Abstracción en Programación

Abstracción: Operación intelectual que ignora selectivamente partes de un todo para facilitar su comprensión.

Abstracción en la resolución de problemas: Ignorar detalles específicos buscando generalidades que ofrezcan una perspectiva distinta, más favorable a su resolución.

Abstracción: descomposición en que se varía el nivel de detalle.

Propiedades de una descomposición útil:

- Todas las partes deben estar al mismo nivel
- Cada parte debe poder ser abordada por separado
- La solución de cada parte debe poder unirse al resto para obtener la solución final

Mecanismos de Abstracción en Programación

- *Abstracción por parametrización*. Se introducen parámetros para abstraer un número infinito de computaciones.

  Ejemplo: cálculo de $\cos \alpha$.

- *Abstracción por especificación*. Permite abstraerse de la implementación concreta de un procedimiento asociándole una descripción precisa de su comportamiento.

  Ejemplo: `double sqrt(double a);`
  requisitos: a $\geq$ 0;
  efecto: `devuelve una aproximación de` $\sqrt{\texttt{a}}$.

  La *especificación* es un comentario lo suficientemente definido y explícito como para poder usar el procedimiento sin necesitar conocer otros elementos.

Abstracción por Especificación

Se suele expresar en términos de:

- **Precondición**: Condiciones necesarias y suficientes para que el procedimiento se comporte como se prevee.

- **Postcondición**: Enunciados que se suponen ciertos tras la ejecución del procedimiento, si se cumplió la precondición.

```
int busca_minimo(float * array, int num_elem)
/*
  precondición:
    - num_elem > 0.
    - 'array' es un vector con 'num_elem'
       componentes.
  postcondición:
    devuelve la posición del mínimo elemento
    de 'array'.
*/
```

Tipos de Abstracción

- **Abstracción Procedimental**. Definimos un conjunto de operaciones (procedimiento) que se comporta como una operación.

- **Abstracción de Datos (TDA)**. Tenemos un conjunto de datos y un conjunto de operaciones que caracterizan el comportamiento del conjunto. Las operaciones están vinculadas a los datos del tipo.

- **Abstracción de Iteración**. Abstracción que permite trabajar sobre colecciones de objetos sin tener que preocuparse por la forma concreta en que se organizan.

Abstracción Procedimental

Permite abstraer un conjunto preciso de operaciones de cómputo como una operación simple. Realiza la aplicación de un conjunto de entradas en las salidas con posible modificación de entradas.

- La identidad de los datos no es relevante para el diseño. Sólo interesa el número de parámetros y su tipo.

- Con abstracción por especificación es irrelevante la implementación, pero no qué hace.
  - *Localidad*: Para implementar una abstracción procedimental no es necesario conocer la implementación de otras que se usen, sólo su especificación.
  - *Modificabilidad*: Se puede cambiar la implementación de una abstracción procedimental sin afectar a otras abstracciones que la usen, siempre y cuando no cambie la especificación.

Especificación de una Abstracción Procedimental (I)

Propiedades:

- Útil
- Completa
- Consistente
- Indicar el comportamiento en todos los casos en que sea aplicable

Debe identificar claramente:

- Entradas
- Salidas
- Requisitos
- Efectos

- Elementos modificados

Especificación de una Abs. Proc. (II)

```
int elimina_deuda(int * partidas,
                  int num_partidas)
/*
  Argumentos:
  ----------
    partidas: Array monodimensional de enteros
              con 'num_partidas' componentes.
               Es MODIFICADO.
    num_partidas: Número de componentes de
                  'partidas'. num_partidas > 0.
  Valores de retorno:
  ------------------
    Valor absoluto de la suma de las partidas
    negativas.
  Efecto:
  ------
    Calcula la suma de las componentes de
    'partidas' con valor negativo. Pone a cero
     cada componente negativa, cambia el signo
    de la suma resultante y la devuelve.
*/
```

Especificación de una Abs. Proc. (III)

a) **Cabecera**: (Parte sintáctica) Indica el nombre del procedimiento y el número, orden y tipo de las entradas y salidas. Se suele adoptar la sintaxis de un lenguaje de programación concreto. Ejemplo, el prototipo de la función en C.

b) **Cuerpo**: (Parte semántica) Está compuesto por las siguientes claúsulas.

   1. Argumentos
   2. Requiere
   3. Valores de retorno
   4. Efecto

Especificación de una Abs. Proc. (IV)

1. *Argumentos*: Explica el significado de cada parámetro de la abstracción, no el tipo, que ya se ha indicado en la cabecera, sino qué representa.

   Indicar las restricciones sobre el conjunto datos del tipo sobre los cuales puede operar el procedimiento. Procedimiento *total* y *parcial*.

   Indicar si cada argumento es modificado o no. Cuando un parámetro vaya a ser alterado, se incluirá la expresión "Es MODIFICADO".

2. *Requiere*: Restricciones impuestas por el uso del procedimiento no recogidas en *Argumentos*. Ejemplo: que exista cierta operación para los objetos del tipo con que se llama a la abstracción o necesidad de que previamente se haya ejecutado otra abstracción procedimental.

Especificación de una Abs. Proc. (V)

3. *Valores de retorno* Descripción de qué valores devuelve la abstracción y qué significan.

4. *Efecto*: Describe el comportamiento del procedimiento para las entradas no excluidas por los requisitos. El efecto debe indicar las salidas producidas y las modificaciones producidas sobre los argumentos marcados con "Es MODIFICADO".

   No se indica nada sobre el comportamiento del procedimiento cuando la entrada no satisface los requisitos. Tampoco se indica nada sobre sobre cómo se lleva a cabo dicho efecto, salvo que esto sea requerido por la especificación. Por ejemplo, que haya un requisito expreso sobre la forma de hacerlo o sobre la eficiencia requerida.

Especificación junto al código fuente

- Falta de herramientas para mantener y soportar el uso de especificaciones $\Longrightarrow$ responsabilidad exclusiva del programador.

- Necesidad de vincular código y especificación.

- Incluirla entre comentarios en la parte de interfaz del código.

- Herramienta `doc++`. (`http://docpp.sourceforge.net`) Permite generar documentación de forma automática en distintos formatos (html, LaTeX, postscript, etc.) a partir del código fuente.

Especificación usando `doc++`

Toda la especificación se encierra entre:

```
/**
   ...
  */
```

1. Se pone una frase que describa toda la función.
2. Cada argumento se precede de: `@param`.
3. Los requisitos adiconales se indican tras `@Precondición`.
4. Los valores de retorno se ponen tras `@return`.
5. La descripción del efecto sigue a `@doc`.

Especificación usando doc++

```
int elimina_deuda(int * partidas,
                  int num_partidas)
/**

  Cancela la deuda pendiente y devuelve
     su montante.

  @param partidas: Array monodimensional de
                enteros con 'num_partidas'
               componentes. Es MODIFICADO.
  @param num_partidas: Número de componentes
               de 'partidas'. num_partidas > 0.

  @return  Valor absoluto de la suma de las
          partidas negativas

  @doc
    Calcula la suma de las componentes de
    'partidas' con valor negativo. Pone a
     cero cada componente negativa, cambia
     el signo de la suma resultante y la
     devuelve.
 */
```

Generalización o parametrización

Subiendo en el nivel de abstracción: crear una abstracción que se pueda usar para más de un tipo de dato. Parametrizar el tipo de dato.

```
int minimo(int a, int b)
{
  return (a < b ? a : b);
}
```

Esta función devuelve el mínimo de sus dos argumentos `int`. Sin embargo, la forma de calcular este valor no sólo es válida para cualquier par de `int`, sino también para cualquier par de valores para los que esté definida la operación <.

De modo más general:

```
T minimo(T a, T b)
{
  return (a < b ? a : b);
}
```

Generalización en C++

El lenguaje C++ soporta la generalización mediante funciones patrón (en inglés, *template*). El parámetro del tipo de dato se designa mediante la construcción `template <class T>`:

```
template <class T>
T minimo(T a, T b)
{
  return (a < b ? a : b);
}
```

Instanciación: Cada vez que se necesita una función específica (p.ej: `minimo` para `int`), se crea particularizando la función genérica (patrón).

Generalización en C++ (2)

```
template <class T>
void ordena(T * array, int num_elems);
/**
   Dispone los elementos de 'array' en
   sentido creciente.

   @param array: array de elementos a
      ordenar. Es MODIFICADO.
   @param num_elems: número de componentes
      de array. num_elems > 0.
   @precondition Debe estar definida la
      operación:
        bool operator<(const &T a, const &T b);

   @doc
   Modifica (si es necesario) la disposición
   de los componentes de array dejándolos en
    orden creciente (según el operador <).
 */
```

Diseño de Abs. Procedimentales

1. Minimalidad. Los requisitos los mínimos posibles.

2. Generalidad. Poder aplicarla en el mayor número de casos posibles.

3. Simplicidad. Realizar una única acción concreta.

4. Carácter de total o parcial.

Abstracción de Datos

Tipo de Dato Abstracto (TDA): Entidad abstracta formada por un conjunto de datos y una colección de operaciones asociadas.

Ej.: tipos de datos en el lenguaje C.

Conceptos manejados:

- *Especificación*: Descripción del comportamiento del TDA.
- *Representación*: Forma concreta en que se representan los datos en un lenguaje de programación para poder manipularlos.
- *Implementación*: La forma específica en que se expresan las operaciones.

Visiones de un TDA

Hay dos visiones de un TDA:

- *Visión externa*: especificación.
- *Visión interna*: representación e implementación.

Ventajas de la separación:

- Se puede cambiar la visión interna sin afectar a la externa.
- Facilita la labor del programador permitiéndole concentrarse en cada fase por separado.

Especificación de un TDA

La especificación es *esencial*. Define su comportamiento, pero no dice **nada** sobre su implementación.

Indica el tipo de entidades que modela, qué operaciones se les pueden aplicar, cómo se usan y qué hacen.

Estructura de la especificación:

1. *Cabecera*: nombre del tipo y listado de las operaciones.

2. *Definición*: Descripción del comportamiento sin indicar la representación. Se debe indicar si el tipo es mutable o no. También se expresa donde residen los objetos.

3. *Operaciones*: Especificar las operaciones una por una como abstracciones procedimentales.

TDA Fecha (I)

```
/**
  TDA Fecha.

  Fecha::constructor, dia, mes, año, siguiente,
   anterior, escribe, lee, menor, menor_o_igual.

  Definición:
  Fecha representa fechas según el calendario
  occidental.
  Son objetos mutables.
  Residen en memoria estática.

  Operaciones:
*/
```

TDA Fecha (II)

```
void constructor (Fecha * f, int dia, int mes,
                  int año)
/**
   Constructor primitivo.

   @param f:   Objeto creado. Debe ser nulo.
   @param dia: dia de la fecha. 0 < dia <= 31.
   @param mes: mes de la fecha. 0 < mes <= 12.
   @param año: año de la fecha.
         Los tres argumentos deben representar
         una fecha válida según el calendario
         occidental.
   @return Objeto Fecha correspondiente a la fecha
      dada por los argumentos.

   @doc
     Crea un objeto Fecha a partir de los argumentos.
   Devuelve el objeto creado sobre f.
*/
```

TDA Fecha (III)

```
void lee(Fecha * f, char * cadena)
/**
   Lee una fecha de una cadena.

   @param f:  Objeto creado. Es MODIFICADO.
   @param cadena: Cadena de caracteres que representa
            una fecha en formato dd/mm/aaaa.

   @return Objeto Fecha que representa la fecha leída
     en cadena.
*/

int dia(Fecha f)
/**
   Obtiene el día del objeto receptor.

   @param f: Objeto receptor.

   @return Dia del objeto f.
*/
```

TDA Fecha (IV)

```
int mes(Fecha f)
/**
   Obtiene el mes del objeto receptor.

   @param f: Objeto receptor.

   @return  Mes del objeto f.
*/

int año(Fecha f)
/**
   Obtiene el año del objeto receptor.

   @param f: Objeto receptor.

   @return Año del objeto f.
*/
```

TDA Fecha (V)

```
char * Escribe (Fecha f, char * cadena)
/**
   Escribe el objeto receptor en una cadena.

   @param f: Objeto receptor.
   @param cadena: Cadena que recibe la expresión
       de f. Debe tener suficiente espacio.
       Es MODIFICADO.

  @return Cadena escrita.

  @doc
    Sobre 'cadena' se escribe una representación
   en formato 'dd/mm/aaaa' del objeto f. Devuelve
   la dirección de la cadena escrita.
*/
```

TDA Fecha (VI)

```
void Siguiente(Fecha * f)
/**
   Cambia f por la fecha siguiente a la que
   representa.

   @param f: Objeto receptor. Es MODIFICADO.
*/

void Anterior(Fecha * f)
/**
   Cambia f por la fecha anterior a la que
   representa.

   @param f: Objeto receptor. Es MODIFICADO.
*/
```

TDA Fecha (VII)

```
bool menor(Fecha f1, Fecha f2)
/**
   Decide si f1 es anterior a f2.

   @param f1, f2: Fechas que se comparan.

   @return
    true, si f1 es una fecha estrictamente anterior
         a f2.
    false, en otro caso.
*/

bool menor_o_igual(Fecha f1, Fecha f2)
/**
   Decide si f1 es anterior o igual que f2.

   @param f1, f2: Fechas que se comparan.

   @return
    true, si f1 es una fecha anterior o igual a f2.
    false, en otro caso.
*/
```

Ejemplo del uso del TDA `Fecha`

```
#include <stdio.h>
#include "fecha.h"

int main()
{
  Fecha f, g;
  int dia, mes, anio;
  char c1[100], c2[100];

  printf("Introduzca el día del mes que es hoy: ");
  scanf("%d", &dia);
  printf("Introduzca el día del mes que es hoy: ");
  scanf("%d", &dia);
  printf("Introduzca el día del mes que es hoy: ");
  scanf("%d", &dia);

  constructor(&f, dia, mes, anio);
  Siguiente(f, &g);
  if (menor(f, g))
    {
      Escribe(f, c1);
      Escribe(g, c2);
      printf("El día %s es posterior a %s\n",
             c1, c2);
    }
  return 0;
}
```

Implementación de un TDA

Implica dos tareas:

a) Diseñar una representación que se va a dar a los objetos.

b) Basándose en la representación, implementar cada operación.

Dentro de la implementación habrá dos tipos de datos:

a) Tipo Abstracto: definido en la especificación con operaciones y comportamiento definido.

a) Tipo *rep*: tipo a usar para representar los objetos del tipo abstracto y sobre él implementar las operaciones.

Representación del TDA

- Tipo abstracto: TDA Fecha

- Tipo *rep*:

```
typedef struct Fecha {
  int dia;
  int mes;
  int anio;
} Fecha;
```

El tipo *rep* no está definido unívocamente por el tipo abstracto: `typedef int Fecha[3];`

TDA Polinomio ($a_n x_n + \cdots + a_1 x + a_0$).

Tipo *rep*: `typedef float polinomio[n + 1];`

Representación del TDA (II)

Es importante:

- Conectar ambos tipos (Función de Abstracción).
- Identificar los objetos del tipo *rep* que representan objetos abstractos válidos (Invariante de Representación).

Abstracción de datos en C (I)

Problemas:

- No existen mecanismos para separar la visión externa y la interna.

  El usuario del TDA Fecha tiene acceso total a los detalles de implementación:

```
Fecha f;
constructor(&f, 19, 3, 2001);
f.dia = 40; /* <<Error!! */
```

- Débil vínculo entre objetos de datos y sus operaciones. Las operaciones se implementa como funciones globales.

Abstracción de Datos en C (II)

- Construcción de objetos: la definición e iniciación son procesos separados.

```
Fecha f, g;

Siguiente (f, &g);
```

  Se está usando `f` antes de que haya sido iniciada: su contenido no es válido.

- Pobre integración natural en el lenguaje. La forma de comparar fechas (función `menor`) es artificiosa.

```
Fecha f, g;
f = constructor(20, 2, 2001);
g = constructor(30, 3, 2001);

if (menor(f, g)) {
  ...
}
```

  Es mucho más natural: `if (f < g)`

TDA y Ocultamiento de Información

Ocultamiento de información: Mecanismo para impedir el acceso a la representación e implementación desde fuera del tipo. Garantiza:

- Representación inaccesible: razonamientos correctos sobre la implementación.
- Cambio de representación e implementación sin afectar a la especificación del TDA.

Encapsulamiento: Ocultamiento de información y capacidad para expresar el estrecho vínculo entre datos y operaciones.

El Lenguaje de Programación C++

- C++ es un lenguaje que incluye como sublenguaje al ANSI C.
- Soporta múltiples paradigmas de programación. En particular, Programación Dirigida a Objetos.
- Soporte para TDA: clases.
- Soporta sobrecarga de operadores y funciones.
- Programación genérica: *templates*.
- Tipos referencia.
- Funciones `inline`.
- Tipo lógico: `bool`.

Clases en C++

Construcción `struct` de C: tipos estructurados no homogéneos, que permiten acceso a sus componentes.

Clase (`class`): construcción que permite implementar un TDA:

- Ocultamiento de información. Control de acceso a los componentes (*miembros*) regulado mediante dos niveles de acceso: público (`public`) y privado (`private`).

- Encapsulamiento. Sus componentes son tanto datos (*variables de instancia*) como operaciones (funciones miembro o *métodos*). La clase establece un fuerte vínculo entre todos los miembros (datos y operaciones). Ambos se indican juntos.

- Iniciación de objetos automática: constructores.

- Destrucción automática de objetos: destructores.

Clase `Fecha` Versión 0.1

```
class Fecha {
public:
  Fecha (int dia, int mes, int anio);
  void Lee(const char * cadena);
  int Dia() const;
  int Mes() const;
  int Anio() const;
  void Escribe(char * cadena) const;
  void Siguiente(Fecha * g) const;
  void Anterior(Fecha * g) const;
  bool menor(Fecha f2);
  bool menor_o_igual(Fecha  f2);

private:
  int dia;
  int mes;
  int anio;
};
```

Clase Fecha Versión 0.2

```
class Fecha {
public:
  Fecha (int dia, int mes, int anio);
  void Lee(const char * cadena);
  int Dia() const;
  int Mes() const;
  int Anio() const;
  void Escribe(char * cadena) const;
  void Siguiente(Fecha & g) const;
  void Anterior(Fecha & g) const;
  bool operator<(const Fecha & f2);
  bool operator<=(const Fecha & f2);

private:
  int dia;
  int mes;
  int anio;
};
```

Clase Fecha Versión 0.3

```
class Fecha {
public:
  Fecha (int dia, int mes, int anio);
  void Lee(const char * cadena);
  int Dia() const;
  int Mes() const;
  int Anio() const;
  void Escribe(char * cadena) const;
  void Siguiente(Fecha & g) const;
  void Anterior(Fecha & g) const;
  bool operator<(const Fecha & f2);
  bool operator<=(const Fecha & f2);

private:
  int dia;
  int mes;
  int anio;

  bool bisiesto(int anio) const;
  const static int dias_mes[12] =
   {31, 28, 31, 30, 31, 30,
   31, 31, 30, 31, 30, 31};
};
```

Clases en C++

- Control de acceso
- Acceso a los miembros
- Funciones miembro
- Constructores
- Funciones miembro inline

Control de acceso

Las clases permite controlar el acceso a sus componentes (miembros) usando dos nuevas palabras reservadas:

- `private`: los miembros definidos tras esta indicación sólo son accesibles por miembros de la clase, pero no por nadie externo a la clase. Ocultamiento de información.

- `public`: los miembros definidos son accesibles para cualquiera.

Estas palabras se incluyen en la definición de la clase entre las declaraciones de los miembros. Tienen efecto desde que se declaran hasta que aparece otra o hasta al final. Por defecto, todos los miembros son privados.

Acceso a los miembros

En la definición de una clase se incluyen tanto definiciones de datos (variables de instancia) como operaciones asociadas (funciones miembro o métodos). Esto junto con el control de acceso (ocultamiento de información) favorece el encapsulamiento.

Todas las funciones miembro tienen acceso a todos los miembros de la clase tanto `public` como `private`.

Cada función miembro recibe implícitamente un argumento oculto que referencia al objeto sobre el que se aplica. Se trata de un puntero al objeto de nombre `this`, que no se declara explícitamente pero que siempre está disponible.

El acceso a otros miembros de la case se hace nombrándolos directamente o a través de `this`:

```
Fecha::Fecha (int D, int M, int A) {
  dia = D;
  this->mes = M;
  anio = A;
}
```

Definición de funciones miembro

Las funciones miembro se declaran dentro de la definición de la clase y se definen fuera de ésta.

Para indicar que son miembros de su clase se usa el operador de ámbito (`::`):

```
int Fecha::Mes() const {
  return mes;
}

void Fecha::Siguiente(Fecha & g) {
  ...
}
```

La definición de las funciones miembro es parte de la implementación (visión interna del TDA). Por ello, su definición se incluye en el fichero `.cpp` y no en el `.h`. El usuario del tipo, no necesita conocer su implementación para usarlas. Su definición sólo es necesaria en tiempo de enlazado, no de compilación.

Funciones miembro `inline`

Existen funciones miembro pequeñas que, por cuestiones puramente de *eficiencia* interesa que sean definidas `inline`.

Esto no afecta al usuario del tipo de dato, pero sí al compilador. Es necesario conocer su definición en tiempo de compilación (antes del enlazado). Por ello es conveniente incluir su definición en el fichero cabecera.

Las funciones miembro `inline` se pueden definir:

a) dentro de la clase

b) fuera de la clase

Definición de funciones miembro `inline`

a) Dentro de la clase: Su definición se pone justo después de de la declaración de la función. No se incluye la palabra reservada `inline`:

```
class Fecha {
  int Dia() const
    { return dia; }
};
```

b) Fuera de la clase: La definición se hace como en el caso general, pero se precede de la palabra reservada `inline` y se incluye en el fichero cabera, una vez finalizada la definición de la clase:

```
class Fecha {
  int Dia() const;
};

inline int Fecha::Dia() const {
  return dia;
}
```

Funciones amigas (`friend`)

En ocasiones es necesario que una función global (no miembro de una clase) tenga acceso a sus miembros privados. Esto implica saltarse los mecanismos de control de acceso. C++ ofrece una solución para estas situaciones especiales: funciones amigas.

Se tienen que declarar dentro de la definición de la clase, precediendo su definición con la palabra reservada `friend`.

```
class Fecha {
  friend ostream & operator<<(ostream &s,
                const Fecha & f);
};
```

Constructores (I)

Son funciones invocadas automáticamente en cuanto se define un objeto de la clase. Su objetivo es poner el objeto en un estado inicial válido.

- No devuelven objetos ni tienen tipo de dato de retorno, ni siquiera `void`.
- Su nombre coincide con el de la clase.
- Puede haber más de un constructor, pero sus declaraciones (prototipos) han de ser distintas.
- Un constructor que no recibe argumentos es el *constructor por defecto*:

```
class racional {
  racional();
};
```

Constructores (II)

Constructor de copia: permite crear un objeto de la clase a partir de otro.

```
class racional {
  racional(const racional & r);
};

racional::racional(const racional & r)
{
  num = r.num;
  den = r.den;
}
```

No es obligatorio definir constructores, aunque sí muy conveniente.

Si no se define ningún constructor, el compilador crea dos: uno por defecto y otro de copia.

Destructores

- Son las operaciones antagónicas de los constructores.
- Su objetivo es liberar todos los recursos asociados al objeto (p.ej.: memoria).
- No tienen tipo de retorno.
- Su nombre se construye anteponiendo ~ al nombre de la clase:

```
class Fecha {
  ~Fecha();
};
```

- Se invocan automáticamente cuando un objeto deja de existir.

Implementación de la clase `Fecha` (I)

```
#include <iomanip>
#include "Fecha.h"


Fecha::Fecha(int Dia, int Mes, int Anio)
{
  this->dia = Dia;
  this->mes = Mes;
  this->anio = Anio;
};


int Fecha::Dia() const
{
  return dia;
};


int Fecha::Mes() const
{
  return mes;
};
```

Implementación de la clase `Fecha` (II)

```
int Fecha::Anio() const
{
  return anio;
};


void Fecha::Escribe(char * cadena) const
{
  cout << dia << '/' << mes << '/' << anio;
};
```

Implementación de la clase `Fecha` (III)

```
void Fecha::Siguiente(Fecha & g) const
{
  if ((dia < dias_mes[mes - 1]) ||
      (bisiesto(anio) && dia < 29))
    {
      g.dia = dia + 1;
      g.mes = mes;
      g.anio = anio;
    }
  else
    {
      g.dia = 1;
      if (mes < 12)
        {
          g.mes = mes + 1;
          g.anio = anio;
        }
      else
        {
          g.mes = 1;
          g.anio = anio + 1;
        }
    }
}
```

Implementación de la clase `Fecha` (IV)

```
void Fecha::Anterior(Fecha & g) const
{
  if (dia > 1)
    {
      g.dia = dia - 1;
      g.mes = mes;
      g.anio = anio;
    }
  else
    if (mes > 1)
      {
        g.mes = mes - 1;
        g.dia = (mes != 2 ? dias_mes[g.mes - 1] :
                 (bisiesto(anio) ? 29 : 28));
        g.anio = anio;
      }
    else
      {
        g.dia = 31;
        g.mes = 12;
        g.anio = anio - 1;
      }
}
```

Implementación de la clase `Fecha` (V)

```
bool Fecha::operator <(const Fecha & f2)
{
  if (anio < f2.anio)
    return true;
  if (anio > f2.anio)
    return false;
  if (mes < f2.mes)
    return true;
  if (mes > f2.mes)
    return false;
  if (dia < f2.dia)
    return true;
  return false;
};
```

Implementación de la clase `Fecha` (VI)

```
bool Fecha::operator<=(const Fecha & f2)
{
  if (anio <= f2.anio)
    return true;
  if (anio > f2.anio)
    return false;
  if (mes <= f2.mes)
    return true;
  if (mes > f2.mes)
    return false;
  if (dia <= f2.dia)
    return true;
  return false;
};
```

```
bool
Fecha::bisiesto(int anio) const
{
  return (((anio % 4 == 0) && (anio % 100 != 0))
          || (anio % 400 == 0));
}
```

Sobrecarga de operadores (I)

- Los operadores permiten conectar operandos para generar valores.
- El conjunto de operadores de C/C++ es amplio y tienen una semántica bien establecida pero fija
- Su uso es deseable para nuevos tipos de datos: números complejos, racionales, matrices, . . .
- C++ permite dar significado a los operadores aplicados a nuevos tipos de datos: *sobrecarga de operadores*.

Sobrecarga de operadores (II)

Para sobrecargar un operador se define una función con nombre el operador antepuesto de la palabra reservada `operator`. Los tipos de retorno y de los argumentos serán los necesarios en cada caso:

```
racional operator+(const racional &r,
                   const racional &s);
vect operator*(const vect &v, const matriz &m);
Fecha & operator++(Fecha);
```

La sobrecarga se puede realizar como:

- función global.
- función miembro: el primer argumento se reemplaza por el objeto receptor.

Se pueden sobrecargar todos los operadores excepto: `.`, `.*`, `?:`, `sizeof`, `::`.

Sobrecarga de operadores (III)

Operador * sobrecargado como función miembro:

```
racional& racional::operator*(const racional &s)
{
  return racional(num * s.num, den * s.den);
}
```

Operador * sobrecargado como función global:

```
racional& operator*(const racional &r, const racional &s)
{
  return racional(r.num * s.num, r.den * s.den);
}
```

El acceso a la representación requiere que la función sea declarada amiga de la clase:

```
class racional {
  friend racional& operator*(const racional &r,
                             const racional &s);
  ...
};
```

Sobrecarga de operadores (IV)

```
#include <iostream>
class Fecha {
  friend ostream & operator<<(ostream &s, const Fecha &f);
  friend istream & operator>>(istream &i, Fecha & f);

  Fecha & operator++();    // Versión prefijo
  Fecha & operator++(int); // Versión postfijo
};

ostream & operator<<(ostream &s, const Fecha &f)
{
  s << f.dia << '/' << f.mes << '/' << f.anio;
  return s;
}
```

Implementación de TDA

En toda implementación existen dos elementos característicos muy importantes:

- Función de abstracción: conecta los tipos abstracto y *rep*.

- Invariante de representación: condiciones que caracterizan los objetos del tipo *rep* que representan objetos abstractos válidos.

Siempre existen aunque, habitualmente, no se es consciente de su existencia.

Función de abstracción

Define el significado de un objeto *rep* de cara a representar un objeto abstracto. Establece una relación formal entre un objeto *rep* y un objeto abstracto.

$$f_A : rep \longrightarrow A$$

Es una aplicación sobreyectiva.

Ejemplos:

- TDA `racional`:

$$\{\texttt{num, den}\} \longrightarrow \frac{\texttt{num}}{\texttt{den}}$$

- TDA `Fecha`:

$$\{\texttt{dia, mes, anio}\} \longrightarrow \texttt{dia/mes/anio}$$

- TDA `Polinomio`:

$$r[0..n] \rightarrow r[0] + r[1]x + \ldots + r[n]x^n$$

Invariante de Representación

Invariante de Representación (I.R.): Expresión lógica que indica si un objeto del tipo *rep* es un objeto del tipo abstracto o no.

Ejemplos:

- TDA `racional`: Dado el objeto *rep* `r`=$\{$`num, den`$\}$ debe cumplir: `den` $\neq 0$.

- TDA `Fecha`: Dado el objeto *rep* `f`=$\{$`dia, mes, anio`$\}$ debe cumplir:
  - $1 \leq$ `dia` ≤ 31
  - $1 \leq$ `mes` ≤ 12
  - Si `mes` == 4,6,9 u 11, entonces `dia` ≤ 30.
  - Si `mes` == 2 y `bisiesto(anio)`, entonces `dia` ≤ 29.
  - Si `mes == 2 y !bisiesto(anio)`, entonces `dia` ≤ 28.

Indicando la F.A. e I.R.

Tanto la función de abstracción como el invariante de la representación deben **FIGURAR ESCRITOS** en la implementación.

racional.cpp

```
#include "racional.h"

/*
   ** Función de abstracción:
      ----------------------
      fA : tipo_rep  ----> Q
           {num, den} ----> q
      La estructura {num, den} representa al número
       racional q = num/den.

   ** Invariante de Representación:
      ---------------------------
      Cualquier objeto del tipo_rep, {num, den},
      debe cumplir:
       -  den != 0
 */
```

Preservación del I.R.

Es fundamental la conservación del I.R. para todos los objetos modificados por las operaciones que los manipulan. Su conservación se puede establecer demostrando que:

1. Los objetos creados por constructores lo verifican.

2. Las operaciones que modifican los objetos los dejan en un estado que verifica el I.R. antes de finalizar.

Sólo se podrá garantizar esto cuando exista ocultamiento de información.

Representaciones mutables

Un TDA se dice *mutable* cuando los objetos del tipo pueden ser modificados. En caso contrario se dicen *inmutables*.

La mutabilidad es una propiedad del tipo abstracto cuyo cumplimiento debe garantizar la representación.

Depende de que existan operaciones de modificación. Funciones `const` en C++.

Parametrización o Generalización

Parametrización de un tipo de dato consiste en introducir un parámetro en la definición del tipo para poder usarlo con distintos tipos.

Ejemplo: `VectorDinamico T`, donde `T` puede ser `int`, `complejo`, `polinomio`, etc.

Las especificaciones de `VectorDinamico` de `int`, `float`, `Fecha`, etc. son todas iguales salvo por la naturaleza específica del tipo de elementos a incluir en el vector. En este caso, todo es común (especificación, representación e implementación).

En lugar de escribir cada especificación, representación e implementación independientemente se puede escribir una sola de cada, incluyendo uno o varios parámetros que representan *tipos de datos*. (Es el mismo mecanismo de abstracción que hizo surgir el concepto de procedimiento).

Así, en la especificación, representación e implementación de `VectorDinamico` aparecerá `T` en lugar de nombres de tipos concretos.

TDA Genéricos

La especificación de un TDA genérico se hace igual que la de un TDA normal y se añaden aquellos requisitos que deban cumplir los tipos para los que se quiera instanciar. Habitualmente, la existencia de ciertas operaciones.

Ejemplo: Especificación de `VectorDinamico`

```
/**
   VectorDinamico::VectorDinamico, ~VectorDinamico,
   redimensionar, dimension, componente,
   asignar_componente

   Este TDA representa vectores de objetos de la
   clase T cuyo tamaño puede cambiar en tiempo de
   ejecución.  Son mutables.
   Residen en memoria dinámica.

   Requisitos para la instanciación:

   La clase T debe tener las siguientes operaciones:
   - Constructor por defecto
   - Constructor de copia
   - Operador de asignación
  */
```

Parametrización de tipos en C++

El mecanismo que ofrece C++ para parametrizar tipos son los *template* de clases.

Declaración de un `template`:
`template` $<$ *`parámetros`* $>$ *`declaración`*

Los parámetros de la declaración genérica pueden ser:

- `class` *identificador*. Se instancia por un tipo de dato.

- *tipo-de-dato identificador*. Se instancia por una constante.

Clases `template`

```
template<class T, int n>
class array_n {
private:
  T items[n];
};

array_n<complejo,1000> w;
```

La definición de las funciones miembros que se hagan fuera de la clase se escriben así:

```
template <class T>
T VectorDinamico<T>::componente(int i) const
{
  return datos[i];
}
```

Tratamiento de los `templates`

Para usar un tipo genérico hay que instanciarlo, indicando los tipos concretos con que se quiere particularizar. Ejemplos:

```
VectorDinamico<int> vi;
VectorDinamico<float> vf;
```

El compilador generará las definiciones de clases y sus funciones miembro correspondientes para cada instanciación que encuentre. Por ello, la definición completa de la clase genérica debe estar disponible: tanto la definición de la clase como las de las funciones, para que el compilador pueda particularizarlas.

Por ello, el código se organizará como siempre: interfaz en el fichero `.h` e implementación en el fichero `.cpp`. Pero la inclusión será al revés. No se incluirá en el `.h` en el `.cpp`, sino el `.cpp` al final del `.h`.

template: organización del código

VD.h

```
#ifndef __VD_H__
#define __VD_H__

template <class T>
class VectorDinamico {
  ...
};

#include "VD.cpp"
#endif
```

VD.cpp

```
#include <cassert>

template <class T>
VectorDinamico<T>::VectorDinamico (int n)
{
 ...
}
...
```

Tipo abstracto: Clases de operaciones

1. **Constructores primitivos**. Crean objetos del tipo sin necesitar objetos del mismo tipo como entrada.

2. **Constructores**. Crean objetos del tipo a partir de otros objetos del tipo.

3. **Modificadores o mutadores**. Operadores que modifican los objetos del tipo.

4. **Observadores o consultores**. Toman como entrada objetos de un tipo y devuelven objetos de otro tipo.

Es usual la combinación de operadores. En particular, de los tipos 3 y 4.

Diseño de TDAs

1. Elección de la mutabilidad.
   - Debe corresponder con la entidad del mundo real que modelan.
   - Seguridad: tipos inmutables.
   - Eficiencia: tipos mutables.

2. Elección de las operaciones:
   - No hay normas estrictas, sólo guías heurísticas.
   - Debe haber: constructores primitivos, observadores y
     - para inmutables: constructores,
     - para mutables: modificadores.
   - No es un buen criterio la abundancia.
   - En objetos compuestos habrá que incluir iteradores.
   - Dependen del uso del tipo.

Diseño de TDAs (II)

3. Operaciones igual, similar y copia:

   - Igualdad: Son el mismo objeto.
   - Similitud: Representan el mismo objeto.
   - Copia: Son dos objetos que poseen los mismos valores.

4. Elección de la representación:

   - Se elige después de haber hecho la especificación. Existe el riesgo de hacerlo a la inversa.
   - La elección viene condicionada por la sencillez y la eficiencia en las implementaciones de las operaciones del tipo.