Clase #3

Programación Procedural

. La idea básica de esta aproximación es la de definir los algoritmos o procedimientos más eficaces para tratar los datos de nuestro problema.

Tipos de datos

Cuando nos planteamos la resolución de problemas mediante computador lo más usual es que queramos tratar con datos que son variables y cuantificables, es decir, que toman un conjunto de valores distintos entre un conjunto de valores posibles, además de poder almacenar los valores de estos datos en alguna forma aceptable para el computador (ya sea en la memoria o en periféricos de almacenamiento externo).

En un lenguaje de programación el concepto de *tipo de datos* se refiere al conjunto de valores que puede tomar una variable. Esta idea es similar a la que se emplea en matemáticas, donde clasificamos las variables en función de determinadas características, distinguiendo entre números enteros, reales o complejos. Sin embargo, en matemáticas, nosotros somos capaces de diferenciar el tipo de las variables en función del contexto, pero para los compiladores esto resulta mucho más difícil. Por este motivo debemos declarar explícitamente cada variable como perteneciente a un tipo. Este mecanismo es útil para que el computador almacene la variable de la forma más adecuada, además de permitir verificar que tipo de operaciones se pueden realizar con ella.

Se suelen diferenciar los tipos de datos en varias categorías:

— *Tipos elementales*, que son aquellos cuyos valores son atómicos y, por tanto, no pueden ser descompuestos en valores más simples. Entre las variables de estos tipos siempre encontramos definidas una serie de operaciones básicas: asignación de un valor, copia de valores entre variables y operaciones relacionales de igualdad o de orden (por lo tanto, un tipo debe ser un conjunto *ordenado*).

Los tipos más característicos son:

booleanos = {verdadero, falso}

enteros = {… -2, -1, 0, +1, +2, …}

reales = {… -1.0, …, 0.0, …, +1.0, …}

caracteres = {… 'a', 'b', …, 'Z', …}

El tipo cadena de caracteres es un caso especial de tipo de datos, ya que algunos lenguajes lo incorporan como tipo elemental (con un tamaño fijo), mientras que en otros lenguajes se define como un vector de caracteres (de longitud fija o variable) que es una estructura indexada. Como se ve, los campos de un registro pueden ser de otros tipos compuestos, no sólo de tipos elementales.

— *Tipos recursivos*, que son un caso especial de tipos compuestos, introduciendo la posibilidad de definir un tipo en función de sí mismo.

Para terminar nuestro análisis de los tipos de datos, hablaremos de la forma en que los lenguajes de programación relacionan las variables con sus tipos y las equivalencias entre distintos tipos.

Hemos dicho que el empleo de tipos de datos nos sirve para que el computador almacene los datos de la forma más adecuada y para poder verificar las operaciones que hacemos con ellos. Sería absurdo que intentáramos multiplicar un carácter por un real pero, si el compilador no comprueba los tipos de los operandos de la multiplicación, esto sería posible e incluso nos devolvería un valor, ya que un carácter se representa en binario como un número entre 0 y 255. Para evitar que sucedan estas cosas los compiladores incorporan mecanismos de *chequeo de tipos*, verificando antes de cada operación que sus operandos son de los tipos esperados. El chequeo se puede hacer estática o dinámicamente. El chequeo estático se realiza en tiempo de compilación, es decir, antes de que el programa sea ejecutable. Para realizar este chequeo es necesario que las variables y los parámetros tengan tipos fijos, elegidos por el programador. El chequeo dinámico se realiza durante la ejecución del programa, lo que permite que las variables puedan ser de distintos tipos en tiempo de ejecución.

Por último, señalaremos que existe la posibilidad de que queramos realizar una operación definida sobre un tipo de datos, por ejemplo reales, aplicada a una variable de otro tipo, por ejemplo entera. Para que podamos realizar esta operación debe de existir algún mecanismo para compatibilizar los tipos, convirtiendo un operando que no es del tipo esperado en éste, por ejemplo, transformando un entero en real, añadiendo una parte decimal nula, o transformando un real en entero por redondeo.

Operadores y expresiones

Como ya hemos dicho con los datos de un tipo podemos realizar determinadas operaciones, pero, ¿cómo las expresamos en un lenguaje de programación? Para resolver este problema aparecen lo que llamamos *operadores*. Podemos decir que un *operador* es un símbolo o conjunto de símbolos que representa la aplicación de una función sobre unos operandos. Cuando hablamos de los operandos no sólo nos referimos a variables, sino que hablamos de cualquier elemento susceptible de ser evaluado en alguna forma. Por ejemplo, si definimos una *variable entera* podremos aplicarle operadores aritméticos (+, -, *, /), de asignación (=) o relacionales (>, <, …), si definimos una *variable compuesta* podremos aplicarle un operador de campo que determine a cual de sus componentes queremos acceder, si definimos un *tipo de datos* podemos aplicarle un operador que nos diga cual es el tamaño de su representación en memoria, etc.

Los operadores están directamente relacionados con los tipos de datos, puesto que se definen en función del tipo de operandos que aceptan y el tipo del valor que devuelven. En algunos casos es fácil olvidar esto, ya que llamamos igual a operadores que realizan operaciones distintas en función de los valores a los que se apliquen, por ejemplo, la

división de enteros no es igual que la de reales, ya que la primera retorna un valor entero y se olvida del resto, mientras que la otra devuelve un real, que tiene decimales.

Un programa completo está compuesto por una serie de *sentencias*, que pueden ser de distintos tipos:

— *declarativas*, que son las que empleamos para definir los tipos de datos, declarar las variables o las funciones, etc., es decir, son aquellas que se emplean para definir de forma explícita los elementos que intervienen en nuestro programa,

— *ejecutables*, que son aquellas que se transforman en código ejecutable, y

— *compuestas*, que son aquellas formadas de la unión de sentencias de los tipos anteriores.

Llamaremos *expresión* a cualquier sentencia del programa que puede ser evaluada y devuelve un valor. Las expresiones más simples son los *literales*, que expresan un valor fijo explícitamente, como por ejemplo un número o una cadena de caracteres. Las expresiones compuestas son aquellas formadas por una secuencia de términos separados por operadores, donde los términos pueden ser literales, variables o llamadas a funciones (ya que devuelven un resultado).

Algoritmos y estructuras de control

Podemos definir un *algoritmo* de manera general como un conjunto de operaciones o reglas bien definidas que, aplicadas a un problema, lo resuelven en un número finito de pasos. Si nos referimos sólo a la informática podemos dar la siguiente definición:

> Un *procedimiento* es una secuencia de instrucciones que pueden realizarse mecánicamente. Un *procedimiento* que siempre termina se llama *algoritmo*.

Al diseñar algoritmos que resuelvan problemas complejos debemos emplear algún método de diseño, la aproximación más sencilla es la del *diseño descendente* (top-down). El método consiste en ir descomponiendo un problema en otros más sencillos (subproblemas) hasta llegar a una secuencia de instrucciones que se pueda expresar en un lenguaje de alto nivel. Lo que haremos será definir una serie de acciones complejas y dividiremos cada una en otras más simples. Para controlar el orden en que se van desarrollando las acciones, utilizaremos las *estructuras de control*, que pueden ser de distintos tipos:

— *condicionales o de selección*, que nos permiten elegir entre varias posibilidades en función de una o varias condiciones,

— *de repetición (bucles)*, que nos permiten repetir una serie de operaciones hasta que se verifique una condición o hayamos dado un número concreto de vueltas, y

— *de salto*, que nos permiten ir a una determinada línea de nuestro algoritmo directamente.

Funciones y procedimientos

En el punto anterior hemos definido los algoritmos como procedimientos que siempre terminan, y procedimiento como una secuencia de instrucciones que pueden realizarse mecánicamente, aquí consideraremos que un *procedimiento* es un algoritmo que recibe unos *parámetros de entrada*, y una *función* un *procedimiento* que, además de recibir unos parámetros, devuelve un valor de un tipo concreto. En lo que sigue emplearé los términos procedimiento y función indistintamente. Lo más importante de estas abstracciones es saber cómo se pasan los parámetros, ya que según el mecanismo que se emplee se podrá o no modificar sus valores. Si los parámetros se pasan *por valor*, el procedimiento recibe una copia del valor que tiene la variable parámetro y por lo tanto no puede modificarla, sin embargo, si el parámetro se pasa *por referencia*, el procedimiento recibe una referencia a la variable que se le pasa como parámetro, no el valor que contiene, por lo que cualquier consulta o cambio que se haga al parámetro afectará directamente a la variable.

¿Por qué surgieron los procedimientos y las funciones? Sabemos que un programa según el paradigma clásico es una colección de algoritmos, pero, si los escribiéramos todos seguidos, nuestro programa sería ilegible. Los procedimientos son un método para ordenar estos algoritmos de alguna manera, separando las tareas que realiza un programa. El hecho de escribir los algoritmos de manera independiente nos ayuda a aplicar el diseño descendente; podemos expresar cada subproblema como un procedimiento distinto, viendo en el programa cual ha sido el refinamiento realizado. Además, algunos procedimientos se podrán reutilizar en problemas distintos.

Por último, indicaremos que el concepto de procedimiento introduce un nivel de abstracción importante en la programación ya que, si queremos utilizar un procedimiento ya implementado para resolver un problema, sólo necesitamos saber cuáles son sus parámetros y cuál es el resultado que devuelve. De esta manera podemos mejorar o cambiar un procedimiento sin afectar a nuestro programa, siempre y cuando no cambie sus parámetros, haciendo mucho más fácil la verificación de los programas, ya que cuando sabemos que un procedimiento funciona correctamente no nos debemos volver a preocupar por él.

Constantes y variables

En los puntos anteriores hemos tratado las variables como algo que tiene un tipo y puede ser pasado como parámetro, pero no hemos hablado de cómo o dónde se declaran, de cómo se almacenan en memoria o de si son accesibles desde cualquier punto de nuestro programa.

Podemos decir que un programa está compuesto por distintos bloques, uno de los cuales será el principal y que contendrá el procedimiento que será llamado al comenzar la ejecución del programa. Serán bloques el interior de las funciones, el interior de las estructuras de control, etc.

Diremos que el *campo* o *ámbito* de un identificador es el bloque en el que ha sido definido. Si el bloque contiene otros bloques también en estos el identificador será válido. Cuando hablo de identificador me refiero a su sentido más amplio: variables, constantes, funciones, tipos, etc. Fuera del *ámbito* de su definición ningún identificador tiene validez.

Clasificaremos las variables en función de su ámbito de definición en *globales* y *locales*. Dentro de un bloque una variable es local si ha sido definida en el interior de este, y es global si se ha definido fuera del bloque, pero podemos acceder a ella.

Como es lógico las variables ocupan memoria, pero, como sólo son necesarias en el interior de los bloques donde se definen, durante la ejecución del programa serán creadas al entrar en su ámbito y eliminadas al salir de él. Así, habrá variables que existirán durante todo el programa (si son globales para todos los bloques) y otras que sólo existan en momentos muy concretos. Este mecanismo de creación y destrucción de variables permite que los programas aprovechen al máximo la memoria que les ha sido asignada.

Para terminar, sólo diré que existen variables constantes (que ocupan memoria). Son aquellas que tienen un tipo y un identificador asociado, lo que puede ser útil para que se hagan chequeos de tipos o para que tengan una dirección de memoria por si algún procedimiento requiere un puntero a la constante.

PROGRAMACIÓN MODULAR

Con la programación procedural se consigue dar una estructura a los programas, pero no diferenciamos realmente entre los distintos aspectos del problema, ya que todos los algoritmos están en un mismo bloque, haciendo que algunas variables y procedimientos sean accesibles desde cualquier punto de nuestro programa.

Para introducir una organización en el tratamiento de los datos apareció el concepto de *módulo*, que es un conjunto de procedimientos y datos interrelacionados. Aparece el denominado *principio de ocultación de información*, los datos contenidos en un módulo no podrán ser tratados directamente, ya que no serán accesibles desde el exterior del mismo, sólo permitiremos que otro módulo se comunique con el nuestro a través de una serie de procedimientos públicos definidos por nosotros. Esto proporciona ventajas como poder modificar la forma de almacenar algunos de los datos sin que el resto del programa sea alterado o poder compilar distintos módulos de manera independiente. Además, un módulo bien definido podrá ser reutilizado y su depuración será más sencilla al tratarlo de manera independiente.