memory_status_report`. Por lo tanto sólo tenéis que preocuparos eventualmente del error `FaltaMemoriaDin` y olvidad las restantes excepciones que puede lanzar mem_din.

3 Creación, copia y destrucción de objetos

Toda clase de C++, tanto si se define mediante `class` como si se define mediante `struct`, tiene necesariamente los siguientes métodos:

- Constructor por copia: es el constructor que se emplea al crear un nuevo objeto a partir de uno existente; se emplea de manera directa, p.e.:

```
pila p = q; // la nueva pila p es una copia de la pila q
pila p1(q); // la nueva pila p1 es una copia de q; equivalente a
            // pila p1 = q;
pila* r = new pila(q); // la nueva pila apuntada por r
                       // es copia de q
```

  en el paso por valor y en el retorno por valor, p.e.:

```
pila inversa(pila p) { // correcto, aunque ineficiente
   pila q;
   ...
   return q;
}
```

  donde el parámetro formal `p` contiene una copia del parámetro actual y el resultado de la función es una copia de `q`. Tanto `p` como `q` son objetos locales y por tanto son destruídos al finalizar la función.

  El perfil de la constructora por copia es siempre de la forma X`(const` X`&)`, es decir, recibe un parámetro de la clase X por referencia constante.

- Destructor: se emplea al destruir objetos de la clase. No tiene parámetros y no puede sobrecargarse. Su perfil es siempre `~`X`()`. Nunca se invoca de manera explícita.

- Asignación: redefine el operador de asignación `=`; es similar al constructor por copia, pero el objeto modificado (la parte izquierda) es un objeto ya existente y devuelve una referencia al objeto destinatario de la copia. Su perfil es X`& operator=(const` X`&)`. El operador retorna una referencia al objeto que recibe la copia permitiendo así el uso de expresiones tales como `a = b = c;`. Otros operadores relacionados con el de asignación (tales como `+=`, `-=`, etc.) suelen tener el mismo perfil por análoga razón.

En aquellos casos en que el programador no defina explícitamente alguno de estos métodos, el compilador los provee de manera automática. Los denominaremos *métodos de oficio*. Además, en el caso en que no se haya definido ningún constructor, el compilador provee un constructor por defecto de oficio. Se denomina constructor por defecto al que puede ser invocado sin suministrar ningún parámetro, bien porque no los tiene, bien porque se han definido valores por defecto para todos sus parámetros.

Frecuentemente los métodos de oficio bastan, pero en otras ocasiones no hacen lo que se espera de ellos; en particular, si los objetos de la clase tienen uno o más atributos a crear dinámicamente.

Todos los constructores (por defecto o no, de oficio o no) invocan a los constructores de los atributos y luego ejecutan lo que haya indicado el programador en el cuerpo del constructor, en su caso. Si definimos un constructor debemos tener en cuenta que, salvo que utilicemos listas de inicialización, los atributos del objeto se inicializarán usando los constructores por defecto correspondientes. Por ello hay ocasiones en las que es imprescindible usar listas de inicialización en un constructor, sea o no por defecto: p.e., si un atributo es `const` o es una referencia se ha de inicializar en la lista de inicialización, ya que constantes y referencias deben necesariamente inicializarse y luego ya no se pueden modificar. Otro caso sería el de un atributo perteneciente a una clase para la cual no existe constructor por defecto.

```
 class Y {
    public:  Y(int n);
    ...
 };

 class X {
    public:  X(string id);
             ...
    private: const string _id;
             int _serial_nr;
             Y _y;
             ...
};

X::X(string id) { _id = id; ... }
  // MAL! _id ya se ha inicializado con el valor por defecto
  // para la clase string (""): ya no se puede inicializar _y

X::X(string id) : _id(id), _y(0) { ... } // OK
```

Una situación que surge frecuentemente es la necesidad de definir una o más constructoras para una clase auxiliar (definida mediante `class` o `struct`) que permitan invocar a las constructoras adecuadas de algunos de sus atributos. El siguiente ejemplo ilustra este punto:

```
class persona {
```

```
   public: persona(string nom, int edad);
           ...
}

class lista_personas {
   public: ...
           void inserta(const persona& q, ...);
           void elimina(...);
   private:
      struct nodo {
         persona per;
         nodo* next;
         // operaciones constructoras de la clase nodo
         nodo(const persona& q);
         nodo(string nom, int edad);
      };
      ...
};

lista_personas::nodo::nodo(const persona& q) : per(q), next(NULL) {}

lista_personas::nodo::nodo(string n, int e) : per(persona(n,e)),
   next(NULL) {}

...

void lista_personas::inserta(const persona& q, ...) {
   ...
   nodo* n = new nodo(q);
   ...
}

void lista_personas::elimina(...) {
   ...
   // 'n' apunta al nodo a eliminar
   delete n;
   ...
}
```

Si la clase auxiliar `nodo` no tuviese al menos una de las dos constructoras que se dan en el ejemplo, el atributo `per` no podría ser correctamente inicializado al crear un objeto del tipo `nodo`. Obsévese que el atributo tiene que ser necesariamente inicializado en la lista de inicialización.

Una alternativa, más complicada y engorrosa, consiste en tener punteros a los atributos que no dispongan de constructora por defecto. Esto exige gestionar explícitamente la memoria dinámica empleada para manipular estos objetos:


```
class lista_personas {
   public: ...
           void inserta(const persona& q, ...);
           void elimina(...);
   private:
      struct nodo {
         persona* per;
         nodo* next;
      };
      ...
};
...
void lista_personas::inserta(const persona& q, ...) {
   ...
   nodo* n = new nodo;
   n -> per = new persona(q);
   n -> next = NULL;
   ...
}

void lista_personas::elimina(...) {
   ...
   // 'n' apunta al nodo a eliminar
   // el objeto apuntado por 'per' tiene que ser liberado previamente!
   delete n -> per;
   delete n;
   ...
}
```

Por otro lado si inicializamos los atributos usando listas de inicialización evitamos trabajo redundante; en otro caso, cada objeto se inicializa usando su constructor por defecto y luego tenemos que asignarle algún valor dentro del cuerpo. Conviene por lo tanto usar las listas de inicialización siempre que resulte factible.

```
class moto {
  ...
  private:
     int cilindrada;
};

moto::moto() { cilindrada = 250; } // OK: pero primero se inicializa a 0
                                   // y luego se le asigna el valor 250
moto::moto() : cilindrada(250) {}  // Mejor! Se inicializa directamente
                                   // `cilindrada' con el valor 250
```

Todo destructor ejecuta lo que se haya indicado al definirlo explícitamente (el de oficio no hace nada) y a continuación invoca a los destructores de cada uno de los atributos. Es importante definir un destructor si los objetos de la clase usan recursos tales como memoria dinámica o canales para entrada/salida con ficheros, pues deben liberarse tales recursos antes de destruir el objeto.

El constructor por copia de oficio invoca los correspondientes constructores por copia de los atributos, uno a uno. Si un atributo es un apuntador el constructor por copia de oficio sufre del problema de *aliasing*.

Otro tanto sucede con la asignación de oficio: aún peor, ésta no invoca al destructor sobre el objeto que recibe la copia, con lo que adicionalmente se producen *memory leaks*. Por ejemplo, si implementamos una pila mediante una lista enlazada de nodos en memoria dinámica, representamos cada pila por un apuntador al nodo de su cima, pero la asignación `p = q;` es inadecuada para hacer la copia de pilas ya que sólo conseguiríamos perder el acceso al nodo originalmente apuntado por `p` y hacer que `p` apunte al mismo nodo que `q`, no a una copia.

Una estrategia útil para implementar el destructor, el constructor por copia y la asignación para clases que usan memoria dinámica consiste en definir una o más operaciones privadas que gestionan correctamente la copia y la destrucción evitando el *aliasing* y los *memory leaks*. Supongamos que se llaman `copia_x` y `destruye_x`. Una posibilidad consiste en hacer que estas operaciones reciban el objeto (o parte) como parámetro y que sean, por lo tanto, *métodos de clase*, es decir, que no estén asociadas a los objetos.

```
// X.hpp
class X {
  ...
  private: struct impl;
           impl* repr;
           static impl* copia_x(impl* p);
           static void destruye_x(impl* p);
};

// X.cpp
X::X(const X& origen) {
   repr = copia_x(origen.repr);
}

X::~X() {
   destruye_x(repr);
}

X& X::operator=(const X& origen) {
   if (this != &origen) { // si this == &origen estamos asignando un
                          // objeto a si mismo y no hemos de hacer nada
      impl* aux = copia_x(origen.repr);
      destruye_x(repr);
```



```
      repr = aux;
   }
   return *this;
}
```

El puntero `this` apunta al objeto cuyo método se ha invocado. Si hacemos `a = b;` se invoca el método `operator=` del objeto `a` y el parámetro es `b`. Dentro de la definición de `operator=` el puntero `this` apunta a `a`. Puesto que se retorna una referencia, el retorno del método no involucra una costosa copia. La asignación real se produce una vez hemos comprobado que no estamos autoasignando. Podríamos sentirnos tentados de escribir

```
...
if (this != &origen) {
   destruye_x(repr);
   repr = copia_x(origen.repr);
}
...
```

pero el objeto de la parte izquierda de la asignación podría quedar destruido irremisiblemente y producirse un fallo (p.e. falta de memoria dinámica) durante la copia. Por otra parte, `copia_x` debe implementarse de manera que no se produzcan *memory leaks* si no hubiese memoria suficente para la copia (véase la sección 4).

En determinadas circunstancias se definen el constructor por copia o la asignación como métodos privados, para evitar realizar copias costosas de manera inadvertida (por ejemplo, un paso de parámetros o un retorno de resultados). Entonces habrán de realizarse todos los pasos de parámetro o retornos por referencia o referencia constante.

4 Gestión de errores

La gestión de una situación anómala[3] en un programa tiene tres componentes o fases: detección, propagación y tratamiento. Una vez detectado un error en un punto de un programa se ha de indicar la presencia del mismo, y dicha información debe llegar hasta el punto (propagación) en que se procederá a actuar (tratamiento). Existen muchas alternativas para realizar la gestión de errores, pero el mecanismo de *excepciones* de C++ ofrece numerosas ventajas y en general será preferible a otras posibilidades. En particular, simplifica notablemente la propagación, y con ello permite un código más "limpio", en el que se separa de manera natural el tratamiento de las situaciones anómalas (errores) del tratamiento habitual.

El diseño de la gestión de errores debe realizarse poniendo tanto cuidado en ella como en el tratamiento normal. En general, un diseño adecuado consiste en

[3] La llamaremos *error* en este sección; observad que el significado de la palabra error no coincide aquí con el que se le da en la sección 6.

1. Asumir que las precondiciones de una determinada función o método público que estamos implementando pueden no cumplirse y detectar cualquier error en ese sentido, salvo que, por razones de eficiencia, convenga no hacer la comprobación correspondiente.
2. Utilizar otras funciones y métodos de la misma o distintas clases asegurándose que se verifican sus respectivas precondiciones y no provocar ningún error, en la medida de lo posible[4]. No incluir código para propagar o tratar errores que no se van a producir, ya que hemos hecho un uso correcto de las funciones, métodos, clases, etc.

```
// Pre: la pila no esta vacia
void pila::desapilar(void) {
   nodo* n = cim;
   cim = cim -> sig; // no robusto: confia en que la precond. se cumple
   delete n;
}

int suma(const pila& p) {
   pila aux = p;
   int s = 0;
   while (!aux.vacia()) {
      try {                  // comprobacion de errores redundante!
         s += aux.cima();
      } catch(PilaVacia) {
         ...
      }
      try {                  // aqui tambien
         aux.desapilar();
      } catch(PilaVacia) {
         ...
      }
   }
}
```

Una excepción es un objeto que puede ser *lanzado*, para ser más tarde *capturado* y tratado. Para lanzar una excepción se pone `throw` seguido del constructor del objeto, p.e.

```
if (cim == NULL)
   throw PilaVacia(); // se lanza un objeto de la clase PilaVacia
...
if (n > 0)
   throw NumElemsIncorrecto(n); // se lanza un objeto de la clase
                                // NumElemsIncorrecto; el valor n
                                // es un parámetro del constructor
```

[4]Hay errores que no pueden ser anticipados o verificados a priori y habrá que tomar precauciones para el caso en que se produjesen.

El flujo de ejecución se interrumpe tan pronto se lanza una excepción. La excepción “salta” inmediatamente al final de la función donde se produce, de ahí al final de la función que hizo la llamada, de ahí al final de la función de que llamó a esta otra, y así sucesivamente, deshaciendo la secuencia de llamadas hasta que se encuentra el código dispuesto a capturar y tratar la excepción. Cada vez que se deshace una llamada se invocan a los destructores apropiados (se destruyen todos los objetos locales). En rigor, la excepción va “saltando” por los sucesivos finales de bloque, desde el más interno hacia el más externo. Por ejemplo, una excepción lanzada dentro de un bloque `try` salta al final de ese bloque (y entonces la podemos capturar).

La secuencia de “saltos” termina cuando la excepción llega a un bloque `try`. Entonces se comprueba inmediatamente si alguna de las cláusulas `catch` captura objetos de ese tipo o no. Si ningún `catch` captura a la excepción, ésta continúa saltando. Si algún `catch` captura a la excepción se le aplica el código que viene a continuación del `catch`. Después de un `catch` viene una lista de tipos, entre paréntesis. Si una excepción es de uno de los tipos mencionados en la lista entonces es capturada. Junto al tipo puede darse un nombre y entonces a la excepción capturada se le da ese nombre en el código que trata la excepción. Se siguen los mismos convenios que al escribir las listas de parámetros formales de una función. Después de un bloque `try` pueden venir varios `catch`’s que funcionan de manera similar a una alternativa múltiple: primero se comprueba si la excepción pertenece al tipo indicado en la primera claúsula `catch`, sino al de la segunda, etc.

```
class No2Grado {};  // nos sirven los metodos de oficio
class NoSolReal{}; // para estas dos clases!

// resuelve ec. 2o grado con coeficientes enteros
void sol_2_grado(int a, int b, int c,
                 double& r1, double& r2) {
   if (a == 0) throw No2Grado();
   int det = b * b - 4 * a * c;
   if (det < 0) throw NoSolReal();
   double arrel = sqrt(det);
   r1 = 0.5 * (-b + arrel) / a;
   r2 = 0.5 * (-b - arrel) / a;
}

int main() {
   ...
   try {
      sol_2_grado(a, b, c, r1, r2);
      cout << "raiz 1 = " << r1 << endl;
      cout << "raiz 2 = " << r2 << endl;
   }
   catch(No2Grado) {
      cerr << "La ec. no es de 2o. grado!" << endl;
   }
```

```
   catch(NoSolReal) {
      cerr << "La ec. no tiene raices reales" << endl;
   }
}
```

El estándar ANSI de C++ permite, y recomienda, el uso de especificaciones de excepciones. Éstas permiten incluir en la cabecera de una función o método cuáles son los tipos de las excepciones que la función puede lanzar o propagar.

```
void sol_2_grado(float a, float b, float c,
                float& r1, float& r2) throw(No2Grado, NoSolReal);
```

Si la lista de excepciones en una especificación es vacía (`throw()`) entonces significa que la función no lanza o propaga ninguna excepción. La ausencia de una especificación de excepciones significa que la función en cuestión puede lanzar o propagar *cualquier* excepción. Este convenio puede parecer un poco extraño pero obedece a la compatibilidad con las versiones anteriores de C++.

4.1 Errores en constructores y copias

Un convenio general que se aplica en todas las prácticas de EDA es que si se produce un error durante la ejecución de una función o método entonces los parámetros de entrada/salida o el objeto al que se le aplica deben permanecer inalterados, es decir, conservar el estado que tenían antes de iniciarse la ejecución de la operación donde se ha producido el error. Toda la memoria dinámica reclamada y obtenida durante la ejecución del método donde se produce el error debe ser retornada, ya que finalmente no va a ser empleada. Debe evitarse el uso de copias para dar respuesta a este requerimiento en la medida de lo posible ya que éstas pueden provocar errores a su vez y consumen bastante recursos (de tiempo y memoria). Obviamente conviene detectar los errores lo antes posible, antes de comenzar a realizar modificaciones sobre estructuras de datos y si ello no es posible se habrá de mantener información que permita deshacer las modificaciones realizadas. Por ejemplo, si una operación inserta n nodos en un punto intermedio de una lista enlazada y no hubiese memoria suficiente para crearlos, pero dicho problema se produce cuando ya se han insertado algunos, entonces habrá que eliminar los nodos recién insertados (para lo cual habremos tomado la precaución de tener un apuntador al primero de los nodos insertados). Por ejemplo,

```
// pila.rep
struct nodo {
   nodo* sig;
   elem info;
};
nodo* cim;

static nodo* copia_pila(nodo* p) throw(error);
```



```
static void destruye_pila(nodo* p) throw();

// pila.cpp
#include <eda/mem_din>
...
pila::nodo* pila::copia_pila(nodo* p) throw(error) {
   if (p == NULL) return p;
   nodo* aux = new nodo;
   aux -> info = p -> info;
   try {
      aux -> sig = copia_pila(p -> sig);
   } catch(error) {
      delete aux;
      throw;
   }
   return aux;
}

pila& pila::operator=(const pila& P) throw(error) {
   if (this != &P) {
      nodo* aux = copia_pila(P.cim);
      destruye_pila(cim);
      cim = aux;
   }
   return *this;
}
```

En el código de `copia_pila` vemos que si durante la copia recursiva de `p -> sig` se produce un error hemos de suponer que todos los nodos que se hayan podido crear se destruyen. Para que esto se cumpla efectivamente, hay que encerrar la llamada recursiva a `copia_pila` en un bloque `try` y si hubo problemas entonces hay que destruir también el nodo creado al que apunta `aux`. Luego se relanza el error de modo que la llamada recursiva previa detecta la excepción, destruye el nodo y la relanza, etc. deshaciendo la secuencia de invocaciones. En otro caso nos encontraríamos con un *memory leak*. Observad que para relanzar una excepción se pone `throw;` y no se explicita cuál es el objeto lanzado. El objeto relanzado es el mismo que fue capturado por el `catch`. Obviamente, capturar una excepción con el único propósito de relanzarla es absurdo: basta dejar que se propague. El método `operator=` deja que se propague el error que se produce en `copia_pila`; aunque no captura ni lanza excepciones sí las propaga, por eso pone `throw(error)` en la cabecera del método.

```
try {
   ...
} catch(error) {
   throw;              // absurdo! no hace falta capturar la excepcion
}                      // para acto seguido relanzarla!
```

El siguiente ejemplo muestra un caso más complicado, pero que se resuelve aplicando ideas parecidas:

```
// arb_bin.rep
struct nodo {
   nodo* hizq;
   nodo* hder;
   elem info;
};
nodo* raiz;

static nodo* copia_ab(nodo* t) throw(error);
static void destruye_ab(nodo* t) throw();
...

// arb_bin.cpp
#include <eda/mem_din>
...
arb_bin::nodo* arb_bin::copia_ab(nodo* t) throw(error) {
   if (t == NULL) return t;
   nodo* aux = new nodo;
   aux -> info = t -> info;
   try {
      aux -> hizq = copia_ab(t -> hizq);
   }
   catch(error) { delete aux; throw; }
   try {
      aux -> hder = copia_ab(t -> hder);
   }
   catch(error) { destruye_ab(aux -> hizq); delete aux; throw; }
   return aux;
}

arb_bin& arb_bin::operator=(const arb_bin& T) throw(error) {
   if (this != &T) {
      nodo* aux = copia_ab(T.raiz);
      destruye_ab(raiz);
      raiz = aux;
   }
   return *this;
}
```

En operaciones que añaden un nuevo elemento a una estructura de datos y para ello reclaman memoria dinámica, no hace falta ningún tratamiento especial ya que si `new` lanza una excepción, el nuevo elemento no se creará ni la estructura de datos original se modificará.


```
void pila::apilar(int x) throw(error) {
   nodo* p = new nodo; // no se hace nada mas si 'new' falla y lanza
                       // una excepcion de tipo 'error'
   p -> info = x;
   p -> sig = cim;
   cim = p;
}
```

4.2 El módulo error

Una de las clases que se emplean en todas las prácticas de EDA es la clase error. A fin de simplicar la gestión de errores, todos los métodos y funciones que aparecen en la práctica y que pueden provocar una excepción lanzan errores (excepto las clases estándar como list, vector o string; éstas se tratan aparte). Un objeto de la clase error es una tupla con dos strings, el nombre del módulo o clase y el mensaje de error, y un entero, el código de error. La clase ofrece las operaciones `modulo`, `mensaje` y `codigo` para consultar cada uno de estos campos por separado. Además ofrece un método `print` y la sobrecarga del operador « para facilitar la entrada/salida. Obviamente, el uso de la clase error debe estar exento de errores; de otro modo entraríamos en una cadena sin fin.

La captura y tratamiento propiamente dicho de errores se hará siempre en el módulo principal, en aquellas prácticas donde se pida. Las clases básicamente no tratarán los errores, sólo los lanzarán o propagarán[5]. En las prácticas siempre se utilizará el convenio de imprimir la información de los errores por el canal estándar de salida (`cout`) y no el de error (`cerr`).

```
// pila.cpp
...
void desapilar(void) throw(error) {
   if (cim == NULL)
      throw error("pila", PilaVacia, "pila vacia");
      // "pila" es el nombre de la clase, PilaVacia es una constante
      // entera correspondiente al codigo y
      // "pila vacia" el mensaje de error
   ...
}

// main.cpp
int main(void) {
   ...
   try {
      ...
      p.desapilar();
      ...
```

[5] Hay un tratamiento de errores que sí se ha de hacer que es la liberación de memoria dinámica que a veces se ha de realizar cuando se produce un error de memoria dinámica.

```
   } catch(error e) {
      cout << e;
   } catch(...) {
      cout << "error desconocido" << endl;
   }
   ...
}
```

A fin de no utilizar directamente valores de tipo string o de tipo entero es interesante emplear constantes simbólicas de clase para gestionar los errores de cada clase:

```
// pila.hpp
#include <string>
#include <eda/error>

class pila {
  public:
     // constantes simbolicas para la gestion de errores
     static const char nom_mod[] = "pila";

     static const int PilaVacia = 31;
     static const int PilaLlena = 32;

     static const char Msg_PilaVacia[] = "Pila vacia";
     static const char Msg_PilaLlena[] = "Pila llena";
     ...
     pila(int max_elems = 10);
     ...
};


// pila.cpp
#include "pila.hpp"
...
void pila::apilar(int x) throw(error) {
   if (cim == max_elems)
      throw error(nom_mod, PilaLlena, Msg_PilaLlena);
   ...
}
```

Ciertas versiones del compilador de C++ sólo admiten la inicialización de constantes estáticas de clase con tipos integrales o punteros. Por esta razón, los mensajes de error o el nombre del módulo (`nom_mod`) son del tipo `const char []` ($\equiv$ `const char*`) y no, como podría parecer más natural, del tipo `string`. Como muestra el ejemplo, los arrays de caracteres se inicializan mediante cadenas de caracteres (literales) entrecomilladas.

Los códigos y mensajes de error de cada clase o módulo son los que se definen en la documentación específica de la práctica, en los apartados *Códigos y mensajes de error* correspondientes.

Un convenio especial que se usará en las prácticas de la asignatura es el relativo al comportamiento en caso de que se produzcan varios errores simultáneos. La regla es simple: si en una invocación concreta de una función se producen varios errores simultáneamente, la excepción lanzada o propagada ha de ser aquélla cuyo código de error sea menor. Para conseguir cumplir esta regla, basta pensar en qué orden deben hacerse las comprobaciones e invocaciones de otras funciones. Supongamos que la función `f` lanza un error de código 27 si su primer parámetro `x` es negativo o provoca (y propaga) un error de código 80 al usar la función `g` si el parámetro `x` es negativo e impar:

```
void f(int x, int y) throw(error) {
   if (x < 0) throw error(..., 27, ...);
   g(x); // g *no* puede lanzar el error 80, x *no* es negativo
}
```

En cambio, si la función `g` lanzase el error de código 12 cuando `x` es negativo e impar habríamos de poner

```
void f(int x, int y) throw(error) {
   g(x); // g puede lanzar el error 12 si x < 0 y x es impar
   if (x < 0)
      throw error(..., 27, ...); // si x < 0 y x es par
                                 // entonces g no falla, pero
                                 // f si ha de lanzar error
}
```

5 Programación genérica: *templates* e iteradores

Se entiende por *programación genérica* un estilo de programación en la que los algoritmos son lo más independientes posible de los tipos de datos sobre los que operan. La programación genérica en C++ tiene su fundamento en las *plantillas* (*templates*) y en la explotación sistemática del concepto de *iterador*. La aplicación más visible de estas técnicas es la potente *Standard Template Library* (STL), presentada muy brevemente en una de las sesiones finales de laboratorio de la asignatura.

5.1 *Templates*

Un *template* permite escribir una función o una clase en la que intervienen tipos no especificados, que al ser usada se instanciará con los tipos adecuados. Por ejemplo, si queremos implementar un TAD genérico de pilas de elementos definiremos la clase `Pila<Elem>`. Aquí `Elem` es el parámetro "formal" de la clase genérica y después se instanciará con el tipo que necesitemos en cada momento.

```
#include "pila.hpp"   // define la clase generica Pila<Elem>
...
Pila<char> p;       // p es una pila de caracteres
Pila<string> q;     // q es una pila de strings
Pila<nodo*> r;      // r es una pila de apuntadores a nodos
Pila<Pila<int> > s; // s es una pila de pilas de enteros
                    // el espacio entre '<int>' y '>' es imprescindible!
```

Definir una clase genérica mediante *templates* es tan simple como definir cualquier otra clase, aunque la sintaxis es algo más engorrosa.

```
// pila.hpp

template <class Elem>
class Pila {
  public:  Pila();
           ~Pila();
           ...
           void apilar(const Elem& x);
           ...
  private: struct nodo { // en realidad es nodo<Elem>
              nodo* sig;
              Elem info;
           };
           nodo* cima;
           int nr_elems;
}
```

La primera línea (`template ...`) indica que en la declaración de la clase cada aparición de `Elem` es en realidad un parámetro formal que será subtituído por un tipo real al instanciarse la clase.

Por lo demás no hay gran diferencia. Puesto que el tipo `Elem` es arbitrario se evita usar parámetros o retornos por valor que podrían ser muy costosos si `Elem` no es un tipo simple y se usan en su lugar pasos o retornos por referencia constante.

La implementación de una clase genérica sigue la misma pauta: hay que anteponer la declaración `template ...` en la definición de cada método:

```
template <class Elem>
Pila<Elem>::Pila() {
   cima = NULL;
   nr_elems = 0;
}

template <class Elem>
```



```
void Pila<Elem>::apilar(const Elem& x) {
   nodo* p = new nodo;
   p -> sig = cima;
   p -> info = x;
   cima = p;
   ++nr_elems;
}

...
```

Hay dos puntos interesantes a comentar. Por un lado, el nombre de la clase genérica no es `Pila`, sino `Pila<Elem>`. Por eso hay que definir `Pila<Elem>::apilar` y no `Pila::apilar`. Una vez hemos dado la “pista” al compilador de que estamos definiendo un método de la clase `Pila<Elem>` ya podemos omitir en lo sucesivo `<Elem>` y escribir, por ejemplo,

```
template <class Elem>
Pila<Elem>& Pila<Elem>::operator=(const Pila& p) {
   ...
}
```

en vez de

```
template <class Elem>
Pila<Elem>& Pila<Elem>::operator=(const Pila<Elem>& p) {
   ...
}
```

Por otro lado, hay que estar pendiente de los supuestos que se hacen sobre el tipo `Elem`. Por ejemplo, la constructora por defecto (de oficio) para la clase `nodo` llamará a la constructora por defecto de `Elem` para el atributo `info`. Así pues las clases con las que instanciemos `Elem` deben tener constructora por defecto. Recordad que si definimos una constructora con parámetros para una clase, entonces no habrá constructora por defecto para esa clase. Análogamente, al destruir un nodo se invocará la destructora de `Elem` y por tanto es necesario que exista. Finalmente, en la asignación `p -> info = x;` que se hace en `apilar`, estamos usando el operador de asignación entre `Elem`s, así que éste también debe estar definido. En otros casos será necesario que se haya definido la igualdad, los operadores de comparación, el constructor por copia, etc.

Hay otras muchas aplicaciones y características de los *templates* que no presentaremos en este documento ya que no se utilizan en la asignatura y por razones de espacio.

Un problema del mecanismo de *templates* es que actualmente no hay ningún compilador de C++ que admita su compilación separada, por lo cual el código que implementa una clase o función genérica debe estar en la misma unidad de compilación que el código que instancia y usa el código genérico.

A fin de “simular” el modelo de separación de especificación e implementación que hemos venido usando con las clases no genéricas, aquí adoptaremos el convenio siguiente: el módulo que usa a la clase genérica incluye al fichero `.hpp` correspondiente; el fichero `.hpp` incluye al fichero que contiene la implementación de la clase y/o funciones genéricas. Para indicar que el fichero que contiene la implementación no se ha de compilar por separado y que usa *templates*, le daremos la extensión `.t`, en vez de la habitual `.cpp`.

Así nuestra clase `Pila<Elem>` se organizaría del siguiente modo:

```
// programa o modulo que usa Pila<Elem>

#include "pila.hpp"
...
Pila<int> p;
...

// pila.hpp

template <class Elem>
class Pila {
  public : Pila();
           ~Pila();
           ...
  private:
           ...
};
#include "pila.t"     // incluye la implementacion

// pila.t

template <class Elem>
Pila<Elem>::Pila() {
   cima = NULL;
   nr_elems = 0;
}
...
```

5.2 Iteradores

Con frecuencia tenemos que manejar estructuras de datos que son colecciones de objetos o elementos. A este tipo de estructuras de datos se les llama habitualmente *contenedores* (*containers*). Ejemplos bien conocidos son las pilas, las colas, las listas, los conjuntos, los diccionarios, las colas de prioridad, etc. A menudo necesitamos recorrer los elementos de uno de estos contenedores. Una posibilidad es dotar a la clase correspondiente de un punto de interés y operaciones que nos permitan desplazar el punto de interés y consultar el elemento "bajo" el punto de interés. Otra solución mucho más potente y flexible consiste en definir una o más clases auxiliares de *iteradores* asociados a la clase contenedora.

Un iterador abstrae la noción de índice o puntero a un elemento. Se parece mucho a un punto de interés, pero mientras que el punto de interés está bajo el control de la propia clase contenedora y sólo puede haber uno, los iteradores son externos a la clase contenedora y podemos crear tantos como sea necesario. Esta flexibilidad conlleva algunas desventajas. Por ejemplo, si hay dos iteradores sobre un mismo elemento de una colección y empleamos uno de ellos para eliminar el elemento, es responsabilidad del programador deshacerse del otro iterador, puesto que ya no "apunta" a un elemento existente.

Los iteradores son también una buena solución a problemas de eficiencia, sin tener que violar la ocultación de tipos. Por ejemplo, una función de búsqueda en una clase puede devolvernos un iterador al elemento buscado o un iterador especial para indicar que el elemento no estaba presente en la colección. Si después hay que volver a consultar la información de ese elemento o modificarla no será necesaria una nueva búsqueda ya que tenemos el "acceso directo" a través del iterador. Pero al mismo tiempo con un iterador sólo podremos llevar a cabo aquello que esté permitido y no modificar de manera peligrosa la estructura de datos sobre la que itera.

Veamos un ejemplo sencillo de iteradores y su uso. Supongamos que tenemos una clase `List<Elem>` que define una clase auxiliar (anidada) `List<Elem>::iterList`. El código para realizar el recorrido sobre una `List<Elem>` e imprimir todos sus elementos podría ser de la siguiente forma:

```
List<string> L;
...
List<string>::iterList it = L.begin();
while (it != L.end()) {
   cout << *it << endl; // trata el elemento al que apunta 'it'
   ++it;                // avanzar
}
```

La clase `List<Elem>` proporciona dos métodos, `begin` y `end`, que nos devuelven un iterador al primer elemento de la lista y un iterador fictício, respectivamente. El iterador `it` se inicializa para apuntar al primero de la lista (si `L` estuviera vacía entonces `it == L.end()`). El operador `*` se sobrecarga para acceder al elemento apuntado por el iterador y el operador de preincremento `++` para realizar el avance. Cuando se han recorrido todos los elementos, el siguiente avance hace que `it == L.end()` y salimos del bucle. Este es el estilo adoptado en la *Standard Template Library* (STL) y todas sus clases contenedoras están dotadas de clases auxiliares de iteradores que funcionan de manera parecida a la mostrada en este ejemplo.

Una opción diferente consiste en tener en la clase `iterList` una constructora que recibe como parámetro la estructura de datos sobre la cual iterar y nos devuelve un iterador al principio (como `begin`). Para detectar el final del recorrido se sobrecarga `operator bool` de tal modo que devuelva cierto si el iterador es válido y falso en caso contrario, es decir, si nos hemos "salido" de la colección. El ejemplo anterior podría escribirse entonces así:

```
List<string> L;
...
List<string>::iterList it(L);
```

```
while (it) { // invoca a operator bool; si 'it' es valido entonces
             // equivale a 'cierto' y se entra en el bucle
   cout << *it << endl; // trata el elemento al que apunta 'it'
   ++it;                // avanzar
}
```

Este otro estilo se ha adoptado en algunas otras librerías de C++, entre las cuales podemos mencionar LEDA. En realidad las diferencias entre uno y otro estilo son sólo de sintaxis, pero no de concepto.

Una característica típica de las clases iteradoras es que, por razones de eficiencia, necesitan conocer la representación privada de la clase contenedora sobre la que iteran. Declarando la clase iteradora como miembro público (declaración anidada) de la clase contenedora, dicho acceso a la parte privada habría de quedar garantizado. Pero algunos compiladores (p.e. `gcc-2.95.2`) todavía no son capaces de procesar correctamente esta situación, por lo que hay que indicar que la clase iteradora es amiga (*friend*). Por otro lado es habitual que la clase iteradora "conceda" permiso a la clase contenedora declarándola *amiga* (*friend*) para poder implementar eficientemente métodos como `begin()` o `end()`, que pertenecen a la clase contenedora y en principio no tendrían acceso a la parte privada de la clase iteradora. Todo esto supone una violación de la ocultación de tipos, pero es necesaria dada la estrecha colaboración que han de tener estas clases, que en cierto modo, pueden verse como una sola. Por otro lado si los métodos de la clase contenedora no devuelven ni reciben iteradores como parámetros no hace falta ni conviene que la clase contenedora sea amiga de la clase iteradora.

A fin de que la clase iteradora pueda hacer referencia a la representación de la clase contenedora, es común definir la parte privada de la clase contenedora antes de la definición de la clase iteradora.

```
template <class Elem> {
class List {
  private:
     ...
  public:
     friend class iterList { // la clase iterList esta anidada en List,
        private:             // es publica ; la declaracion 'friend' es
           ...               // necesaria en algunos compiladores
        public:
           friend class List; // List es amiga de iterList
              ...
     }
     ...
}
```


5.3 Un ejemplo

En este ejemplo vamos a mostrar algunos puntos significativos de la implementación de una clase genérica List, equipada con iteradores de recorrido hacia adelante, es decir, con avance al siguiente.

Puesto que en la definición de los iteradores se precisa conocer la representación de la clase List, la parte privada la ponemos al principio (en general, hemos puesto la parte pública al comienzo de las declaraciones de la clases). Por otro lado algunas de las operaciones de la clase List reciben iteradores como parámetros o devuelven iteradores, lo que hace necesaria una declaración avanzada de la clase iter.

Comenzamos con las cuatro operaciones básicas: constructora por defecto, por copia, destructora y asignación:

```
#include <eda/error>

template <typename Elem>
class List {
  private:
     ...
  public:
     List() throw(error);
     List(const List& L) throw(error);
     List& operator= (const List& L) throw(error);
     ~List() throw();
     ...
}
```

Puesto que queremos mantener nuestra clase List lo más sencilla posible, contemplaremos sólo una operación de inserción y una operación de eliminación. La operación `insert` inserta el elemento dado `e` por delante del elemento apuntado por el iterador dado, salvo que el iterador sea `end()`, en cuyo caso el elemento se inserta al final de la lista. Si en la invocación de `insert` no se pasa un iterador, el valor por defecto de `it` es justamente `end()` (ver explicación más abajo sobre los iteradores `begin()` y `end()`). La operación `remove` elimina el elemento al que apunta el iterador dado y hace que el iterador `it` no apunte a ningún elemento de la lista, i.e., le da el valor `end()`. Además tendremos dos consultoras que nos dan el número de elementos de la lista y determinan si ésta es vacía o no, respectivamente.

```
...
iter insert(const Elem& e, const iter& it = end()) throw(error);
void remove(iter& it) throw(error);

int size() const throw();
bool is_empty() const  throw();
...
```

Necesitamos además una función `begin()` que nos devuelva un iterador al principio de la lista; resolvemos el problema de determinar cuándo un iterador ha llegado al final de la lista con una función `end()` que nos devuelve un iterador "ficticio" que apunta uno más allá de la lista, al estilo de la STL:

```
...
iter begin() const;
iter end() const;
...
```

En la clase iter, anidada dentro de la clase List, y que declararemos amiga de List, tendremos una constructora `iter()` que nos devuelve un iterador que no está asociado a ninguna lista ni a ningún elemento. Sobrecargando los operadores de pre- y postincremento tendremos la operación de avance y con el operador '*' el acceso al `Elem` al que apunta un iterador. Además necesitamos sobrecargar los operadores de comparación (== y !=).

```
#include <eda/error>

template <typename Elem>
class List {
  private:
     ...
  public:
     ...
     friend class iter {
        private:
           ...
        public:
           friend class List;
           iter();
           ...
           const Elem& operator*() const throw(error);
           iter& operator++ () throw(error);
           iter& operator++ (int) throw(error);
           bool operator==(const iter& it) const;
           bool operator!=(const iter& it) const;
     };
};
```

Para la representación de la lista, usaremos una lista doblemente enlazada dinámica (para permitir la inserción y borrado eficiente). Cada iterador constará de un puntero a un nodo de la lista, y de un puntero a la propia lista. Este segundo puntero nos permitirá comprobar que cuando se usa un iterador para actuar sobre una lista, el iterador está efectivamente asociado a la lista en cuestión y no a otra.


```
#include <eda/error>

template <typename Elem>
class List {
  private:
      // usamos una lista doblemente encadenada
      // sin cierre circular ni fantasmas
      struct nodo {
         nodo* prev;
         nodo* seg;
         Elem info;
      };
      nodo* primero; // puntero al primer nodo
      int nr_elems;
  public:
     ...
     friend class iter {
        private:
           nodo* p;  // puntero al nodo sobre el cual esta el iterador
                     // si p == NULL el iterador es invalido
           List* L;  // puntero a la lista a la que esta asociado
                     // el iterador
        public:
           ...
     };
};
```

Uniendo todas las piezas tendremos completada la especificación y representación de la clase en el fichero `lista.hpp`. Puesto que para representar un iterador no se usa memoria dinámica el constructor por copia, el destructor y la asignación de oficio nos sirven (los atributos de un iterador son punteros a nodos pero los métodos de la clase iter no crean ni destruyen objetos en memoria dinámica!).

Ahora sólo nos queda escribir la implementación en el fichero `lista.t`. A modo de ejemplo se da parte del código necesario.

```
// lista.t

template <class Elem>
List<Elem>::List() throw(error) {
   nr_elems = 0;
   primero = NULL;
}
...
template <class Elem>
```

```
void List<Elem>::remove(iter& it) throw(error) {
   // si el iterador no es valido o esta asociado a otra lista
   // se lanza error
   if (it == end() || it.L != this)
      throw error( ..., ..., ...);
   it.p -> prev -> seg = it.p -> seg;
   it.p -> seg -> prev = it.p -> prev;
   delete it.p;
   it = end();
}
...
template <class Elem>
List<Elem>::iter List<Elem>::begin() const {
   iter it;
   it.p = primero;
   it.L = this;
   return it;
}

template <class Elem>
List<Elem>::iter List<Elem>::end() const {
   iter it;
   it.p = NULL;
   it.L = this;
   return it;
}


template <class Elem>
List<Elem>::iter::iter () : p(NULL), L(NULL) {
}
...
template <class Elem>
const Elem& List<Elem>::iter::operator*() const throw(error) {
   if (p == NULL)
      throw error( ..., ..., ...);
   return p -> info;
}
...
```


6 Juegos de pruebas, *testing* y *debugging*

Ningún *juego de pruebas* o *test* de un programa puede demostrar la corrección de éste. La razón es simple: el número de entradas posibles de cualquier programa no trivial es tan enorme (eventualmente infinito) que no es posible probar todos los casos. La mejor, y en realidad única, "receta" para escribir programas correctos es un buen diseño, una buena metodología y razonar y argumentar la corrección de cada decisión tomada, de cada algoritmo utilizado, de cada representación de datos empleada. En definitiva, hay que emplear razonamientos rigurosos, formales o informales, para demostrar la corrección de cada una de las piezas que componen el programa, en sus distintos niveles de abstracción, y de las interacciones entre ellas. Esta es una de las muchas razones por las que el concepto de TAD es tan importante en la programación moderna: minimiza las interacciones entre componentes y consigue que éstas tengan lugar a través de interfaces definidos con precisión.

Dada la ineficacia de los juegos de prueba para demostrar la corrección, ¿podemos prescindir de ellos? La respuesta es *no*.

Supongamos que hubiéramos demostrado formalmente que cada uno de los algoritmos que intervienen en el programa satisface su especificación, que la implementación de las operaciones de un TAD mantiene el invariante de representación y "conmuta" con la función de abstracción, etc.

Pero los juegos de prueba nos permitirán validar la especificación realizada en primer lugar y comprobar que correspondía a lo que se pretendía. Tal vez no estaba claro del todo que se quería resolver con el programa, y probando su funcionamiento podemos "refinar" nuestras especificaciones. Por esa razón es muy común en los proyectos a gran escala que se implementen *prototipos* antes de comenzar con el desarrollo de los programas reales. O puede suceder que hayamos entendido mal el problema y que nuestra especificación no sea correcta. O que sea incompleta o ambigua en algún punto. También puede suceder que un leve error de codificación cambie la semántica del programa (y por tanto deje de ser correcto), pero que no sea detectado por el compilador. Por ejemplo,

```
bool es_vacio() {
   return nr_elems = 0;
}
```

no devuelve cierto si y sólo si el atributo `nr_elems` es 0, como probablemente era nuestra intención. Por el contrario, devuelve siempre falso y hace que el atributo `nr_elems` tome por valor 0. Es posible que algunos compiladores nos avisen (*warning*) de que la operación está haciendo una conversión implícita de `int` a `bool`. Si hubiéramos indicado que la operación es consultora (agregando `const` en el perfil), el compilador nos daría un error ya que la operación modifica el campo `nr_elems`. Pero hay otros errores similares que no serán nunca detectados por un compilador.

Veamos otro ejemplo:

```
// inicializa la matriz A de dimension n x n con 0's en todas las
```

```
// componentes, excepto la diagonal que se rellena con los reciprocos:
// diag(A) = (1, 1/2, 1/3, 1/4, ..., 1/n)
void rellena_matriz(double A[][], int n) {
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         if (i = j)
            A[i][j] = 1 / (i + 1);
         else
            A[i][j] = 0;
}
```

Sin embargo, si ejecutamos `rellena_matriz` obtenemos una matriz con una diagonal de 0's y los valores iniciales en las restantes componentes (ejercicio: averiguar porqué y cómo corregir el (los) error(es)).

Un juego de pruebas consiste en una descripción de una entrada posible para un programa y la descripción del comportamiento observable del programa para tal entrada (normalmente, la salida que producirá). Si el programa recibe la entrada y su comportamiento es el descrito en el juego de pruebas se dice que el programa *ha pasado* ese juego de pruebas. En caso contrario, se dice que ha fallado, fracasado o que no lo ha pasado.

Por lo general, lo que debemos hacer es preparar una entrada en un fichero (p.e., con extensión `.in`) y la salida esperada en otro fichero (p.e., con extensión `.out`). Si el programa que queremos testear es `prog` entonces podemos escribir:

```
prog < test_prog.in | diff - test_prog.out
```

que ejecuta `prog` con entrada `test_prog.in` y compara (`diff`) la salida que produce el programa (-) con la esperada (`test_prog.out`). Si queremos probar un cierto módulo para realizar lo que se denomina un juego de pruebas *unitario*, habremos de crear un pequeño programa que nos permita comprobar las diferentes operaciones del módulo. A este tipo de programas se les llama convencionalmente *drivers*.

No os deis por satisfechos si vuestra implementación de un módulo o clase pasa varios juegos de pruebas; razonad sobre la corrección de vuestra implementación y no penséis que los juegos de pruebas son un buen sustituto de una cuidada argumentación que demuestre que la implementación efectivamente satisface la especificación. Una buena garantía para que esto suceda es emplear una buena metodología de programación y evitar la tentación de ponerse a escribir código sin un diseño previo o ir haciendo correcciones sobre la marcha a medida que se detectan problemas, programando por "ensayo y error".

Algunas recomendaciones útiles:

1. "Ejecutar" pruebas sobre el propio algoritmo, en papel. Testear el comportamiento del algoritmo en casos extremos. Por ejemplo, si un algoritmo ha de hallar un elemento en una lista, deberíamos comprobar qué sucede si la lista está vacía, si la lista contiene un solo elemento, si el elemento no está presente, si es el primero y si es el último.

2. Verificar mediante *chivatos* o usando `assert` si se cumplen las propiedades esperadas antes y después de un cierto punto. En los casos más difíciles se puede recurrir a un *debugger*, pero habitualmente es más sencillo usar los chivatos y la función `assert`. Por ejemplo:

   ```
   cout << "Antes: i = " << i << "n = " << n << endl;
   // aqui hacemos un cierto calculo con i y con n
   cout << "Despues: i = " << i << "n = " << n << endl;
   ...
   // x nunca tendria que ser < 0 en este punto
   // assert(x >= 0) no hara nada si, efectivamente, x >= 0
   // y abortara la ejecucion del programa si x < 0
   assert(x >= 0);
   ```

   Pero no os descuidéis de eliminar todos los chivatos en el programa que entregáis!

   Para usar un *debugger* hay que compilar con `g++` y el flag `-g`. Por ejemplo,

   ```
   g++ -c -Wall -g -o prog prog.cpp miclase.cpp -leda
   ```

   Existen diversos *debuggers* para todo tipo de plataformas; en el entorno Solaris del LCFIB está `ddd` (también existe una versión para Linux), que ofrece una interfície gráfica bastante sencilla de usar. Para detalles sobre su uso debéis consultar la información que le acompaña o los enlaces en la sección *Recursos en la Red* de las páginas Web de EDA.

3. Testear incrementalmente: no dejéis todos los tests para el final. Si, por ejemplo, un módulo tiene tres funciones `f`, `g`, `h` podemos implementar `f`, definir `g` e `h` trivialmente (con cuerpo vacío) y testear `f`. Luego implementar `g` y testear `g`, así como casos que involucren a `f` y `g`, y así sucesivamente.

4. Preparar juegos de pruebas de caja negra y de caja blanca. Los juegos de pruebas deben construirse en función de la especificación (caja negra) y en función de vuestra implementación (caja blanca). Éstos últimos deben procurar que cada "trozo" de código acabe siendo ejecutado. Los juegos de pruebas que nosotros os suministramos son siempre de caja negra puesto que no conocemos vuestra implementación.

5. Automatizar los tests (*tests regresivos*). Suponed que tenéis un programa o módulo (con su driver) que pasa un cierto número de tests, y en un nuevo test descubrís un error. Después de pensar un rato, examinar el código, etc., localizáis el error y hacéis las modificaciones pertinentes. *Es vital volver a pasar todos los tests*, ya que la corrección de un error puede haber introducido otros. Por ello es sumamente recomendable automatizar el proceso. Uno puede utilizar un *shell script* en Unix/Linux o un fichero `.bat` con este propósito. O usar un lenguaje de *scripting* más sofisticado como AWK, Perl o Tcl. Por ejemplo, podemos editar un fichero llamado `testear` con el siguiente contenido:

   ```
   prog < test_prog1.in | diff - test_prog1.out
   prog < test_prog2.in | diff - test_prog2.out
   ```

```
prog < test_prog3.in | diff - test_prog3.out
prog < test_prog4.in | diff - test_prog4.out
prog < test_prog5.in | diff - test_prog5.out
```

y no tendremos más que permitir que sea ejecutable y usarlo cada vez que queramos pasar los 5 tests de `prog`:

```
% emacs testear              creamos y editamos el shell script
% chmod a+x testear          hacemos que sea ejecutable
% ./testear
```

Si no hay ningún *output* como resultado de ejecutar `testear` es que todo ha ido bien. Otro ejemplo más sofisticado es el siguiente *script*:

```
#! /bin/csh
foreach i (test_prog*.in)
   prog_ant < $i > out1
   prog_nuevo < $i > out2
   if (! cmp -s out1 out2)
      echo $i: DIFIEREN
   end
end
```

que prueba todos los ficheros `test_prog*.in` con la versión antigua (`prog_ant`) y la versión nueva (`prog_nuevo`) de un cierto programa e imprime un mensaje cada vez que hay diferencias entre una y otra versión. Es sumamente importante conservar las versiones previas. Idead algún esquema para llevar el control (p.e., mediante los propios nombres de los ficheros u organizando las distintas versiones en subdirectorios distintos) o emplead una herramienta de control de versiones (por ejemplo, `rcs` o `cvs`).

Para la evaluación de vuestras prácticas se usan dos tipos de juegos de pruebas: los públicos y los privados. Los primeros se ponen a vuestra disposición con antelación a la fecha de entrega. Los privados no se hacen públicos hasta que no ha finalizado el proceso de ejecución automática de las prácticas. Aunque por lo general son similares en espíritu a los públicos, los juegos privados testean "combinaciones" no comprobadas por los juegos públicos.

Los juegos de pruebas (los ficheros `.in` y `.out`) se podrán obtener a través de las páginas Web de la asignatura. En ocasiones también os proporcionaremos los *drivers* para realizar pruebas unitarias. Utilizad los juegos de pruebas públicos como modelo para desarrollar vuestros propios juegos de pruebas; no os contentéis con superar los juegos de pruebas públicos. Intentad pensar todas las situaciones ante las que puede encontrarse el programa, tanto las correctas como las que originan errores. Prestad particular atención a posibles deficiencias de vuestra práctica en la gestión de memoria dinámica o en la gestión de errores.

6.1 La clase gen_driver

La librería `libeda` incluye una clase denominada gen_driver que facilita la creación de *drivers* para que podáis comprobar el funcionamiento de las clases y los módulos de vuestras prácticas.

Para ello habrá de prepararse un programa siguiendo las directrices que se explican con detalle más adelante en esta subsección. Una vez escrito dicho programa, digamos `driver_pract.cpp`, se compilará y montará con

```
g++ -c -ansi -Wall driver_pract.cpp
g++ -o driver driver_pract.o clase1.o clase2.o ... -leda
```

El ejecutable obtenido puede usarse tanto para realizar pruebas de la práctica en modo interactivo, como para hacerlo *off-line* (suministrando un fichero de entrada `.in` y enviando la salida a otro fichero `.out`). En la mayoría de prácticas se os proporcionará el fichero fuente `driver_pract.cpp` junto a los juegos de pruebas; esta subsección describe lo necesario para usar un *driver* de manera efectiva e incluso cómo construir uno propio.

6.1.1 Uso de los *drivers*

Todo *driver* construído mediante gen_driver acepta líneas de comando; cada línea es leída, interpretada y ejecutada. Las líneas leídas se imprimen precedidas del carácter '#' en la salida estándar, siempre que se ha activado el *eco* (por defecto está activado).

Por lo general una línea de comandos tiene la estructura

$$\textit{operación}\ arg_1 \ldots arg_n$$

o bien

$$\textit{id\_obj operación}\ arg_1 \ldots arg_n$$

En el primer caso se escribe en primer lugar el nombre de la operación y a continuación sus argumentos; en el segundo caso se escribe primero el identificador del objeto sobre el que se aplica la operación. El primer formato se utiliza para operaciones que no corresponden a métodos de una clase o cuando se quiere aplicar un método sobre el *objeto en curso*. En todo momento de la ejecución del *driver*, el objeto en curso es el último objeto inicializado o bien el que se haya seleccionado explícitamente con `sel_curr_obj`. En las explicaciones posteriores los dos formatos se presentarán de manera combinada escribiendo

$$[\textit{id\_obj}]\ \textit{operación}\ arg_1 \ldots arg_n$$

donde los corchetes indican que `id_obj` es opcional.

Las operaciones admisibles van a depender, por supuesto, de la práctica concreta que se esté probando, pero hay toda una serie de operaciones comunes que siempre estarán disponibles.

- `#` *`comentario`*: sirve para escribir un comentario en un fichero de entrada y no tiene ningún efecto adicional.
- `%` *`comando`*: ejecuta el comando Unix dado (p.e. `% ls`).
- `init` *`id clase arg(s)`*: crea un objeto `id` de la clase `clase` con los argumentos dados
- `initcopy` *`id1 = id2`*: crea un objeto `id1` inicializado con una copia de `id2`.
- `[`*`id1`*`] =` *`id2`*: copia el objeto `id2` sobre el objeto `id1` o sobre el objeto en curso en su defecto.
- `[`*`id1`*`] destroy`: destruye el objeto `id1` o el objeto en curso en su defecto.
- `apply` *`operación fich`*: aplica la operación sobre cada una de las líneas del fichero dado, donde cada línea se interpreta como una línea de argumentos para la operación; el eco de entrada y la impresión de resultados se desactivan.
- `load` *`fich`*: procesa el fichero `fich` de líneas de comandos; el eco de entrada y la impresión de resultados se desactivan,
- `curr_obj`: imprime el identificador del objeto en curso.
- `list_objects`: imprime una lista de todos los objetos presentes en el sistema, y la clase a la que pertenecen; imprime una flecha para marcar al objeto en curso.
- `sel_curr_obj` *`id1`*: selecciona el objeto `id1` como objeto en curso.
- `echo_input` *`switch`*: activa el eco de las líneas de comandos (`switch=on`) o lo desactiva (*switch*=`off`).
- `echo_output` *`switch`*: activa la impresión de los resultados en la salida (`switch=on`) o la desactiva (*switch*=`off`).
- `print_memory` *`n`*: imprime un informe de estado de la memoria dinámica de nivel `n`.
- `set_memory` *`maxchunks maxbytes`*: fija los parámetros de la memoria dinámica a los valores dados.
- `test_memory` *`clase`*: ejecuta los tests de memoria dinámica para la clase dada.
- `start_t`: pone en marcha el cronómetro interno.
- `stop_t`: detiene el cronómetro interno e imprime el tiempo transcurrido (en segundos) desde que se puso en marcha.
- `applies_to` *`operación`*: imprime el nombre de la clase sobre la que se aplica la operación, imprime un asterisco (`*`) si la operación no es un método de ninguna clase, e imprime `any` si la operación es aplicable sobre objetos de más de un tipo.

- `?` *`operación`*$\|$*`clase`*: imprime una ayuda breve sobre la operación o la clase indicada.
- `help`: imprime una lista todas las operaciones disponibles y proporciona una breve ayuda de cada una.

6.1.2 Construcción de un *driver*

Mostraremos paso a paso cómo crear un *driver* ayudados por un ejemplo sencillo en el que tenemos dos clases `A` y `B`:

```
// A.hpp
class A {
public:
        A();
        A(const A& a);
        A& operator=(const A& a);
        ~A();
        void inserta(const string& x);
        void elimina_menor();
        string menor() const;
        ...
        static const char nom_mod[] = "A";
}

// B.hpp
class B {
public:
        B(int n = 0);
        B(const B& b);
        B& operator=(const B& b);
        ~B();
        string& operator[](int i);
        int size() const;
        ...
        static const char nom_mod[] = "B";
}
```

Adicionalmente supondremos que tenemos un módulo heapsort:

```
// heapsort.hpp

namespace heapsort {
    void sort(B& b);
}
```

Para el correcto funcionamiento del driver es imprescindible que cada clase que se quiera probar defina una constante pública de clase llamada `nom_mod`. Por lo general su valor será el nombre la clase (aunque no es obligado) y ha de coincidir con el que se utilizará en la definición del comando `init`.

Para cada método de las clases (excepto constructoras, destructora y asignación) tenemos que escribir una función de la forma `void f(gen_driver& D)`. La función pedirá los parámetros al `gen_driver`, los convertirá al tipo adecuado, y llamará al método sobre el objeto que corresponda. Finalmente, se imprimirían los resultados cuando proceda. Veamos un primer ejemplo, con los métodos sin parámetros:

```
// driver_pract.cpp
...
void menor(gen_driver& D) { // usamos el mismo nombre para la
                            // funcion que en la clase por comodidad

     A* a = D.object<A>();          // obtiene un puntero al objeto;
                                    // la sintaxis es D.object<T>()
                                    // cuando el objeto es de tipo T

     ostream os = D.get_ostream();  // obtener el ostream para imprimir

     os << a -> menor() << endl;
}

void elimina_menor(gen_driver& D) {
     D.object<A>() -> elimina_menor(); // no hay que imprimir resultado
}

void size(gen_driver& D) { // se puede escribir de manera
                           // más breve que en el ejemplo de 'menor'

     D.get_ostream() << D.object<B>() -> size() << endl;
}
...
```

Si la operación tiene parámetros tales como `strings`, enteros, reales, etc. el tratamiento también es bastante simple. En estos casos resultarán útiles las funciones de conversión del módulo `<eda/util>`. El argumento i-ésimo se obtiene con `D.args(`i`)` si es un valor (siempre lo devuelve como un string, sobre el que posiblemente habría que aplicar una conversión) o con `D.object<T>(`i`)` si es un identificador de un objeto del tipo `T`.

```
// driver_pract.cpp
...
void inserta(gen_driver& D) {
```



```
     string s = D.args(1);
     D.object<A>() -> inserta(s);
}

void acceso(gen_driver& D) {
        // corresponde a string& B::operator[](int i) para consulta
        // Comando: [id] acceso i
     int i = util::toint(D.args(1));
     D.get_ostream() << D.object<B>() -> operator[](i) << endl;
}

void asig(gen_driver& D) {
        // corresponde a string& B::operator[](int i) para asignación
        // Comando: [id] asig i s
     int i = util::toint(D.args(1));

     D.object<B>() -> operator[](i) = D.args(2);
}
...
```

En la operación `sort` del módulo heapsort no hay objeto sobre el que aplicarla pues no es un método. Sin embargo recibe un parámetro de la clase `B` al que accederemos mediante `D.object<B>(`i`)`. Por otro lado, sería una buena idea poder ver el resultado de la ordenación, así que nos definiremos una operación auxiliar para imprimir.

```
// driver_pract.cpp
...
void operator<<(ostream& os, const B& b) { ... }

void sort(gen_driver& D) {
   B* b = D.object<B>(1); // el identificador del objeto es el
                          // primer argumento
   heapsort::sort(*b);

   D.get_ostream() << b;  // usamos la auxiliar para imprimir
}
...
```

La "mayor" complicación que nos queda pendiente es la de escribir la función de inicialización llamada `user_init`. El nombre no puede cambiarse y su perfil ha de respetarse. El comportamiento de `user_init` depende del tipo de objeto a crear (el nombre de la clase se obtiene con `D.args(2)`) y habrá de verificarse la corrección de los parámetros en número y tipo, pues al ser variable y dependiente de las clases específicas a testear no puede hacerlo la clase gen_driver.

```
...
void* user_init(gen_driver& D) {
     string idobj = D.args(1);
     string idclass = D.args(2);

     if (idclass == "A") {
         // el comando ha de ser de la forma init <id> <class>
         // por tanto sólo hay dos argumentos ...
         if (D.nargs() != 2)
            throw error(gen_driver::nom_mod, gen_driver::WrongNumArgs,
                    gen_driver::WrongNumArgsMsg);
         // se crea un nuevo objeto y el puntero se convierte a tipo
         // void* antes de retornarlo
         return static_cast<void*>(new A());
    } else if(idclass == "B") {
         // el comando ha de ser de la forma init <id> <class> [int]
         // por tanto hay dos o tres argumentos ...
         if (D.nargs() != 2 && D.nargs() != 3)
            throw error(gen_driver::nom_mod, gen_driver::WrongNumArgs,
                        gen_driver::WrongNumArgsMsg);
         // si son dos argumentos se usa el parámetro por defecto
         // para construir el objeto de tipo B
         if (D.nargs() == 2)
            return static_cast<void*>(new B());

         int n = util::toint(D.args(3));
         return static_cast<void*>(new B(n));
     }
     // si llegamos aquí estabamos haciendo init con un nombre de clase
     // que no es ni A ni B ...
     throw error(gen_driver::nom_mod, gen_driver::WrongTypeArgs,
                 gen_driver::WrongTypeArgsMsg);
}
...
```

Para terminar hay que escribir la función `main`. Ésta definirá un objeto `D` de la clase gen_driver, "registrará" las funciones que hemos definido y los tipos de objetos que han de existir, y finalmente haremos `D.go()`.

```
#include <string>
#include <iostream>
#include <eda/error>
#include <eda/gen_driver>
#include <eda/util>
...
```



```
#include "A.hpp"
#include "B.hpp"
#include "heapsort.hpp"

void* user_init(gen_driver& D) { ... }

void menor(gen_driver& D) { ... }

...

int main() {
  gen_driver D;

  // registro de funciones
  D.add_call(...);
  ...
  // registro de tipos
  // la sintaxis general es install_type<T>()
  D.install_type<A>();
  D.install_type<B>();

  D.go();
}
```

Si en nuestro sistema necesitamos objetos de tipos estándar entonces deberemos "registrarlos" con `install_std_type`. Por ejemplo,

```
  d.install_std_type<vector<int> >("vector<int>");
  d.install_std_type<string>("string");
```

Se utiliza `install_std_type` y no `install_type` porque en principio no necesitaremos hacer pruebas de memoria dinámica sobre estas clases y además no definen la constante de clase `nom_mod`. El método `install_type` no puede aplicarse sobre clases que no definan esta constante.

El registro de las funciones y métodos se hace mediante `add_call`. Este método tiene la siguiente declaración:

```
  void add_call(const string& fname, DriverFunction f,
                const string& applies_to = "*",
                const string& type_args = "",
                const string& helpmsg = "");
```

El primer string es el nombre de la operación tal como lo reconocerá el *driver*. Por ejemplo, el `operator[]` de la clase `B` tiene dos posibles usos (para consulta y para actualización) y por eso hemos hecho dos funciones de *driver* que podemos activar con los nombres `acceso` y `asig` (o cualesquiera otros que nos gusten). El nombre de una función puede contener cualquier carácter no espaciador, p.e. `acceso`, `dame1`, `++`, ... El segundo parámetro es la función[6] `void f(gen_driver& d)` que hayamos escrito para invocar al método o función. Luego viene el nombre de la clase de objeto sobre el cual se aplica la operación; se usará `*` para indicar que la operación no es un método en una clase y se usará el nombre `any` si el método se aplica sobre varios tipos de objetos. El siguiente parámetro es un string que lista los nombres de tipos de los argumentos, separados por espacios, y en el orden apropiado. Por defecto el string es vacío, lo que corresponde a una operación o método sin argumentos. P.e. para la operación `acceso` tendríamos dos argumentos, el primero de tipo entero y el segundo de tipo string, así que cuando hagamos el `add_call` correspondiente pondremos `"int string"`. Si un parámetro puede ser de varios tipos se pondrá en el lugar correspondiente `any`. El último parámetro de `add_call` es un string con el que podemos asociar un mensaje de ayuda para la operación en cuestión. Si este parámetro es la cadena vacía o se omite, `add_call` genera de manera automática un mensaje de ayuda para la operación.

Para el ejemplo concreto que hemos usado, el registro de funciones es el siguiente:

```
int main() {
  gen_driver D;

  ...

  // registro de funciones
  // tenemos dos formas de llamar a inserta
  D.add_call("inserta", inserta, "A", "string");
  D.add_call("ins", inserta, "A", "string");

  // el nombre de la operación no tiene que coincidir con el de la
  // función
  D.add_call("elimina", elimina_menor, "A");

  D.add_call("menor", menor, "A");
  D.add_call("acceso", acceso, "B", "int");
  D.add_call("asig", asig, "B", "int string");
  D.add_call("size", size, "B");

  // sort no se aplica sobre un objeto; tiene un argumento de tipo B
  D.add_call("sort", sort, "*", "B");
  ...
```

Finalmente, para poder comprobar el correcto funcionamiento de la memoria dinámica sobre una clase dada también será necesario escribir una serie de funciones auxiliares y registrar los tests de memoria dinámica a realizar.

[6]Técnicamente es un puntero a la función.

La clase gen_driver ofrece tres funciones básicas con el fin de realizar los tests: `generic_memtest`, `copyctor_memtest` y `assgn_memtest`. El método `generic_memtest` tiene un primer parámetro que es la función que se usará para construir un objeto de la clase y un string que describe el tipo de test que se quiere realizar. El método idgeneric_memtest se usará típicamente para comprobar el correcto uso de la memoria dinámica en constructores o para inserciones en una estructura de datos.

En el fichero `driver_pract.cpp` escribiremos una o más funciones para construir objetos de la clase a testear (siempre tendrán el perfil `void f(T*&)`) y las correspondientes funciones (denominadas funciones de test de memoria dinámica) que se encargan de hacer la llamada al método `generic_memtest` (ó `copyctor_memtest` ó `assgn_memtest`, según el caso).

Las funciones de test de memoria dinámica siempre tendrán el perfil `bool f(gen_driver&)`.

```
...
void crea_un_A(A*& a) {
   a = new A();
   a -> inserta("aaaa");
   a -> inserta("bbbb");
   ...
}

bool test_inserta_A(gen_driver& D) {
   return D.generic_memtest<A>(crea_un_A, "inserta");
}

void inicializa_un_B_50(B*& b) {
   b = new B(50);
}

bool test_ctor_B_n(gen_driver& D) {
   return D.generic_memtest<B>(inicializa_un_B_50, "ctor(int n)");
}
...
```

Finalmente hay que registrar la función de test de memoria dinámica de manera parecida a cómo se registran las funciones de *driver*. Por comodidad, para crear la función de test de memoria dinámica de la constructora por defecto de la clase A no hace falta escribirla, se puede usar `test_defctor<A>`:

```
...
int main() {
  gen_driver D;
  ...
  D.add_memory_test<A>(test_defctor<A>);
  D.add_memory_test<A>(test_inserta_A);
```

```
  D.add_memory_test<B>(test_defctor<B>);
  D.add_memory_test<B>(test_ctor_B_50);
}
```

Para pasar tests de memoria dinámica sobre la constructora por copia o el operador de asignación se siguen pasos similares a los anteriores, pero las funciones de test de memoria dinámica correspondientes se escribirán en términos de `copyctor_memtest` y `assgn_memtest` en vez de usando `generic_memtest`. El método `copyctor_memtest` recibe como parámetro la función de creación de un objeto, del que luego se intenta crear una copia. El método `assgn_memtest` recibe como parámetros dos funciones para crear los dos objetos entre los que luego se intentará la asignación (del segundo sobre el primero).

Si la función que se quiere usar para construir un objeto es simplemente la constructora por defecto entonces podemos utilizar `_defctor<T>` para la clase T.

```
bool test_copyctor_A(gen_driver& D) {
   return D.copyctor_memtest<A>(crea_un_A);
}

// crea un objeto en el que se han hecho varias inserciones, crea otro
// objeto con la constructora por defecto y luego asigna el segundo al
// primero

bool test_assgn_A(gen_driver& D) {
   return D.assgn_memtest<A>(crea_un_A, _defctor<A>);
}
...
int main() {
  gen_driver D;

  ...
  D.add_memory_test<A>(test_copyctor_A);
  D.add_memory_test<A>(test_assgn_A);
  ...
}
```

7 Estilo de programación y documentación

Un estilo de programación consistente y una documentación clara, concisa y precisa son dos elementos clave de un buen programa. La detección de errores o la modificación de un programa mal documentado y sin una mínima coherencia (nombres de los identificadores, sangrado[7], estructuración de los componentes del programa, etc.) es una tarea poco menos que imposible. A continuación os damos una serie de recomendaciones sobre estos dos aspectos.

[7]"Indentación".

1. Usad nombres descriptivos para constantes, clases y funciones visibles en varios puntos del programa, y nombres breves para las variables locales o los parámetros formales de una función. Por ejemplo, es inapropiado llamar a un método `f` o a una clase `X` (salvo para ilustrar una característica del lenguaje de programación!). Sus identificadores deben describir su propósito y por lo tanto suelen ser largos y estar compuestos por varias palabras: `copia_pila`, `inserta`, `SopaLetras`, ... Por el contrario, para un parámetro formal o una variable local el identificador `n` es adecuado, `npuntos` aceptable y `numero_de_puntos` es un "cañón para matar moscas". Si una variable local se va a usar de modo convencional podemos darle un identificador mínimo: `i`, `j` y `k` se suelen usar para los índices de bucles, `p` y `q` para apuntadores, `s` y `t` para `strings`, etc. Por ejemplo,

   ```
   void inicializa_tabla(elem tabla_elems[], int nr_elems) {
      int indice_elem;
      for (indice_elem = 0; indice_elem < nr_elems; indice_elem++)
         tabla_elems[indice_elem] = indice_elem;
   }
   ```

   no es más comprensible que

   ```
   void inicializa_tabla(elem A[], int n) {
      for (int i = 0; i < n; i++)
         A[i] = i;
   }
   ```

2. "Inventad" un esquema para expresar los identificadores y aplicadlo sistemática y consistentemente. Por ejemplo, suele recomendarse emplear nombres en mayúsculas para constantes (p.e., `ELEM_SIZE`, `MAX_ELEMS`). Un convenio que hemos empleado en este documento y en el material de la práctica es el de combinar mayúsculas y minúsculas para los identificadores de códigos y mensajes de error (p.e., `NoSolReal`, `PilaLlena`). Por lo general, las conectivas (p.e., artículos y preposiciones) se dejan en minúsculas. Algunos programadores utilizan el convenio descrito para todos sus identificadores de funciones, clases, y otros elementos globales: `class Pila { ...}`, `CopiaPila`, `laCima`, `unNodo`, ... Otros prefieren utilizar minúsculas y separar las palabras mediante el símbolo de subrayado: `copia_pila`, `la_cima`, ... Un convenio que también tiene muchos adeptos es el de anteponer el carácter de subrayado a los elementos privados de una clase:

   ```
   template <typename T> class pila {
     ...
     private:
        int _cima;
   ```

```
      T _cont;
      int _max_elems;
};
```

Se recomienda usar verbos en voz activa para los métodos y las funciones, y reservar adjetivos o nombres de la forma `es_...` para funciones o métodos cuyo resultado es un booleano. Conviene pensar los identificadores desde el punto de vista del usuario, no del implementador y tener en cuenta que los métodos se aplican sobre objetos. Por ejemplo, comparad

```
conjunto C;
if (C.pertenece(x)) ...
```

con

```
conjunto C;
if (C.contiene(x)) ...
```

El convenio o esquema de nombres utilizado debe basarse en lo que las entidades representan, no a cómo lo representan. Son desaconsejables por lo tanto los convenios basados en una característica de bajo nivel o ligada a la implementación, como por ejemplo anteponer el prefijo `i_` a las variables y atributos de tipo entero y el prefijo `f_` a las variables y atributos de tipo real (`float`).

Sobre todo, es importante ser consistente: al principio puede ralentizar un poco el trabajo tener que recordar los convenios que hemos elegido; pero luego lo haremos casi sin pensar.

Acabamos este punto con un ejemplo de inconsistencia caricaturizado, pero indicativo de la importancia que tiene este aspecto del estilo:

```
class Pila {
 public:  Pila(int Max_Elementos);
          ~Pila();
          Pila(const Pila& s);
   const Pila& operator=(const Pila& la_pila);
   void Apilar(int elemento);
   int desapila(void);
   int Cima(void);
   bool Vacia(void);
 private:
   struct tnodo {
     int info;
     tnodo* siguiente;
   };
   tnodo* cima;
```



```
   int Nr_Elems_Pila;
   int maxElems;
};
```

3. Un nombre no sólo identifica; conlleva información. Un nombre inadecuado inducirá a confusiones y errores. Por ejemplo, en la siguiente función la variable local `encontrado` significa lo contrario de lo que su nombre indica, y la función retorna lo contrario de lo que su nombre indica. Probablemente se trata de un error inducido por el identificador `encontrado`, pero cambiar `return encontrado` por `return !encontrado` no es una buena solución.

```
bool conjunto::contiene(int x) {
   bool encontrado = true;
   nodo* p = primero;

   while (p != NULL && encontrado) {
      encontrado = p -> info != x;
      p = p -> sig;
   }
   return encontrado;
}
```

4. Sangrad el código y utilizad los paréntesis y espacios en blanco para mejorar la legibilidad del código. La mayoría de editores de texto modernos (p.e., EMACS) incorporan el sangrado automático, de manera que no conlleva demasiado problema. También conviene sustituir, al final, todos los tabuladores (introducidos mediante la tecla TAB o automáticamente por el editor) por espacios en blanco. En otro caso, las impresiones en papel pueden quedar desajustadas. Usad los paréntesis y los espacios en blanco para resolver ambigüedades y facilitar la comprensión de las expresiones. Por ejemplo,

```
bool es_bisiesto(int y) {
return y%4==0 && y%100!=0 || y%400==0;
}
...
for(int i=0;i<n;i++)
bisiesto[i]=es_bisiesto(llista_anys[i]);
...
```

es, para la mayoría de la gente, más difícil de comprender que

```
bool es_bisiesto(int y) {
   return ((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0);
}
...
for (int i = 0; i < n; i++)
   bisiesto[i] = es_bisiesto(llista_anys[i]);
...
```

Otro tanto ocurre con las llaves ({}) de apertura y cierre de bloques. Si un bloque sólo contiene una instrucción no hace falta usarlas, pero puede ser útil para mejorar la legibilidad y evitar errores como en el siguiente ejemplo:

```
if (mes == FEBRERO) {
   correcto = true;
   if (es_bisiesto(any))
      if (dia > 29)
         correcto = false;
   else    // este else NO empareja con el segundo if!!
      if (dia > 28)
         correcto = false;
}
```

Se puede arreglar el problema usando las llaves en el lugar adecuado o mejor aún escribiendo:

```
if (mes == FEBRERO)
    correcto = (dia <= 28) || (dia == 29 && es_bisiesto(any));
```

Sed consistentes en el estilo de sangrado y el uso de las llaves. Algunos estilos de sangrado y uso de las llaves populares son: 1) las llaves se abren y cierran en líneas separadas; 2) la llave se abre en la línea que abre el bloque `for`, `while`, etc., y las llaves de cierre van en líneas separadas; 3) todas las llaves de cierre consecutivas se ponen en la misma línea. Conviene emplear alguno de los estilos usuales ya que son bien soportados por los editores de texto y no resultarán chocantes para otra gente que lea el código.

```
// Ejemplo del estilo 1
for (int j = 0; j < k; j++)
{
   i = j % 2;
   if (i == 0)
   {
      ...
   }
   else
   {
      ...
   }
}

// Ejemplo del estilo 2
for (int j = 0; j < k; j++) {
   i = j % 2;
   if (i == 0) {
```



```
      ...
   }
   else {
      ...
   }
}

// Ejemplo del estilo 3
for (int j = 0; j < k; j++) {
   i = j % 2;
   if (i == 0) {
      ...
   } else {
      ...
} }
```

Una excepción con respecto a las reglas habituales de sangrado son las alternativas múltiples. En C y C++ es típico escribir:

```
if (B1)
   S1
else if (B2)
   S2
...
else if (Bn)
   Sn
else
   Sn+1
```

con todos los `else`'s alineados; esta construcción corresponde a

$$
\begin{array}{l}
[\, B_1 \longrightarrow S_1 \\
[\!]\, B_2 \longrightarrow S_2 \\
\quad\quad \ldots \\
[\!]\, B_n \longrightarrow S_n \\
[\!]\, \neg\,(B_1 \vee \cdots \vee B_n) \longrightarrow S_{n+1} \\
]
\end{array}
$$

cuando las B_i's son mutuamente excluyentes (los `if ... else`'s se evalúan secuencialmente y no hay indeterminismo).

5. Evitad un "flujo" o lógica del programa antinatural y "factorizad" el código común. Por ejemplo, en vez de

```
int s = 0;
nodo* p = primero;
```

```
if (p == NULL) {  // la lista es vacia; no hacemos nada
}
else {
   while (p != NULL) {
      s = s + p -> valor;
      p = p -> sig;
   }
}
return s;
```

deberíamos haber escrito

```
int s = 0;
nodo* p = primero;
while (p != NULL) {
   s += p -> valor;
   p = p -> sig;
}
return s;
```

o bien

```
int s = 0;
for (nodo* p = primero; p != NULL; p = p -> sig)
   s += p -> valor;
return s;
```

Otro ejemplo es la siguiente función para insertar un elemento en un conjunto implementado mediante una lista enlazada ordenada, con fantasma y apuntadores al primero y al último nodo:

```
void conjunto::inserta(const string& x) {
   nodo* q = primero;       // q apunta al fantasma
   while (q -> sig != NULL && q -> sig -> info < x)
      q = q -> sig;
   if (q -> sig == NULL) {
      nodo* p = new nodo;   // el nuevo nodo sera el ultimo
      p -> info = x;
      p -> sig = NULL;
      ultimo = p;
      q -> sig = p;
   } else if (q -> sig -> info == x) { // no se hace nada
   } else {
      nodo* p = new nodo;
      p -> info = x;
```



```
      p -> sig = q -> sig;
      q -> sig = p;
   }
}
```

Hubiera sido mucho mejor “factorizar” la parte común y simplificar la “lógica” de la parte que sigue al bucle de búsqueda:

```
void conjunto::inserta(const string& x) {
   nodo* q = primero; // q apunta al fantasma
   while (q -> sig != NULL && q -> sig -> info < x)
      q = q -> sig;
   if (q -> sig == NULL || q -> sig -> info != x) {
      nodo* p = new nodo;
      p -> info = x;
      p -> sig = q -> sig;
      if (q -> sig == NULL)
         ultimo = p;
      q -> sig = p;
   }
}
```

6. Disminuid la complejidad mediante un uso juicioso de la descomposición funcional. Aprovechad las soluciones a problemas similares mediante una adecuada descomposición funcional. Por ejemplo, en una clase implementada como una lista enlazada es frecuente tener un bucle, como en el ejemplo previo, o el código correspondiente a la inserción en un punto concreto de la lista. Por lo tanto puede ser conveniente tener sendas operaciones privadas e implementar las operaciones públicas usando las privadas:

```
// lista_ord.hpp
class lista_ord {
  public :
     ...
     // inserta x en la lista, en orden
     void inserta(int x);
     ...
};

// lista_ord.rep
...
static void inserta_tras(nodo* p, int x);
nodo* localiza_elem(int x);

// lista_ord.cpp
...
```

```
void lista_ord::inserta(int x) {
   nodo* q = localiza_elem(x);
   if (q -> sig == NULL || q -> sig -> info > x)
      inserta_tras(q, x);
}

// metodo privado
// devuelve un apuntador al ultimo nodo de la lista tal que su info
// es < x; eventualmente al fantasma, si la lista esta vacia o el
// primero de la lista es >= x, o al ultimo nodo, si x es mayor que
// la info de cualquier nodo en la lista.
lista_ord::nodo* lista_ord::localiza_elem(int x) {
   nodo* q = primero;
   while (q -> sig != NULL && q -> sig -> info < x)
      q = q -> sig;
   return q;
}

// metodo privado de clase
// inserta en la lista enlazada un nuevo nodo, con info == x, como
// sucesor del nodo apuntado por p
// Pre: p != NULL
void lista_ord::inserta_tras(nodo* p, int x) {
   nodo* n = new nodo;
   n -> info = x;
   n -> sig = p -> sig;
   p -> sig = n;
}
...
```

Puesto que sólo los métodos de la clase puede utilizar a los métodos privados es adecuado suponer que las precondiciones se cumplirán al ser invocados y evitar una gestión de errores compleja.

En cualquier caso, si la implementación de una función ocupa más de dos tercios de página o se necesita una muy larga explicación para documentarla, entonces es casi seguro que convendría rediseñar el algoritmo o descomponer la implementación en "piezas" más manejables.

Otro aspecto a considerar es el orden en el que definimos las funciones. Existen varias alternativas razonables: todas las funciones privadas en primer lugar y luego las públicas; o justo al revés. Un convenio probablemente mejor es el de situar las operaciones privadas lo más próximas (justo antes o justo después) de la operación u operaciones que las usan.

7. Usad construcciones similares para realizar tareas similares. Si en un punto del programa inicializáis una tabla `A` mediante:


```
for (int i = 0; i < n; i++)
   A[i] = 0;
```

entonces no escribáis el bucle que calcula cuántos elementos no nulos hay en `A` de la siguiente manera:

```
i = 0; nnulos = 0;
while (i <= n - 1) {
   if (A[i] != 0)
      nnulos++;
   i++;
}
```

aunque sea totalmente correcto.

8. No uséis variables globales, excepto cuando sea estrictamente imprescindible. Una variable u objeto global es externo a cualquier función o método. Los atributos de clase son básicamente objetos globales, excepto que el acceso a ellas puede restringirse si se declaran en la parte privada.

```
int nr_elems;              // variable global! evitar su uso
const int MAX_ELEMS = 30;  // constante global, OK

class X {
   ...
   static const int MAX_SIZE = 20; // constante de clase, OK
   static int nr_objetos;          // variable de clase! evitar su uso
   ...
};
```

El problema con los objetos globales es podemos tener efectos laterales en las funciones y métodos, y se rompen los principios de modularidad. En el siguiente ejemplo la función `esta` sólo funciona para el array `A` cuyo tamaño es `n`, y sin embargo, el algoritmo que implementa es igualmente válido para cualquier array:

```
// variables globales
int A[20];
int n;

// retorna cierto si y solo si x esta en A[0..n-1]
bool esta(int x) {
   for (int i = 0; i < n && A[i] != x; i++)
      ;      // el cuerpo del bucle es vacio
   return i < n;
}
```

Toda comunicación entre las funciones y los métodos con su entorno debería producirse a través de sus parámetros o retornos. Observad que para un método de una clase `X` el objeto al cual se aplica el método es un parámetro implícito y por lo tanto no supone una violación de esta regla.

Las llamadas *variables locales estáticas* son otra forma encubierta de romper la modularidad. Una variable de este tipo es una variable local a una función o método, pero retiene su valor entre ejecuciones sucesivas.

Hay casos excepcionales y plenamente justificados para el uso de variables globales o estáticas; p.e., los objetos `cout`, `cin` y `cerr` son objetos globales. Fijaos, no obstante, que solemos definir funciones del tipo `print` o los operadores « y » de modo que reciban un parámetro de tipo `ostream` o `istream` explícito.

Un ejemplo clásico de uso justificado de variables estáticas y globales es un generador de números pseudo-aleatorios: cada número es generado a partir del anterior (excepto la primera vez) y no es adecuado ni cómodo que el usuario tenga que gestionarlo a través de parámetros explícitos:

```
double semilla = 0.0; // variable global
void inicializa_rand(double sm) {
   semilla = sm;
}
double rand() {
   static double x = semilla;
   // la primera vez se inicializa con el valor de la variable global;
   // en lo sucesivo, la variable estatica local x empieza con su
   // valor en la ejecucion previa
   x = funcion_complicada(x);
   return x;
}
```

El lenguaje C++ nos ofrece mecanismos que nos permiten dar una solución más "limpia" a este tipo de situaciones (sólo por mencionar un defecto del ejemplo anterior, observad que nada impediría que cualquier función accediese y modificase la variable `semilla`).

En particular podemos usar variables privadas de clase:

```
class Random {
   public:
     Random(double sm = 0.0) { _x = sm; }
     double rand() { _x = funcion_complicada(_x); return _x; }
   private:
     static double _x; // variable de clase
}
```

Otra excepción a la regla son las variables globales que en realidad se usan como constantes globales, pero que no son declaradas como `const` porque no pueden ser inicializadas en un sólo paso, su valor inicial debe ser calculado algorítmicamente o existe la

necesidad de poder efectuar (muy ocasionalmente) cambios en su valor. Es usual que éstas también se implementen usando variables de clase privadas, para restringir su manipulación e impedir en la medida de lo posible usos incorrectos.

9. Utilizad variables locales y evitad métodos o funciones con largas listas de parámetros. No incluyáis atributos en un objeto o parámetros en una operación innecesarios si su misión es realizable mediante una o más variables locales. Por ejemplo, si una clase lista no necesita la noción de punto de interés entonces es absurdo incluir este tipo de atributo para hacer un recorrido o poner un apuntador como parámetro de una función que hace el recorrido iterativamente:

```
bool lista::contiene(const T& elem) const {
   actual = primero;   // usar aqui var. local, NO un atributo 'actual'!
   while (actual != NULL && actual -> info < x)
      actual = actual -> sig;
   return actual != NULL && actual -> info == x;
}

bool lista::contiene_priv(const T& elem, nodo* p) {
                       // usar var. local, NO un parametro 'p'!
   while (p != NULL && p -> info < x)
      p = p -> sig;
   return p != NULL && p -> info == x;
}

bool lista::contiene_rec(const T& elem, nodo* p) {
// OK; aqui p no es un apuntador para el recorrido, en realidad
// representa a la sublista que queda por explorar
   if (p == NULL) return false;
   if (p -> info >= x) return p -> info == x;
   return contiene_rec(x, p -> sig);
}
```

10. El código debe ser estructurado: cada bloque debe tener un único punto de entrada y un único punto de salida. No uséis `breaks` (salvo en los `switchs`), `continues` o `gotos`. No hagáis `return` desde el interior de un bucle. Usad el esquema de búsqueda cuando sea procedente, no un `return` ó `break` desde el interior de un bucle que hace un recorrido. Tampoco es buen estilo "romper" la iteración modificando la variable de control que recorre la secuencia:

```
for (i = 0; i < n; i++) {
   if (A[i] == x) i = n;    // estilo chapucero!
   ...
}
```

Las únicas desviaciones aceptables respecto a esta regla son las excepciones—que, por definición, rompen inmediatamente el flujo normal de ejecución—, los `switch`s y las composiciones alternativas (no internas a un bucle) típicas de funciones recursivas

```
if (i > 1)
   result = x;
else if (i == 1)
   result = y;
else            // if (i < 1)
   result = z;
return result;
```

puede escribirse

```
if (i > 1)
   return x;
else if (i == 1)
   return y;
else            // if (i < 1)
   return z;
```

o incluso

```
if (i > 1)
   return x;
if (i == 1)
   return y;
return z;
```

11. Una buena documentación es esencial. La representación de una clase debe contener una explicación detallada de cómo la implementación concreta representa a los valores abstractos y de cuál es el invariante de la representación. Por ejemplo,

```
class lista {
     ...
  private:
     struct nodo {
        string clave;
        int valor;
        nodo* sig;
     };
     ...
     nodo* primero;
};
```


no nos dice si la lista está implementada mediante una lista simplemente enlazada, si está cerrada circularmente o no, si hay o no un nodo fantasma, si está ordenada o no crecientemente por el campo clave de cada nodo, etc. Debe, por lo tanto, documentarse adecuadamente la forma en que la representación será utilizada.

No comentéis código autoexplicativo ni repitáis lo obvio. He aquí unos pocos ejemplos de comentarios inútiles, superfluos:

```
// retornamos cierto si encontrado es cierto
return encontrado;

// incrementamos el contador
cont++;

// inicializamos a 0 todas las componentes de v
for (int i = 0; i < n; i++)
    v[i] = 0;
```

Un comentario debe aportar información que no es inmediatamente evidente. Por lo general, conviene efectuar un comentario general previo sobre el comportamiento de cada función o método, con indicaciones sobre los puntos sutiles de la implementación. Evitad el uso de comentarios intercalados con el propio código, salvo que resulten absolutamente imprescindibles. La documentación no es un sustituto adecuado de una descomposición funcional correcta, de manera que si la implementación de un determinado método o función es larga y compleja, la solución no es poner abundantes comentarios sino descomponerla en "piezas" manejables y autoexplicativas (veáse los ejemplos de los puntos 5 y 6 de esta misma sección). Recordad que unos identificadores bien escogidos ayudan considerablemente a la labor de documentación. Lo usual es documentar exhaustivamente la especificación (precondición, postcondición, errores, coste, etc.) de cada función o método público en su punto de declaración, ya que el usuario necesita esa documentación para hacer un uso correcto. En la parte de implementación (en el fichero `.cpp` o `.t`) de un método público sólo se documentarían detalles relativos a la implementación. Por otro lado, para los métodos y funciones privados interesa situar toda su documentación (tanto la relativa a su especificación como la relativa a su implementación) en el fichero `.cpp` o `.t` ya que es allí donde se usa.

Es importante que en vuestras prácticas añadáis documentación adicional que habitualmente no se pondría: las justificaciones para las decisiones de diseño tomadas, con la discusión pertinente de las ventajas e inconvenientes, de las alternativas consideradas, y de porqué éstas fueron desechadas.

Por ejemplo:

```
// dicc.rep

// Justificacion de la representacion escogida:
// Para la implementacion hemos decidido usar una tabla de hash
```

```
// con encadenamientos separados; la operacion que mas a menudo se
// utiliza en esta clase es 'localiza' y conviene que tenga maxima
// eficiencia. Por otro lado el numero de elementos en la tabla estara
// siempre en torno al valor conocido M en el momento de crear la
// tabla. Elegimos los encadenamientos separados por su
// sencillez y para evitar un degradacion del tiempo de busqueda para
// factores de carga proximos a 1. Otras alternativas consideradas
// estaban basadas en arboles de busqueda, pero fueron desechadas
// porque no precisabamos recorrer los elementos en orden o accesos
// por rango, y las implementaciones mediante arboles ofrecen un
// rendimiento en busquedas algo inferior al de la tabla de hash.
// ... (mas explicaciones)

...
```

8 Entorno de programación

Salvo que se diga lo contrario en la documentación específica de la práctica, los ficheros `.hpp` correspondientes a todos los módulos/clases que intervienen en una práctica estarán disponibles en las páginas Web de la asignatura.

Trabajando en las máquinas Solaris del LCFIB las clases error, util y mem_din se habrán de incluir necesariamente mediante

```
#include <eda/error>
#include <eda/util>
...
```

El módulo util incluye funciones diversas para conversión de strings a números y viceversa, generación de números aleatorios, etc. Su uso se documenta en el propio fichero de cabecera `<eda/util>`.

Observad que los ficheros de cabecera no tienen extensión `.hpp` siguiendo el convenio de C++ para ficheros de cabecera estándar (p.e., `iostream`) pero que se encuentran en un subdirectorio llamado `eda` de un subdirectorio estándar de inclusión; por eso se ha de escribir `#include <eda/error>` y no `#include <error>`. Para las restantes clases o módulos que intervengan en la práctica deberéis bajar los ficheros de cabecera correspondientes desde las páginas Web de EDA e instalarlos en el mismo subdirectorio que los ficheros `.rep` `.cpp` y `.t`. Si, por ejemplo, téneis que desarrollar o usar el módulo o clase X, bajad el fichero X`.hpp` desde la Web, copiadlo en vuestro subdirectorio y haced las inclusiones mediante:

```
...
#include <eda/error>
#include <eda/util>
```



```
#include <iostream>
...
#include "X.hpp"
...
```

Podéis conseguir el mismo efecto en vuestras instalaciones particulares en casa, creando un subdirectorio `eda` en un subdirectorio estándar de vuestra máquina (será algo del estilo `/usr/include` en Linux) y copiando allí los ficheros `error` y `mem_din`, y los restantes en el subdirectorio en el que estéis desarrollando la práctica. Alternativamente, si ponéis los ficheros en un subdirectorio `eda` del directorio `/x/y/z/` (es decir, los ficheros están en `/x/y/z/eda`) se puede informar al compilador cómo encontrar los ficheros de cabecera usando el *flag* `-I` (*flag* de inclusión):

```
g++ -c -ansi -Wall -I/x/y/z mi_prog.cpp
```

Para el montaje en el entorno Solaris del LCFIB tendréis que escribir sólo los nombres de los ficheros objeto desarrollados por vosotros y añadir `-leda` para indicar que usaréis la librería llamada `libeda`:

```
g++ -o pract mi_pract.o mi_mod1.o -leda
```

La *librería* `libeda`, aunque no es estándar, está situada en un subdirectorio estándar y contiene todos los ficheros objeto que os suministramos: `error.o`, `mem_din.o`, `util.o`, etc. Si trabajáis con Linux podéis descargar la versión para este sistema operativo de la librería `libeda` desde las páginas Web de EDA e instalarla en un un subdirectorio estándar (típicamente `/usr/lib`) de tal modo que la podréis usar del mismo modo que en el entorno Solaris.

Nada impide que durante el desarrollo de la práctica modifiquéis una copia particular de un fichero `.hpp`, aunque no es demasiado recomendable. Por ejemplo, si estamos desarrollando una clase que define los métodos `f` y `g`, hemos escrito la implementación de `f` y queremos probar si compila y se comporta correctamente podemos eliminar la declaración del método `g` en el fichero `.hpp`, o mejor aún, anteponer `//` para comentar la línea correspondiente. Pero existe una tercera posibilidad, más "segura", que consiste en añadir una definición trivial para `g` en el fichero `.cpp`:

```
// X.cpp

void f(...) { // definimos f
   ...
}

int g(...) {} // definimos g como una funcion con cuerpo vacio
```

Finalmente, tened presente que para la ejecución automatizada de las prácticas sólo se usarán vuestros ficheros `.rep`, `.cpp` y `.t`. Los ficheros de cabecera usados en la ejecución automática serán los originales; asimismo, los ficheros objeto no provenientes de un `.cpp` vuestro serán los originales incluídos en la librería `libeda` o los que os proporcionemos nosotros.

9 Normas de programación

Las siguiente normas son de obligado cumplimiento:

- No se pueden usar variables globales, atributos variables de clase, o variables estáticas salvo que se especifique lo contrario.

- No se pueden usar técnicas (p.e., funciones o clases `friend`) que contravengan la ocultación de tipos o que permitan modificar una constante una vez inicializada, salvo que se especifique lo contrario.

- Se han de respetar todos los convenios relativos a gestión de errores. En particular, se ha de garantizar que ninguna estructura de datos se modifica si se produce un error y respetar la regla de prioridad de códigos en el caso de situaciones de error simultáneas.

- No se debe usar ninguna clase o módulo salvo los indicados explícitamente por la documentación específica de la práctica; en su caso, se respetarán las restricciones de uso que puedan haber. En particular, no se pueden usar ninguna de las clases o módulos de la STL, excepto cuando y donde la documentación de la práctica lo permita. La clase `string` constituye una excepción a esta norma.

  Es frecuente el caso en el que uno o más métodos de una clase tienen parámetros de la clase `list` para recibir datos o devolver resultados; salvo que se indique lo contrario la clase `list` no podrá utilizarse con ningún otro fin, p.e. en la representación de los objetos de la clase.

- No se pueden utilizar apuntadores genéricos (`void*`), salvo que se especifique lo contrario.

- Todos los módulos (tanto la representación `.rep` como la implementación `.cpp` y `.t`) deben estar debidamente documentados y debe usarse sangrado para garantizar una legibilidad mínima del código.

- No se puede compartir código o documentación con otros equipos. Os animamos al intercambio de ideas y la colaboración entre equipos. Pero no crucéis la frontera entre una sana cooperación y un vulgar plagio. La política de la asignatura con relación a los casos de copia implica el suspenso (0) directo de la asignatura y la eventual solicitud de apertura de expediente de desvinculación para los alumnos implicados.

En caso de duda sobre la posibilidad o no de emplear una determinada construcción de C++ o si con ello se incumple alguna de las normas o no, recomendamos que consultéis a vuestro profesor de laboratorio, en sus horas de consulta o por correo electrónico.

Por otra parte, se valorarán negativamente o muy negativamente los siguientes aspectos:

- Diseño de la representación mal o incorrectamente concebido y/o justificado.

- Documentación insuficiente o incomprensible. Documentación en flagrante contradicción con la implementación. Documentación trivial (especialmente si las partes no triviales de la clase o módulo no se documentan o están mal documentadas).

- Descomposición funcional inexistente o inadecuada (no responde a criterios lógicos, no se usa cuando es claramente apropiado hacerlo, etc.)

- Código enrevesado, ilegible, innecesariamente oscuro o complicado. "Lógica" de las funciones y métodos antinatural y complicada. Código no estructurado. Uso abusivo o inapropiado de `breaks`, `continues`, `returns`, etc.

- Elección de identificadores o estilo de programación confuso o incoherente. Uso inconsistente de los convenios de nombramiento, de sangrado, ...

- Adaptación inadecuada o inconsistente de algoritmos o código procedente de fuentes externas: libros, artículos, Internet, ... En particular, en Internet es fácil encontrar el código fuente de un gran número de algoritmos y estructuras de datos, pero con frecuencia contiene errores o es muy chapucero (tiene defectos de estilo graves).

- Ineficiencias en espacio o en tiempo manifiestas.

- Existencia de avisos (*warnings*) en la compilación. Recordad que habéis de utilizar los *flags* `-ansi` y `-Wall` para compilar.

A La clase string

Un *string* es una cadena de caracteres. La librería estándar de C++ ofrece una clase `string` que nos permite manejarlos con toda comodidad. Ocasionalmente, habremos de emplear cadenas de caracteres representadas según las convenciones de C: es decir, un array de caracteres (`const char*`) terminado con el carácter cuyo código ASCII es 0 ('\0'). A estos últimos les llamaremos `C-strings` para distinguirlos de los strings que proporciona C++.

Para usar strings habremos de incluir el fichero del mismo nombre:

```
#include <string>
...
```

La clase ofrece las operaciones habituales de construcción, asignación, etc. y podemos leerlos o imprimirlos de igual forma que los tipos elementales. También podemos asignar a string una cadena literal entrecomillada (un C-string constante). A veces necesitamos producir un C-string a partir de un string: para ello usaremos `c_str()`:

```
string s;
cout << "Nombre del fichero: ";
cin >> s;
```

```
fstream f(s.c_str()); // la constructora de la clase fstream
                      // (ver apendice B) tiene como parametro
                      // el nombre del fichero que es un const char*
```

Los operadores + y += sirven para concatenar.

```
string s, t; // crea dos strings vacios

s = "Hola";         // asigna el C-string constante "Hola" al string s
t = s + " mundo!"; // concatena s con " mundo!"

cout << t << endl; // imprime Hola mundo!

t += " y adios!";

cout << t << endl; // imprime Hola mundo! y adios!
```

La longitud de un string se obtiene con `length()` y podemos acceder a los caracteres individualmente, como si se tratase de un array (pero no lo es):

```
bool es_vocal(char c) { ... }

int cuantas_vocales(const string& s) {
    int nv = 0;
    for (int i = 0; i < s.length(); ++i)
       if (es_vocal(s[i])) ++nv;
    return nv;
}
```

Los operadores de comparación entre strings están definidos respetando el orden alfabético:

```
string s = "casa";
string t = "cama";
string u = "dado";
string v = u + "s"; // v == "dados"

cout << (s < t) << endl;  // imprime false
cout << (s <= u) << endl; // imprime true
cout << (u > v) << endl;  // imprime false
cout << (s != t) << endl; // imprime true
```

El método `substr()` nos permite extraer *substrings* de uno dado:

```
string s = "portaaviones";
```



```
s.substr()          // retorna una copia de s
s.substr(5);        // retorna el substring "aviones"
s.substr(1,6);      // retorna "ortaav"
```

En general, `substr`(i, n) retorna el substring que comienza en la posición i y tiene longitud n. Los caracteres se indexan de 0 en adelante. Si n no se da, entonces se retorna el substring que va desde la posición i hasta el final. Si i tampoco se da entonces se considera que $i = 0$. Se produce un error si $i >$ `length()`.

Con `replace()` podemos reemplazar un substring por otro. Por ejemplo,

```
string s = "portaaviones";

s.replace(5,6,"helicoptero");
cout << s << endl;    // imprime portahelicopteros
```

Existen versiones más sofisticadas de `replace`; en la versión básica del ejemplo damos la posición inicial y longitud del substring a reemplazar y el string que reemplaza.

Para terminar esta breve descripción comentamos algunas de las facilidades que proporciona la clase `string` para la búsqueda dentro de un string. El método básico se denomina `find()` y nos permite hallar la primera ocurrencia de un carácter o substring en un string a partir de una cierta posición. El método devolverá la posición de inicio del substring o carácter buscado o `npos` (un valor especial) para indicar el fracaso de la búsqueda.

```
string s = "Lola, pasame la cola";

s.find("ola");       // retorna 1
s.find("ola", 3);    // retorna 17
s.find("pase");      // retorna npos
s.find('p');         // retorna 7
s.find('l', 3);      // retorna 13
```

El primer argumento de `find()` es el carácter o substring que se busca. El segundo es la posición (inclusive) a partir de la cual se inicia la búsqueda. Por defecto, la búsqueda se inicia en el principio del string.

El convenio de usar `npos` para indicar el fracaso de una búsqueda introduce ciertos inconvenientes a la hora de usar el método `find()`. El resultado de una búsqueda deberá siempre recogerse en una variable del tipo `string::size_type` y habremos de testear si el resultado es `string::npos`.

```
// reemplaza todas las apariciones en s de s1 por s2
//
// ej: { s = "ana se come una banana" }
//     global_replace(s, "ana", "alle")
```

```
//     { s = "alle se come una ballena" }

string global_replace(string& s, const string& s1, const string& s2) {
        string::size_type idx = 0;
        int l1 = s1.length();
        int l2 = s2.length();
        while (idx != string::npos && s.find(s1, idx) != string::npos) {
           s.replace(idx, l1, s2);
           idx += l2;
        }
}
```

B La clase **fstream**

C++ ofrece varios medios para manipular información almacenada en ficheros. Uno de los más simples es el uso de `fstreams`, esto es, flujos (secuencias) de caracteres asociados a un fichero de texto. De hecho, en `fstream` se definen varios clases de objetos. Un `ifstream` es similar al flujo estándar de entrada (`cin`), pero la información se lee de un fichero. Por su parte, un `ofstream` es similar al flujo estándar de salida (`cout`), escribiéndose la información sobre un fichero de texto. Un `fstream` nos permite lecturas y escrituras sobre un mismo fichero de texto.

Para escribir en un fichero hemos de crear ("abrir") un `ofstream` dando como parámetro de la constructora el nombre del fichero. Dicho parámetro es un C-string (veáse el apéndice A).

Se puede producir un error al crear el flujo (p.e. el fichero no tiene los permisos adecuados) y habremos de comprobarlo.

```
string nom_fich;
cout << "Nombre del fichero. " << endl;
cin >> nom_fich;

ofstream f(nom_fich.c_str());

if (!f) { // no se pudo abrir el flujo por la razon que sea ...
} else {
  // la apertura de f ha sido exitosa
  ...
}
```

Una vez abierto correctamente podemos escribir en el fichero del mismo modo que en `cout`. La siguiente función escribe en un fichero, cuyo nombre se nos da, un valor n y la secuencia de los primeros n números primos:

```
bool es_primo(int n) { ... }
```



```
// n > 0
void escribe_primos(ofstream& fich, int n) {
     int i = 1, escritos = 0;
     fich << n;
     while (escritos != n) {
       while (!es_primo(i)) ++i;
       fich << " " << i;
       ++escritos;
       ++i;
     }
}
```

La operación destructora de la clase `ofstream` se encarga de volcar lo que quede en el *buffer* y cierra el flujo y el fichero (lo que permite que después pueda ser abierto para lectura, por ejemplo). Se puede forzar el volcado del *buffer* en cualquier momento enviando un `flush` (ej: `f « flush;`). También se puede enviar un `endl` que imprime un salto de línea y envia un `flush`. Un *buffer* es una zona de memoria intermedia donde se almacenan temporalmente caracteres antes de enviarlos físicamente al fichero, de modo que se imprimen varios caracteres con una sola operación de E/S y se evitan constantes accesos al disco.

Hay otros métodos para manipular los `ofstreams` pero los elementos vistos acá son suficientes para vuestros propósitos.

Para leer desde un fichero hay que crear ("abrir") un `ifstream`. En muchos aspectos funciona de modo análogo a un `ofstream`.

```
// leemos los contenidos de un fichero "primos.dat"
// generado previamente con escribe_primos()
// y lo guardamos en un array primos[]

ifstream f("primos.dat");

if (!f) {
   cout << "No pude abrir el fichero" << endl;
} else {
   int n, leidos = 0;
   f >> n;
   int* primos = new int[n];
   while (leidos != n) {
     f >> primos[leidos];
     ++leidos;
}
...
```

Para leer carácter a cárácter o por líneas un fichero de texto la clase `ifstream` podemos usar el método `get()` y la función `getline()`:

```
ifstream& ifstream::get(char& c)
          // obtiene el siguiente caracter del flujo y lo pone
          // en la variable c; devuelve el flujo
          // el flujo queda en el "estado" 0 si se ha llegado al final

ifstream& getline(ifstream& f, string& s)
          // lee una linea del flujo f y la pone el string s
          // devuelve el flujo f
          // el flujo queda en el "estado" 0 si se ha llegado al final
```

Por ejemplo, la siguiente funcion copia el contenido de un fichero de entrada (`finp`) sobre un fichero de salida (`fout`):

```
void copia_fich(ifstream& finp, ofstream& fout) {
   char c;
   while (finp.get(c))
     fout << c;
}
```

Y la siguiente función imprime en un flujo de salida las líneas del fichero de entrada que contienen el string s, precedidas de su número de línea:

```
void print_matching_lines(ostream& os, ifstream& finp,
                          const string& s) {
     string line;
     int nlineas = 0;
     while (getline(finp, line)) {
        nlineas++;
        if (line.find(s) != string::npos)
           os << nlineas << ": " << line;
     }
}
```

Ocasionalmente, leemos un carácter pero no deseamos "avanzar" en el flujo. Puesto que la lectura desde ficheros de texto también se hace mediante *buffers* esto es fácil: el último carácter leído mediante `get` puede ser "devuelto" al flujo mediante el método `putback()`.

```
// buscamos la primera vocal en minusculas en fich

char c;
bool hallado = false;
bool fin = fich.get(c);
while (!fin && !hallado) {
 hallado = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
 fin = fich.get(c);
```



```
}

if (!fin) {
  // hemos encontrado una vocal pero ya hemos avanzado en fich
  fich.putback(c);
  // ahora esta funcion se encuentra a la vocal al principio del flujo
  otra_funcion(fich);
  ...
}
```

En esta breve descripción se han omitido muchos detalles, incluyendo el tratamiento de errores que se produzcan durante la lectura o la escritura en ficheros. No hemos descrito los métodos para manipular `fstreams`, muchos de los métodos que se aplican sobre `ifstreams` o `ofstreams`, otras clases para el manejo de ficheros, ficheros binarios (no de texto), métodos para el acceso directo, etc.

Referencias

1. Cline, M., Lomow, G., Girou, M.: *C++ FAQs, 2nd edition*. Addison-Wesley, 1999.
2. García de Jalón, J., Rodríguez, J.I., Sarriegui, J.M., Goñi, R., Brazález, A., Funes, P., Larzabal, A., Rodríguez, R.: *Aprenda C++ como si estuviera en primero*. Escuela Sup. Ingenieros Industriales, Universidad de Navarra, 1998. Disponible en https://www.fib.upc.es/˜eda.
3. Josuttis, N.: *The C++ Standard Library*. Addison-Wesley, 1999.
4. Kernighan, B.W., Pike, R.: *The Practice of Programming*. Addison-Wesley, 1999. Existe traducción al castellano (Addison-Wesley Iberoamericana).
5. Ribó, J.M.: *C++ orientat a objectes*. Universitat de Lleida, 1999. Disponible en https://www.fib.upc.es/˜eda
6. Stroustrup, B.: *The C++ Programming Language, 3rd ed.*. Addison-Wesley, 1997. Existe traducción al castellano (Addison-Wesley Iberoamericana).