Tratamiento de errores.

Estrategias posibles para tratar errores:

- Devolver (vía `return`) un "valor de error".
  - Requiere que haya un valor de error disponible (típicamente NULL ó -1) que no puede ser confundido con una respuesta de no-error; cuando no lo hay, lo común es devolver un booleano indicando el estado de error y pasando el antiguo valor devuelto a un argumento por referencia.
  - Requiere que el código que llama a la función verifique si ha habido o no error mediante algún tipo de condicional, y lleva a código difícil de leer.

Tratamiento de errores.

- Lanzar una excepción
  - Requiere soporte del lenguaje de programación (C ó Pascal no las soportan; Java, C++ ó Delphi sí). En general, casi todos los lenguajes con orientación a objetos soportan excepciones.
  - Requiere que el código que llama a la función decida si quiere manejar la excepción (lanzada con `throw`), usando una sentencia `catch` apropiada.
- Mostrar un mensaje por pantalla y devolver cualquier valor. Esta estrategia sólo es admisible cuando se está depurando, la librería es muy pequeña, y somos su único usuario; e incluso entonces, es poco elegante. Su única ventaja es que resulta fácil de implementar.
- Exigir una función adicional, pasada como parámetro, a la que llamar en caso de error. Esto es común en librerías con un fuerte componente asíncrono; por ejemplo, cuando se hacen llamadas "AJAX" desde JavaScript.

Tratamiento de errores.

Ejemplo: Lanzando y capturando excepciones en C++

```
#include <iostream>
int divide(int a, int b) {
    if (b==0) {
        throw "division por cero"; // 'const char *'
    }
    return a/b;
}

using namespace std;
int main() {
    try {
        cout << divide(12,3) << "\n";
        cout << divide(1, 0) << "\n";
    } catch (const char * &s) {    // ref. al tipo
        cout << "ERROR: " << s << "\n";
    }
    return 0;
}
```

Tratamiento de errores.

- Las excepciones lanzadas pueden ser de cualquier tipo.
- No obstante, **se deben usar clases específicas**, tanto por legibilidad como por la posibilidad de usarlas para suministrar más información dentro de las propias excepciones.
- En el siguiente ejemplo, usamos una clase base `Excepcion`, con subclases `DivCero` y `Desbordamiento`:

Tratamiento de errores.

```
#include <iostream>
#include <string>
#include <climits>

class Excepcion {
    const std::string _causa;
public:
    Excepcion(std::string causa) : _causa(causa) {};
    const std::string &causa() const { return _causa; };
};

class DivCero : public Excepcion {
    public: DivCero(std::string causa) :
                        Excepcion(causa) {};
};
class Desbordamiento : public Excepcion {
    public: Desbordamiento(std::string causa) :
                        Excepcion(causa) {};
};
```

Tratamiento de errores.

```
int suma(int a, int b) {
    if ((a>0 && b>0 && a>INT_MAX-b) ||
        (a<0 && b<0 && a<INT_MIN-b)) {
        throw Desbordamiento("excede limites en suma");
    }
    return a+b;
}

int divide(int a, int b) {
    if (b==0) {
        throw DivCero("en division");
    }
    return a/b;
}
```

Tratamiento de errores.

```
using namespace std;
int main() {
 try {
    cout << divide(12,3) << "\n";
    cout << suma(INT_MAX, -10) << "\n";
     cout << divide(1, 0) << "\n";  // lanza excepcion
     cout << suma(INT_MAX,  10) << "\n";  // no evalua
} catch (DivCero &dc) {              // trata DivCero
     cout << "Division por cero: " << dc.causa() << "\n";
} catch (Desbordamiento &db) {   //  Desbordamiento
     cout << "Desbordamiento: " << db.causa() << "\n";
} catch (...) {    // trata cualquier otra excepcion
     cout << "Excepcion distinta de las anteriores!\n";
}
 return 0;
}
```

El modificador `const`

- En C++, es posible usar el modificador `const` en 3 contextos completamente distintos:

  para indicar una constante – usado en la declaración de la constante, y preferible al uso de `#define`.

  para indicar un método observador – en el interior del cual no se modifica el valor de la instancia actual de una clase ó estructura.

  como modificador de tipo – en los argumentos de entrada ó salida de un método, para indicar que el valor pasado no se puede cambiar dentro de la función. Tiene más sentido para argumentos pasados por referencia (`&`) o puntero (`*`), porque los argumentos pasados por valor siempre son copias, y por tanto modificarlos nunca tiene efectos externos a la función.

El modificador `const`

Ejemplo:

```
// devuelve el vértice del polígono más cercano
//al punto pasado
const Punto &
  Poligono::contiene(const Punto &p) const;
```

El primer y segundo `const` indican que el punto devuelto y el punto pasado son referencias no-modificables (uso como modificador de tipo). El último const indica que esta operación es observadora, y no modifica el polígono sobre el que se llama.

Paso de parámetros en las operaciones.

Convenciones a seguir en cuanto a los valores devueltos, asumiendo que se están usando excepciones para el manejo de las condiciones de error:

- Si la función no tiene un "valor devuelto" claro, como es el caso de las operaciones de mutación, no se debe devolver nada (declarando la función como `void`).
- En el caso de observadores, si *sólo se devuelve un valor*, el tipo de retorno de la función debe ser el del valor devuelto.
  - Si el valor es un bool, un int, un puntero, o similarmente pequeño, se debe devolver una copia.
  - Si el valor es más grande, pero no se va a guardar el original en ninguna parte (variable local elimina[illegible] al finalizar la función), también se debe devolver una copia.
  - Si el valor es grande, y además el original va a seguir estando disponible después de hacer la llamada, es más eficiente devolver una referencia constante.

copy and swap
move semantics

Paso de parámetros en las operaciones.

Si hay que devolver *más de un valor a la vez*, se puede devolver a través de argumentos "por referencia". El tipo de retorno de la función queda libre, y de nuevo se debería devolver `void`. Alternativamente, es posible agrupar todos los valores a devolver en una estructura ó clase, y devolverla como único valor.

```
// devuelve coordenadas del i-esimo punto en x e y
void Poligono::punto(int i, float &x, float &y) const;
// version mejorada: devuelve referencia a Punto
const Punto &Poligono::punto(int i) const;
```

Implementaciones dinámicas y estáticas

- Aquellos TADs cuyo tamaño pueden variar mucho, no tiene sentido asignarles un espacio fijo "en tiempo de compilación", y es mejor solicitar memoria al sistema operativo (SO) a medida que va haciendo falta.
- La memoria obtenida en tiempo de ejecución mediante solicitudes al SO se denomina *dinámica*. La que no se ha obtenido así se denomina *estática*
- La memoria dinámica se gestiona mediante punteros (dirección de memoria)

Implementaciones dinámicas y estáticas

En C++, la memoria se reserva y se libera mediante los operadores **new** y **delete**:

```
// conjunto de enteros en memoria dinámica
Conjunto<int> *c = new Conjunto<int>();
c->inserta(4);   // inserto un 4
(*c).inserta(2);  // "(*algo)." equivale a "algo->"
c->elimina(4);  // quito el primer 4
delete c;       // libero el conjunto
```

Y sus variantes **new[]** y **delete[]**, que se usan para reservar y liberar vectores, respectivamente:

```
// = 2^30 (y<<x equivale a y·2^x)
int cuantos = 1<<30;
// reserva 1GB de memoria dinamica
int *dinamicos = new int[cuantos];
...            // usa la memoria
delete[] dinamicos; // libera memoria anterior
```

Implementaciones dinámicas y estáticas.

Errores comunes con punteros y memoria dinámica:

- **Usar un puntero sin haberle asignado memoria** (mediante un `new` / `new[]` previo).
- **No liberar un puntero tras haber acabado de usarlo** (mediante un `delete` / `delete[]`). "fuga de memoria", o "escape de memoria" (*memory leak*).
- **Liberar varias veces el mismo puntero.**
- **Acceder a memoria ya liberada.**

Implementaciones dinámicas y estáticas

- Para evitar estos errores, se recomiendan las siguientes prácticas:
  - Usa memoria dinámica sólo cuando sea necesaria. No es "más elegante" usar memoria dinámica cuando no es necesaria; pero sí es más difícil y resulta en más errores.
  - Inicializa tus punteros nada más declararlos. Si no puedes inicializarlos inmediatamente, asígnales el valor NULL (= 0) .
  - Nada más liberar un puntero, asígnale el valor NULL (= 0); **delete()** nunca libera punteros a NULL; :

Implementación de un TAD con memoria dinámica

Ejemplo:

```
class Mapa {
  unsigned short *_celdas; // 2 bytes por celda
  int _ancho;
  int _alto;
public:
  // Constructor
  Mapa(int ancho, int alto) : _ancho(ancho), _alto(alto)
  {   if (ancho<1||alto<1) throw "dimensiones mal";
      _celdas = new unsigned short[_alto*_ancho];
    for (int i=0; i<alto*ancho; i++) _celdas[i] = 0;
    std::cout << "construido: mapa de "
      << _ancho << "x" << _alto << "\n";
 }
```

Implementación de un TAD con memoria dinámica

```
    // Modificador
    void celda(int i, int j, unsigned short v) {
            _celdas[i*_ancho + j] = v;
    }
    // Observador
    unsigned short celda(int i, int j)  const {
       return _celdas[i*_ancho + j];
    }
    // Destructor
     ~Mapa() {
       delete[] _celdas;
       std::cout << "destruyendo: mapa de "
            << _ancho << "x" << _alto << "\n";
     }
};
```

Implementación de un TAD con memoria dinámica

```
int main() {
 // llama a constructor (m0)
 Mapa *m0 = new Mapa(128, 128);
 // llama a constructor (m1)
 Mapa m1 = Mapa(2560, 1400);
 {    // comienza un nuevo bloque
      // llama a constructor (m2)
      Mapa m2(320, 200);
 }    // al cerrar bloque llama a los destructores (m2)
 delete m0;  // destruye m0 explicitamente
 return 0;
}
// al cerrar bloque llama a destructores
// (m1; m0 es puntero)
```

nunca hacer esta mierda de inicialización

Implementación de un TAD con memoria dinámica

- Variables estáticas (`m1` y `m2`) se liberan automáticamente al cerrarse los bloques en que están declarados
- Variables dinámicas (`m0`) se liberan explícitamente, ya que son reservadas con un `new`).
- Salida producida por el ejemplo:

```
construido: mapa de 128x128
construido: mapa de 2560x1400
construido: mapa de 320x200
destruyendo: mapa de 320x200
destruyendo: mapa de 128x128
destruyendo: mapa de 2560x1400
```

Implementación de un TAD con memoria dinámica

Operaciones que se crean por defecto en cualquier clase nueva:

- operador de asignación:

```
UnTAD& operator=(const UnTAD& otro);
```

- constructor de copia:

```
UnTAD(const UnTAD& otro);
```

- Ejemplo de uso:

```
Conjunto<int> c1().inserta(42);
Conjunto<int> c2;
c2 = c1;                  // operador asignacion
Conjunto<int> c3 = c1; // copia equivale a c3(c1)
Mapa m1(100, 100);
Mapa m2;
m2 = m1;          // operador asignacion
Mapa m3 = m1; // copia, equivale a m3(m1)
```

- Se copia campo a campo un objeto en el otro
- Si se utilizan punteros puede no funcionar como se desea.

Implementación de un TAD con memoria dinámica

Ventajas de las implementaciones estáticas:

- Más sencillas que las dinámicas, ya que no requieren código de reserva y liberación de memoria.
- El operador de copia por defecto funciona tal y como se espera (asumiendo que no se usen punteros de ningún tipo), y las copias son siempre independientes.
- Más rápidos que los dinámicos; la gestión de memoria puede requerir un tiempo nada trivial, en función de cuántas reservas y liberaciones hagan los algoritmos empleados. Esto se puede solucionar parcialmente haciendo pocas reservas grandes en lugar de muchas pequeñas.

Implementación de un TAD con memoria dinámica

Ventajas de las implementaciones dinámicas:

- Permiten tamaños mucho más grandes que las estáticas: hasta el máximo de memoria disponible en el sistema, típicamente varios GB; en comparación con los ∼8MB que pueden ocupar (en total) las reservas estáticas.
- No necesitan malgastar memoria - pueden reservar sólo la que realmente requieren.
- Pueden compartir memoria, mediante el uso de punteros (pero esto requiere mucho cuidado para evitar efectos indeseados).

Probando TADs

Tipos de pruebas:

- Pruebas formales: verifican la corrección teórica de los algoritmos/TADs, sin acceder a una implementación concreta.

  - Algoritmos: Especificaciones pre-post y verificación
  - TADs: Especificaciones algebraicas

  Muy trabajosas, susceptibles de error, solo se utilizan en campos de aplicación donde la seguridad e integridad de los datos es realmente crítica (como aviónica o control de centrales nucleares).

- Pruebas de caja negra: comprueban una implementación desde el punto de vista de su interfaz. No acceden alcódigo. Se usan para ver si "desde fuera de la caja" todo funciona como debe. Encuentran errores pero no demuestran que no los haya.
- Pruebas de caja blanca: verifican aspectos concretos de una implementación, teniendo en cuenta el código de la misma.

Pruebas de caja negra

Deben incluir, para cada operación, comprobaciones de que producen la salida esperada. Se elige un subconjunto representativo de los casos.

Ejemplo para el TAD `Conjunto`:

```
#include <cassert>
Conjunto<int> vacio;    Conjunto<int> c;
// tras insertar un elemento este existe
c.inserta(21);   assert(c.contiene(21));
// y se borra
c.elimiina(21);  assert(!c.contiene(21));
// borrarlo de nuevo produce error
try {  c.elimina(21);
} catch (ElementoInvalido e) {
error = true;
}
assert(error);
// el conjunto queda vacío
assert(c == vacio);
```

Pruebas de caja blanca.

Ejercitan ciertas trazas de ejecución para verificar que funcionan como se espera.

- Una traza de ejecución (execution path) es una secuencia de instrucciones que se pueden ejecutar con una entrada concreta; Ejemplo:

```
if (a() && b()) c(); else d();
```

  Trazas posibles:

  1. `a()`, `d()` (`a()` → `false`)
  2. `a()`, `b()`, `d()` (`a()` → `true` y `b()` → `false`)
  3. `a()`, `b()`, `c()` (`a()` → `true` y `b()` → `true`)

Las trazas relacionadas se agrupan en "tests unitarios" (*unit tests*), cada uno de los cuales ejercita una función o conjunto de funciones pequeño.

Pruebas de caja blanca.

- Cobertura: porcentaje de código que está cubierto por las trazas de un conjunto de pruebas de caja blanca.
- Idealmente, el 100 % del código debería estar cubierto.
- Importante: concentrarse en los fragmentos más críticos: Se usan más o pueden producir los problemas más serios.
- Existen herramientas que ayudan a automatizar el seguimiento de la cobertura de un programa.
- código muerto: código que no puede ser cubierto por ninguna traza (por ejemplo, `if (false) cout << "imposible!\n";`). Sólo puede ser detectado por inspección visual, herramientas de cobertura, o (en algunos casos), compiladores suficientemente avanzados.

Documentando TADs

- Todo TAD debe incluir en su documentación:
  - Descripción de alto nivel. Ejemplo: TAD `Sudoku`,
    Un tablero de Sudoku 9x9, que contiene las soluciones reales, las casillas iniciales, y los números marcados por el usuario en casillas inicialmente vacías.
  - Descripción de cada uno de sus campos públicos, ya sean operaciones, constantes, o cualquier otro tipo. La descripción debe especificar cómo se espera que se use.
- En C++, la documentación se escribe en el fichero `.h`. En el `.cpp` se escriben los comentarios a la implementación, es útil incluir cabeceras de función resumidas.
- Los TADs auxiliares deben tener el mismo nivel de documentación que el TAD principal.
- El estilo de la documentación de un proyecto, junto con el estilo del código del proyecto, se especifican al comienzo del mismo (idealmente mediante una "guía de estilo").

Ejemplo de uso de TADs

Problema:

Dado un número x, se cogen sus dígitos y se suman sus cuadrados, para dar x_1. Se realiza la misma operación, para dar x_2, y así mucho rato hasta que ocurra una de las dos cosas siguientes:

- se llega a 1, y se dice entonces que el número es "feliz"
- nunca se llega a 1 (porque se entra en un ciclo que no incluye el 1), y se dice entonces que el número es "infeliz"

Ejemplos:

- el 7 es feliz: $7 \rightarrow 49 \rightarrow 97 \rightarrow 130 \rightarrow 10 \rightarrow 1$
- el 38 es infeliz: $38 \rightarrow 73 \rightarrow 58 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4 \rightarrow 16 \rightarrow 37 \rightarrow 58$

Ejemplo de uso de TADs

```
#include "conjunto.h"
#include <iostream>
using namespace std;

int siguiente(int n) {
    int suma = 0;
    while (n>0) {
        int digito = n%10;
        suma += digito*digito;
        n /= 10; // avanzo de digito
    }
    return suma;
}
```

Ejemplo de uso de TADs

```
void psicoanaliza(int n) {
    Conjunto<int> c;
    while ( ! c.contiene(n)) {
         c.inserta(n);
        n = siguiente(n);
    }
    if (n==1) {
        cout << "feliz (llega a 1) " << endl;
    } else {
        cout << n << " infeliz (repite del " << n
            << " en adelante)" << endl;
    }
}

int main() {
    psicoanaliza(7);
    psicoanaliza(38);
}
```