10 TIPOS Y ESTRUCTURAS DE DATOS

10.1 Programación y Abstracción

La abstracción es un mecanismo fundamental para la comprensión de fenómenos o situaciones que implican gran cantidad de detalles. La idea de abstracción es uno de los conceptos más potentes en el proceso de resolución de problemas. Se entiende por abstracción la capacidad de manejar un objeto (tema o idea) como un concepto general, sin considerar la enorme cantidad de detalles que pueden estar asociados con dicho objeto. Sin abstracción no sería posible manejar, ni siquiera entender, la gran complejidad de ciertos problemas. Por ejemplo, es muy difícil entender la organización y funcionamiento de una gran empresa multinacional si se piensa en ella en términos de cada trabajador individual (posiblemente miles, distribuidos por todo el mundo) o de cada uno de los productos que fabrica, sin embargo, es más sencilla su comprensión si se ve, simplemente, como una agrupación de departamentos especializados.

En todo proceso de abstracción aparecen dos aspectos complementarios:

(i) destacar los aspectos relevantes del objeto.

(ii) ignorar aspectos irrelevantes del mismo.

La relevancia de los detalles (información) depende del nivel de abstracción considerado, ya que si se pasa a niveles más concretos, es posible que ciertos aspectos pasen a ser relevantes. Se puede decir que la abstracción permite estudiar los fenómenos complejos siguiendo un método jerárquico, es decir, por sucesivos niveles de detalle.

La abstracción implica reducción de información y, por tanto, simplificación del problema tratado.

En el aspecto concreto que nos interesa, la programación, hay que tener en cuenta que los programas son entidades complejas que pueden estar compuestos por miles de instrucciones, cada una de las cuales puede dar lugar a un error del programa y que, por lo tanto, necesitan mecanismos de definición que eviten, en la medida de lo posible, que el programador cometa errores. Así, por ejemplo, los lenguajes de programación de alto nivel permiten al programador abstraerse de la gran cantidad de detalles que es necesario controlar al programar con los lenguajes ensambladores y permiten trabajar de manera independiente respecto a las máquinas sobre las que finalmente se hará funcionar el programa.

La utilización de un lenguaje de programación de alto nivel, supone un elevado nivel de abstracción, ya que al emplear el lenguaje de programación se consigue independizar la solución del problema de las características propias de cada máquina (conjunto de operaciones definidas en cada tipo de procesador). De hecho, es como si el programador siempre utilizase la misma máquina, una máquina que posee como operaciones disponibles las definidas en el lenguaje de programación. Por lo tanto, se puede decir que los lenguajes de programación definen una máquina virtual que es independiente de todas las posibles máquinas reales (soporte físico) sobre las que después funcionaran los programas.

En el proceso de programación se puede extender el concepto de abstracción tanto a las acciones, mediante la llamada abstracción procedimental (del inglés *procedure* = procedimiento), caracterizada por la utilización de subrutinas, como a los datos, mediante los llamados tipos abstractos de datos.

La idea de abstracción procedimental aparece ya en los primeros lenguajes de alto nivel (Fortran o Cobol) a través de la utilización de subrutinas. La aparición de la llamada programación estructurada profundiza más en la descomposición del programa en procedimientos. Los procedimientos permiten generalizar el concepto de operador, de manera que el programador es libre de definirse sus propios operadores y aplicarlos sobre datos que no tienen porque ser necesariamente simples, como hacen habitualmente los constructores de expresiones de los lenguajes de programación.

Los procedimientos permiten encapsular algoritmos (o partes de ellos). De este modo, la abstracción procedimental destaca qué hace el procedimiento (operador) ignorando cómo lo hace. El programa, como usuario de un procedimiento, sólo necesita conocer la especificación de la abstracción (el qué), limitándose a usar el procedimiento con los datos apropiados. Por lo tanto, la abstracción produce un ocultamiento de información y simplifica el proceso de programación.

10.2. Tipos de datos y abstracción

La aplicación a los datos de las ideas de abstracción y de ocultación de información ha tardado más tiempo en producirse si lo comparamos con la abstracción en algoritmos. El concepto de tipo abstracto de datos, propuesto hacia 1974 por John Guttag y otros, vino a desarrollar este aspecto. Análogamente a los procedimientos, los llamados tipos abstractos de datos constituyen un mecanismo que permite generalizar y encapsular los aspectos relevantes sobre la información (datos) que maneja el programa.

Los datos son las propiedades o atributos (cualidades o cantidades) asociados a hechos u objetos y que son procesados por el ordenador. El tipo de datos, en el contexto de un lenguaje de programación, define el conjunto de valores que una determinada variable puede tomar, así como las operaciones básicas sobre dicho conjunto, es decir, definen cómo se representa la información y cómo se interpreta.

Los tipos de datos pueden variar de un lenguaje de programación a otro, tanto los tipos simples como los mecanismos para crear tipos compuestos. Los tipos de datos constituyen un primer nivel de abstracción, ya que no se tiene en cuenta cómo se representa realmente la información sobre la memoria de la máquina, ni cómo se manipula. Para el usuario el proceso de representación es invisible. El programador no manipula directamente las cadenas de bits que constituyen los datos, sino que hace uso de las operaciones previstas para cada tipo de datos. Por ejemplo, el tipo simple ENTERO define un conjunto de valores enteros comprendidos en un determinado intervalo, para los que están definidas las operaciones suma, resta, multiplicación, división entera, asignación, etc. Para el programador es imposible manipular un dato entero si no es a través de las operaciones definidas para ese tipo de datos, cualquier otro proceso de manipulación está prohibido por el lenguaje. De esta manera, al escribirse los programas independientemente de la representación última de los datos en la memoria, si cambiase, por ejemplo, la forma de representar la información en los ordenadores los programas escritos en lenguajes de alto nivel sólo necesitarían ser recompilados para ejecutarse correctamente en las nuevas máquinas.

La memoria del ordenador es una estructura unidimensional (secuencia de elementos) formada por celdas iguales que pueden almacenar números binarios con un número fijo de cifras (bits) [1]. Cada tipo de datos tiene asociada una función de transformación que permite pasar los datos del formato en que se manejan en un programa al formato de la memoria y viceversa. De manera que cambiar la representación en memoria de los datos sólo implica modificar la función de transformación, el programador no ve afectado para nada su trabajo, ya que se encuentra en un nivel superior de abstracción.

Los tipos de datos que un programador utiliza en un lenguaje de alto nivel suelen ser de dos tipos: predefinidos en el lenguaje y definidos por el usuario. Esta última posibilidad contribuye a elevar el nivel del lenguaje, pues permite definir tipos de datos más próximos al problema que se desea resolver. Para ello, el lenguaje suministra constructores genéricos de tipos mediante los cuales el programador puede definir tipos concretos. Sin embargo, en los lenguajes de alto nivel más tradicionales (procedimentales y no orientados a objetos), al programador no se le permite definir cuáles son las operaciones permitidas para los nuevos tipos de datos. En general, los lenguajes suministran unas operaciones predefinidas muy genéricas que, en la mayoría de los casos, no resultan las más apropiadas para el nuevo tipo.

Supóngase, por ejemplo, la definición de un tipo de datos para manipular fechas del calendario, una posible definición en C/C++ sería:

```
struct fecha
{
   int dia;
   int mes;
   int año;
};
```

Se pueden definir variables del nuevo tipo:

```
fecha f1, f2;
```

También se pueden definir subprogramas que operen sobre valores de este tipo y que, de alguna manera, representarían los operadores válidos sobre esos datos. Por ejemplo, en una determinada aplicación se puede considerar adecuado manejar datos del tipo `fecha` mediante las siguientes operaciones (procedimientos o funciones):

Asignar un valor (día, mes y año) válido a una fecha:

```
void asignarFecha (fecha f, int d, int m, int a);
```

Incrementar en un día una fecha:

```
void incrementarFecha (fecha f);
```

Decrementar en un día una fecha:

```
void decrementarFecha (fecha f);
```

Comprobar si una fecha es menor (anterior) que otra:

```
bool menorFecha (fecha f, fecha g);
```

1 El número de cifras almacenadas en cada celda depende de la máquina y es lo que se conoce como longitud de palabra del procesador, normalmente este número puede ir desde 8 bits, para los procesadores más simples, hasta 64 bits o más, en los procesadores más potentes.

Visualizar los datos de una fecha:

```
void visualizarFecha (fecha f);
```

Todas estas operaciones pueden incluir las comprobaciones pertinentes para verificar que todas las fechas manejadas sean válidas (meses de 30 o 31 días, años bisiestos, etc). Sin embargo, no se puede impedir que se generen, mediante otros operadores, valores que no tengan sentido (según la interpretación del programador.) Por ejemplo, hacer:

```
f1.dia = 30;
f1.mes = 2;
/* ¡¡¡día 30 de febrero!!! */
```

o bien,

```
f1.dia = 5 * f2.mes;
```

son operaciones perfectamente válidas desde el punto de vista del lenguaje C/C++, pero carecen de sentido en términos del concepto real de fecha.

Sería conveniente que el lenguaje estuviera dotado de un mecanismo que permitiera definir conjuntamente representaciones de datos y operaciones de manipulación sobre ellos y que ese mecanismo impidiera que los datos se pudieran manejar de manera incorrecta (operaciones no permitidas).

10.3. Clases en *C++*

En C++ la manera en que se puede solucionar el problema de definición de tipos de datos anteriormente descrito es mediante la declaración de tipo usando el concepto de clase.

Una clase en C++ es una especificación de un tipo de datos que contiene, además de la información propia de elemento del tipo (igual que un registro), los procedimientos y funciones propias para manipular 'correctamente' la información contenida en el registro.

El concepto de clase es un elemento propio (y básico) en los llamados lenguajes de programación orientados a objetos.

La palabra reservada para declarar el nuevo tipo de datos 'clase' es **`class`**.

Continuando con el ejemplo del tipo `fecha`, la declaración del nuevo tipo sería:

```
class fecha
{
     public:
          void asignarFecha (int d, int, m, int a);
          void incrementarFecha ();
          void decrementarFecha ();
          bool menorFecha (fecha f);
          void visualizarFecha ();
     private:
          int dia;
```

```
            int mes;
            int anyo;
};
```

Como se puede observar, la declaración incluye tanto la especificación de los datos necesarios para representar una fecha (día, mes y año), como las operaciones que permiten su manipulación.

En la declaración de la clase hay que destacar el significado de las palabras reservadas `public` y `private`. La finalidad de estos elementos del lenguaje es especificar el nivel de visibilidad de los componentes definidos. La palabra `public` permite especificar aquellos componentes (datos y/o operaciones) que está permitido usar "fuera" de la clase, es decir, por cualquier programa o función que declare fechas y desee manipularlas de alguna manera. Por el contrario, la palabra `private` especifica aquellos componentes (también datos y/o operaciones) que se desea que estén protegidos por el lenguaje, de manera que se impide su acceso desde "fuera" de la clase. Sólo las operaciones definidas en la clase pueden tener acceso a estos elementos.

Ejemplo: El siguiente programa hace uso de la clase `fecha`

```
int main ()
{
      fecha f;                          //1
      f.dia = 3;                        //2, error
      f.mes = 5;                        //3, error
      f.anyo = 2100;                    //4, error
      f.visualizarFecha ();             //5
      f.asignarFecha(4,10,2020);        //6
      f.visualizarFecha ();             //7
      return 0;
}
```

Lo primero que hay que destacar es que un dato de tipo `fecha`, como `f`, se declara de igual forma que una variable estándar (línea 1). No obstante, en lugar de variables, estos elementos reciben el nombre de objetos (objeto = dato perteneciente a una clase). Por su lado, el acceso a los elementos definidos en el interior de la clase se realiza mediante la sintaxis típica del constructor `struct`. En la forma: `objeto.componente`, como se puede ver en las líneas 2-7. Las operaciones definidas en la clase siempre deben de ser invocadas a través de un objeto. Por ejemplo, la sentencia:

```
visualizarFecha ();
```

no es válida. Puesto, que está declarada en el interior de la clase `fecha`, debe de ser invocada asociada con un objeto de dicha clase:

```
f.visualizarFecha ();
```

En realidad, se le está pidiendo al objeto `f` que ejecute dicha operación. De ahí que los objetos sean considerados algo más que variables. Las variables se limitan a almacenar información de manera pasiva, sin embargo, los objetos son elementos capaces de realizar acciones y para ello, adicionalmente almacenan información.

No todas las sentencias del programa de ejemplo son correctas. Así, las líneas 2, 3 y 4 darían lugar a errores de compilación, puesto que se está intentando acceder a elementos que han sido declarados como privados a la clase. Este es un mecanismo de protección frente al uso incorrecto de la información. Por su parte, las líneas 5, 6 y 7 muestran cual debería ser la forma correcta de acceder a las fechas, a través de los elementos públicos de la clase.

Como regla general, se establece que: (i) todos los datos declarados en el interior de una clase deben de ser privados a la misma, (ii) deben de ser declaradas como públicas aquellas operaciones a través de las cuales se permitan manipular (correctamente) los objetos que se declaren de esta clase.

La descripción de las clases permite observar claramente los dos niveles que existen en la manipulación de datos en un programa. Por un lado, el proceso de diseño y construcción de los datos y por otro lado, la utilización de estos datos en los programas para describir soluciones a problemas. Durante el primero de los procesos es necesario conocer y tener acceso a todos los componentes declarados en la clase (tanto públicos como privados). Sin embargo, para el uso de objetos de la clase sólo es necesario conocer la especificación de la misma y la forma en que se puede manipular (elementos públicos).

Los tipos de datos se pueden presentar con distinto nivel de detalle (abstracción), desde la especificación formal (tipos abstractos de datos) hasta el nivel de implementación en un lenguaje de programación concreto.

10.3. Tipos Abstractos de Datos

El nivel más alto de abstracción en la representación de los tipos de datos lo constituye la especificación formal de los mismos, dando lugar a los denominados tipos abstractos de datos. Éstos se pueden ver como modelos matemáticos sobre los que se definen una serie de operaciones. El tipo abstracto de datos es independiente del lenguaje de programación, ya que un tipo abstracto de datos (en adelante TAD) es una especificación que no incluye ningún detalle sobre la forma de representación.

Un tipo abstracto de datos se puede definir como una terna de conjuntos (D, O, A), donde D representa el dominio del tipo, es decir, los posibles elementos (valores) que pueden constituirlo; O es el conjunto de operaciones que permiten manipular la información y A es el conjunto de axiomas que describen la semántica (significado) de las operaciones.

La especificación de un tipo de datos (TAD) incluye: el conjunto de valores sobre los que se define, las operaciones válidas para el tipo y la especificación de cual debe ser su comportamiento. En ningún momento se hace referencia al modo en que se deben implementar los valores o las operaciones. Los TAD se pueden ver como los planos que describen los datos y que especifican como deben de ser construidos.

La construcción de un TAD se debe realizar a partir de otros tipos de datos. Algunos de estos tipos de datos pueden ser los que proporcionan los lenguajes de programación, pero otros pueden estar definidos a su vez por otros TAD.

Obviamente, la representación de cualquier tipo de datos va a necesitar almacenar información. Si la cantidad de información es relativamente grande será normal hacer uso de tipos de datos orientados al almacenamiento de información (los lenguajes incorporan algunos constructores de tipos de este estilo, arrays o estructuras). Estos tipos de datos, a menudo denominados *contenedores* o *colecciones* de datos, tienen como operaciones básicas las orientadas al almacenamiento y recuperación de datos (almacenar, recuperar, consultar).

Existen diversas formas de abordar la construcción de los tipos contenedores. Aunque las operaciones son muy similares en todos ellos, pueden diferir sustancialmente en las propiedades de las mismas o en la eficiencia de su implementación:

Cuándo se almacena un dato ¿dónde se ubicará el dato dentro del contenedor? ¿es posible seleccionar la posición que debe ocupar?

Cuando se elimina un dato ¿es posible seleccionar el dato a eliminar? ¿se elimina el dato que ocupa una determinada posición?

¿Es posible acceder a la información de todos los datos almacenados en el contenedor o sólo es posible acceder al dato ubicado en una posición determinada? En el primer caso, ¿Es posible buscar de manera eficiente un dato dentro del contenedor?

La respuesta a estas preguntas y a otras similares da lugar a distintos tipos de contenedores de información. Estos tipos de datos tan especializados han recibido tradicionalmente el nombre de estructuras de datos y constituirán el objeto de estudio hasta el final la asignatura.

Resumiendo, se puede decir que:

Los tipos abstractos de datos especifican las propiedades de los tipos de datos (dominio de definición, operaciones válidas y propiedades de las operaciones), sin ningún tipo de orientación sobre la forma de implementarlos (son los planos que permitirán la construcción). Existen múltiples formas de implantar un mismo TAD, pero todos los TAD se deben implementar utilizando otros tipos de datos (predefinidos o no en el lenguaje de programación).

Las estructuras de datos son tipos de datos orientados al almacenamiento de información (contenedores) que son habitualmente utilizados como soporte para construir otros tipos de datos y que, por tanto, son fundamentales en el proceso de programación. Son tipos de aplicación generalizada a un amplio espectro de aplicaciones informáticas.

10.4. Ejemplo de construcción de tipos con clases en C++

En esta sección se muestra un ejemplo completo de construcción de un tipo de datos utilizando clases en el lenguaje de programación C++.

Se desea construir un tipo de datos fecha a partir de la siguiente especificación del tipo abstracto de datos:

TAD `Fecha`

Dominio:

Conjunto de fechas válidas especificadas por tres valores enteros que identifican el día, mes y año.

Operaciones:

```
asignarFecha (Entero, Entero, Entero ) → Fecha
incrementarFecha (Fecha) → Fecha
decrementarFecha (Fecha) → Fecha
menorFecha (Fecha, Fecha) → Lógico
visualizarFecha (Fecha)
```

Axiomas (propiedades):

La especificación de los axiomas de un tipo se suele hacer, por precisión, siguiendo una notación algebraica. Sin embargo, por simplicidad, ahora se va a realizar una descripción en lenguaje natural de las propiedades que debe cumplir cada operación.

Sea `d,m,a ∈ Entero`:

`asignarFecha (d, m, a)`, debe generar una fecha válida cuyo día sea `d`, su mes sea `m` y su año sea `a`.

Sea `f ∈ Fecha`:

`incrementarFecha (f)`, debe generar la fecha correspondiente a un día posterior a `f`.

Sea `f ∈ Fecha`:

`decrementarFecha (f)`, debe generar la fecha correspondiente a un día anterior a `f`.

Sea `f,g ∈ Fecha`:

`menorFecha (f, g)`, devolverá `cierto` si `f` correspondiente a una fecha anterior a `g` y `falso` en caso contrario.

Sea `f ∈ Fecha`:

`visualizarFecha (f)`, debe imprimir en pantalla los datos de día, mes y año correspondientes a la fecha `f`.

Como se puede observar esta descripción del TAD Fecha no incluye ningún detalle sobre la forma en que se debe implementar el tipo (datos a utilizar o algoritmos a emplear para implementar las operaciones). Una posible implementación vendría dada por la siguiente clase en C++:

Archivo "`fecha.h`": Interfaz de la clase (sólo declaración de prototipos)

```
class Fecha
{
     public:
          //Operaciones descritas en el TAD
          void asignarFecha (int d, int m, int a);
          void incrementarFecha ();
          void decrementarFecha ();
          bool menorFecha (Fecha f);
          void visualizarFecha ();
     private:
          //Detalles de implementación
          //Datos a emplear para la representación
          int dia;
          int mes;
          int anyo;
          //Funciones auxiliares para implementar
          //las operaciones públicas
```

```
            bool fechaValida (int d, int m, int a);
            int diasMes (int m, int a);
            bool bisiesto (int a);
};
```

Archivo "`fecha.cpp`": Implementación de la clase

```
#include "fecha.h"

int Fecha::diasMes (int m, int a)
//método privado que indica los días del mes
{
      int dias;

      switch (m)
      {
            //Meses de 31 días
            case 1: case 3: case 5:
        case 7: case 8: case 10: case 12:
            dias = 31; break;
            //Meses de 30 días
            case 4: case 6: case 9: case 11:
                  dias = 30; break;
            //Febrero puede tener 28 o 29 días
            case 2:
                  if (bisiesto(a)) dias = 29;
                  else dias = 28; break;
            default: //Valor de m no válido
                  dias = 0;
      }
      return (dias) ;
}

bool Fecha::fechaValida (int d, int m, int a)
//Método privado que indica si una fecha es válida o no
{
      bool valida;

      if ((m>=1)&&(m<=12))
            if ((d >= 1)&&(d <= diasMes (m,a)))
              valida = true;
        else
              valida = false;
```

```
      else //Valores no válidos de m
        valida = false;

      return (valida);
}

bool Fecha::bisiesto (int a)
//Método privado que indica si un año es bisiesto
{
  bool b;

  if ((a % 400) == 0)
       b = true;
  else
       if ((a % 100) == 0)
            b = false;
       else
            if ((a % 4) == 0)
                 b = true;
            else
                 b = false;

   return (b);
}

void Fecha::asignarFecha (int d, int m, int a)
//Asigna un día, mes y año a la fecha
{
      //Como especifica el TAD, sólo asigna fechas válidas
      if (fechaValida (d, m, a))
      {
           dia = d;
           mes = m;
           anyo = a;
  }
  else
      cerr << "Error: fecha no valida." << ende;
}

void Fecha::incrementarFecha ()
//Incrementa en un día la fecha
{
```

```
      //En días intermedios del mes sólo es preciso
      //incrementar el valor de dia
      if (fechaValida (dia+1, mes, anyo))
            dia = dia + 1;
      else
            //Si estamos en el último día del mes
            //es necesario pasar al día 1 del mes siguiente
            if (fechaValida (1, mes+1, anyo))
            {
                  dia = 1;
                  mes = mes + 1;
            }
            else
            //Estamos en el 31 de diciembre y es preciso
            //pasar al 1 de enero del siguiente año
            {
                  dia = 1;
                  mes = 1;
                  anyo = anyo + 1;
            }
}

void Fecha::decrementarFecha ()
{
      //En días intermedios del mes sólo es preciso
      //decrementar el valor de dia
      if (fechaValida (dia-1, mes, anyo))
            dia = dia - 1;
      else
      {
            //Si estamos en el primer día del mes
            //es necesario pasar al último día
        //del mes anterior
            int dias_mes;
            dias_mes = diasMes (mes-1, anyo);
            if (fechaValida (dias_mes, mes-1, anyo))
            {
                  dia = dias_mes;
                  mes = mes - 1;
            }
            else
            //Estamos en el 1 de enero y es preciso
```

```
            //pasar al 31 de diciembre del año anterior

            {
                  dia = 31;
                  mes = 12;
                  anyo = anyo - 1;
            }
      }
}

bool Fecha::menorFecha (Fecha f)
//Comprueba si la fecha es menor que f
{
      bool menor = false;

      if (anyo < f.anyo) menor = true;
      else
            if (anyo == f.anyo)
                  if (mes < f.mes) menor =true;
                  else
                        if (mes == f.mes)
                              if (dia < f.dia) menor = true;
      return (menor);
}

void Fecha::visualizarFecha ()
//Muestra por pantalla la fecha
{
      string nomMes;

      switch (mes)
      {
            case 1: nomMes = "Enero"; break;
            case 2: nomMes = "Febrero"; break;
            case 3: nomMes = "Marzo"; break;
            case 4: nomMes = "Abril"; break;
            case 5: nomMes = "Mayo"; break;
            case 6: nomMes = "Junio"; break;
            case 7: nomMes = "Julio"; break;
            case 8: nomMes = "Agosto"; break;
            case 9: nomMes = "Septiembre"; break;
            case 10: nomMes = "Octubre"; break;
```

```
            case 11: nomMes = "Noviembre"; break;
            case 12: nomMes = "Diciembre";
        }
        cout << dia << " de ";
        cout << nomMes << " de ";
        cout << anyo;
}
```