10. TIPOS DE DATOS, ESTRUCTURAS DE DATOS Y TIPOS ABSTRACTOS DE DATOS

1.1. 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 objetos 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 irrelevancia depende del nivel de abstracción, 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. Generalmente, en la especificación se sigue un sentido descendente, desde los niveles más generales a los niveles más concretos.

En el aspecto concreto que nos interese, la programación, hay que tener en cuenta que los programas son objetos complejos 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, para especificar el conjunto de operaciones que debe realizar la máquina para resolver un determinado problema, 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 procedural (utilización de subprogramas), como a los datos, mediante los llamados tipos abstractos de datos.

La idea de abstracción procedural 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 partes de un algoritmo (modularizar), localizando en una sección del programa aquellas proposiciones relacionadas con cierto aspecto del mismo. De este modo, la abstracción procedural 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 problema en ese nivel de abstracción.

1.2. 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. 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.

Hay que tener en cuenta que no se deben confundir el concepto de tipo de datos con los de estructura de datos y tipo abstracto de datos. Todos ellos constituyen diferentes niveles en el proceso de abstracción referida a los datos.

Los datos son las propiedades o atributos (cualidades o cantidades) sobre hechos u objetos que procesa el ordenador. El tipo de datos, en un lenguaje de programación, define el conjunto de valores que una determinada variable puede tomar, así como las operaciones básicas sobre dicho conjunto. 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 (en el caso de Pascal determinado por una constante predefinida, MAXINT), para los que están definidas las operaciones suma, resta, multiplicación y división entera. 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 (si, por ejemplo, dejaran de trabajar en base dos y pasaran a hacerlo en base tres), 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

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.

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. Además, la posibilidad de especificar tipos estructurados introduce la idea de generalidad. 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, 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++ 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, pero, sin embargo, no se puede impedir que se generen, mediante otros operadores, valores que no tienen 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. Sin embargo, carecen de significativo en términos del concepto real de fecha.

Sería conveniente definir operaciones específicas, y válidas, sobre el tipo fecha, más allá de las operaciones heredadas por los tipos de datos componentes, que nos aseguraran que los valores que se obtienen son perfectamente válidos. Así sería interesante definir una operación de asignación que a partir de unos ciertos valores para un día, un mes y un año nos asegurase que la fecha introducida en la estructura fuera correcta.

Entre el nivel de los tipos de datos y el nivel de los tipos abstractos se suele ubicar el concepto de estructuras de datos. Las estructuras de datos son agrupaciones de datos, quizás de distinta naturaleza (tipo), relacionados (conectados) entre sí de diversas formas y las operaciones definidas sobre esa agrupación. Ejemplos de estructuras las podemos encontrar en muchos ámbitos, desde las matemáticas (estructuras algebraicas: grupo, anillo o cuerpo) hasta el mundo de los negocios (estructura organizativa de una empresa). Los elementos de una estructura de datos, en última instancia, dependerán del lenguaje de programación a través de los tipos de datos que los definan. Sin embargo,

el concepto de estructura en sí, que está definida por el tipo de relación entre los elementos, no depende del lenguaje de programación empleado.

Las estructuras de datos se caracterizan por el tipo de los elementos de la estructura, las relaciones definidas sobre los elementos y las operaciones permitidas sobre la estructura. Operaciones típicas sobre estructuras de datos suelen ser: buscar y acceder a los elementos (por la posición que ocupan en la estructura o por la información que contienen), insertar o borrar elementos, modificar las relaciones entre los elementos, etc.

Al nivel de las estructuras de datos, ya no tiene relevancia las operaciones que se puedan realizar sobre los componentes individuales de la misma, solamente la tienen las operaciones que implican a la estructura globalmente.

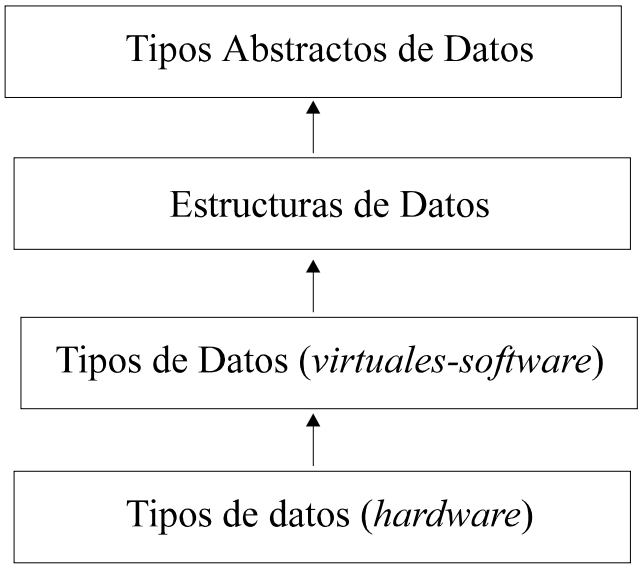
1.3. Tipos Abstractos de Datos

En el nivel más alto de abstracción estarían los tipos abstractos de datos, éstos se pueden ver como modelos matemáticos sobre los que se definen una serie de operaciones. En realidad son una formalización del concepto de dato que el programador considera adecuado para resolver el problema. El tipo abstracto de datos, al igual que las estructuras de datos, es independiente del lenguaje de programación, ya que un tipo abstracto de datos (en adelante TAD) es una colección de valores y de operaciones que se definen mediante una especificación que es independiente de cualquier 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 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.

Para representar el modelo matemático básico de un TAD se emplean estructuras de datos y las operaciones definidas en el modelo se representan mediante procedimientos. Por ejemplo, se podría definir el TAD número complejo como un objeto con dos componentes que son números reales y sobre el que se define las operaciones suma y producto de números complejos. En una definición de este tipo no importa si en un programa concreto se utiliza una estructura tipo registro (record en Pascal, structure en otros lenguajes, p.ej. C) con dos campos para representar los datos o bien una estructura tipo array con dos elementos. Desde el punto de vista formal ese tipo de información es irrelevante. Así mismo, no es necesario conocer cómo se realizan las operaciones definidas formalmente, sólo importa qué hacen para poder usarlas correctamente.

Para resumir finalmente la idea y los diferentes niveles de abstracción en lo referente a los datos podemos recurrir a la siguiente figura:



1.3.1. Tipos Abstractos de Datos en C++: Clases

Hemos visto que un problema de las estructuras de datos en los lenguajes es que no se puede restringir la aplicación de las operaciones básicas heredadas de los tipos fundamentales que componen las estructuras. En C++ la manera en que se puede solucionar este problema es mediante la utilización de las clases.

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

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

> *Si seguimos con el ejemplo que tenemos de fechas la cabecera de la declaración del nuevo tipo será:*
>
> ```
> class Fecha
> ```
>
> *Y la información contenida en la clase será la que ya propusimos en el registro: tres enteros para guardar el dia, el mes y el año.*
>
> *Habrá que añadir qué operaciones creemos que deberiamos tener para manipular las fechas.*
>
> *Vimos que la asignación directa de valores a los campos del registro podía producir valores incorrectos de fechas. sería lógico pensar en tener una función que sólo pusiese valores 'correctos' en el registro:* IniciarFecha.
>
> *También sería lógico poder recuperar la información para tenerla disponible en variables:* RecuperarFecha.
>
> *Finalmente, también es habitual comparar valores de fechas para saber si un cierto evento se produjo antes o después de otro:* IgualFecha, MayorFecha, MenorFecha.

Cuando pensemos en el diseño de una clase en C++, vamos a tener en cuenta básicamente dos niveles: Nivel programador de la clase y nivel usuario de la clase.

Nivel programador de la clase: Pensemos en este nivel, como el nivel que tiene que cuidar que la clase funcione correctamente y guarde la información deseada.

Nivel usuario de la clase: El usuario de la clase sólo necesita poder manipular adecuadamente la clase, no va a tener que preocuparse ni de cómo se guarda la información en la clase, ni de cómo se realizan las operaciones. Sólo le va a importar qué información es capaz de guardar la clase y las funciones válidas para manipular esa información.

Así en las clases de C++ se definen dos apartados: apartado público, y apartado privado.

En la parte pública de la clase, se situarán aquellas facetas de la clase que deseemos que sean conocidas y utilizadas por el usuario. En general se situarán en la parte pública de la clase, las funciones (o métodos) de manipulación de la clase.

En nuestro ejemplo, situaremos en la parte pública las funciones que hemos decidido:

```
public:
   void IniciarFecha (int, int, int);
   void RecuperarFecha (int &, int &, int &);

   bool IgualFecha (Fecha);
   bool MayorFecha (Fecha);
   bool MenorFecha (Fecha);
```

En la parte privada situaremos los elementos que no deseemos que sean manipulados (ni incluso conocidos) por el usuario, así como aquellas funciones que como programadores de la clase, deseemos utilizar pero que no sean accesibles por el usuario.

En la parte privada de la clase fecha situaremos básicamente la información que va a contener la clase:

```
private:
   int dia;
   int mes;
   int anyo;
```

Así cuando instanciemos (declaremos) un nuevo objeto (una nueva variable) de este nuevo tipo de dato tendremos en el objeto, no sólo la información propia del objeto, sino también los métodos para manipular esa información. Pero el usuario sólo podrá acceder a aquellas partes del objeto que el programador haya decidido poner como publicas. Mientras que desde la clase, el programador podrá acceder a cualquier elemento que esté en la clase.

Ejemplo: Clase Fecha.

fecha.h

```
class fecha
{
   public:
      bool fecha::IniciarFecha (int dia, int mes, int anyo);
      void fecha::DeterminarFecha (int & dia, int & mes, int & anyo);
      void fecha::EscribirFecha (void);
      void fecha::IgualFecha (fecha fec);
      void fecha::MayorFecha (fecha fec);
      void fecha::MenorFecha (fecha fec);

   private:
      int day, mon, yea;
      bool fecha::Bisiesto (int anyo);
};
```

fecha.cpp

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

bool fecha::Bisiesto (int anyo)
{
   bool b_aux;

   if ( (anyo % 400) == 0)
      b_aux = true;
   else
      if ( (anyo % 100) == 0)
         b_aux = false;
      else
         if ( (anyo % 4) == 0)
            b_aux = true;
         else
            b_aux = false;

   return b_aux;
}

bool fecha::IniciarFecha (int dia, int mes, int anyo)
{
   bool error;

   if ( (dia <= 0) || (mes <= 0) )
      error = true;
   else
   {
      if ( ( (mes == 1) || (mes == 3) || (mes == 5) || (mes == 7)
         || (mes == 8) || (mes == 10) || (mes == 12) ) && (dia <= 31) )
         error = false;
      else
      {
         if ( ( (mes == 4) || (mes == 6) || (mes == 9) || (mes == 11) ) &&
            (dia <= 30) )
            error = false;
         else
         {
            if ( (bisiesto (anyo) ) && (dia <= 29) )
               error = false;
```

```
            else
            {
               if (dia <= 28)
                  error = false;
               else
                  error = true;
            }
         }
      }
   }

   if (!error)
   {
      day = dia;
      mon = mes;
      yea = anyo;
   }

   return error;
}

void fecha::RecuperarFecha (int & dia, int & mes, int & anyo)
{
   dia = day;
   mes = mon;
   anyo = yea;
}

void fecha::EscribirFecha (void)
{
   cout << "fecha: " << day << "/" << mon << "/" << yea << " ";
}

void fecha::IgualFecha (fecha fec)
{
   return (day == fec.day) && (mon == fec.mon) && (yea == fec.yea);
}

void fecha::MayorFecha (fecha fec)
{
   bool b_aux;

   if (yea > fec.yea)
      b_aux = true;
   else
   {
      if ( (yea == fec.yea) && (mon > fec.mon) )
         b_aux = true;
      else
      {
         if ( (yea == fec.yea) && (mon == fec.mon) && (day > fec.day) )
            b_aux = true;
         else
            b_aux = false;
      }
   }

   return b_aux;
}

void fecha::MenorFecha (fecha fec)
{
   bool b_aux;
```

```
   if (yea < fec.yea)
      b_aux = true;
   else
   {
      if ( (yea == fec.yea) && (mon < fec.mon) )
         b_aux = true;
      else
      {
         if ( (yea == fec.yea) && (mon == fec.mon) && (day < fec.day) )
            b_aux = true;
         else
            b_aux = false;
      }
   }

   return b_aux;
}
```

Con estos ficheros terminaría la tarea del programador de la clase.

Un posible ejemplo de utilización de la clase, podría ser el siguiente programa que utiliza la clase fecha, para ordenar por quick_sort un vector de fechas.

```
#include <iostream.h>
#include <fstream.h>
#include <string>

#include "fecha.h"

typedef Fecha vector_fechas[500];

struct Vector
{
   vector_fechas info;
   int num_fechas;
};

bool LeerFicheroFechas (Vector &);
void OrdenarFechas (Vector &);
void Q_S_R(vector_fechas, int, int);
void MostrarFechas (Vector);

int main (void)
{
   Vector vect;

   if (LeerFicheroFechas (vect) == false)
   {
      OrdenarFechas (vect);
      MostrarFechas (vect);
   }
   else
      cout << "Error leyendo el fichero.\n";
}

bool LeerFicheroFechas (Vector & vect)
{
   ifstream f_in;
   bool error;
   int dia, mes, anyo;
   string s_aux;

   f_in.open ("fechas.dat");
```

```
   if (!f_in)
      error = true;
   else
   {
      error = false;
      vect.num_fechas = 0;
      while (f_in)
      {
         f_in >> dia >> mes >> anyo;
         if (vect.info[vect.num_fechas].IniciarFecha (dia, mes, anyo) )
         {
            cout << "Error. La fecha: " << dia << "/" << mes << "/" << anyo;
            cout << " es incorrecta.\n";
            cout << "Pulsa enter para seguir";
            getline (cin, s_aux);
         }
         else
            vect.num_fechas++;
      }
   }
   return error;
}

void OrdenarFechas (Vector & vect)
{
   Q_S_R (vect.info, 0, vect.num_fechas);
}

void MostrarFechas (Vector vect)
{
   int i;

   for (i = 0; i < vect.num_fechas; i++)
   {
      vect.info[i].EscribirFecha();
      if ( (i % 3) == 0 )
         cout << "\n";
      else
         cout << "\t";
   }
}

/*
 *  Funci¢n de ordenaci¢n por Quick-Sort
 */

void Q_S_R(vector_fechas vect, int izda, int dcha)
{
   int i, k;
   int dia_aux, mes_aux, anyo_aux;
   Fecha f_aux;
   Fecha pivote;

   i = izda;
   k = dcha;

   vect[(i + k) / 2].DeterminarFecha (dia_aux, mes_aux, anyo_aux);
   pivote.IniciarFecha (dia_aux, mes_aux, anyo_aux);

   do
   {
      while (vect[i].MenorFecha (pivote) )
         i++;
```

```
            while (vect[k].MayorFecha (pivote) )
               k--;

            if (i <= k)
            {
               vect[i].DeterminarFecha (dia_aux, mes_aux, anyo_aux);
               f_aux.IniciarFecha (dia_aux, mes_aux, anyo_aux);

               vect[k].DeterminarFecha (dia_aux, mes_aux, anyo_aux);
               vect[i].IniciarFecha (dia_aux, mes_aux, anyo_aux);

               f_aux.DeterminarFecha (dia_aux, mes_aux, anyo_aux);
               vect[k].IniciarFecha (dia_aux, mes_aux, anyo_aux);

               i++;
               k--;
            }
         }
         while(i <= k);

         if (izda < k)
            Q_S_R (vect, izda, k);

          if (i < dcha)
             Q_S_R (vect, i, dcha);

          return;
      }
```