**TEMA 14: Tipos abstractos de datos.**

En este tema veremos:

* Introducción a los TADs
* Declaración de un TAD, implementación de las funciones dentro de un TAD y declaración de elementos ocultos en un TAD.
* Tratamiento de excepciones (**try , catch, throw**);
* Concepto de programación orientada a objetos.

1. **Concepto de tipo de abstracto de datos (TADs).**

Si analizamos que necesita un programa para poder manejar datos de un tipo, encontramos:

* Hacer referencia al tipo en sí, mediante un nombre, para poder definir variables, subprogramas…etc.

**int** entero; 🡪 Nombre de tipo “int”;

* Hacer referencia a valores particulares, generalmente constantes con nombre.
* Invocar operaciones de manipulación de los valores de ese tipo, bien usando operadores, expresiones aritméticas o invocación de sub-programas.

El conjunto de todas estas características los denominamos tipo abstracto de datos, podemos definirlo como una agrupación de una colección de valores y una colección de operaciones de manipulación de dichos valores.

La colección de valores es finita, y está limitada a lo que se haya declarado. Además, como se representan los valores y como se opera con ellos pueden estar ocultos para quien utiliza el TAD.

La programación orientada a objetos, se basa esencialmente en el uso de tipos abstractos de datos, con algunas modificaciones, por ejemplo, en la terminología, se habla de clases y objetos en lugar de tipos y datos, y de métodos en lugar de operaciones. Se añade el concepto de herencia.

1. **Realización de un TAD en C+-.**

Las clases son implícitamente TADs. En C+-la definición de clases está limitada a el uso de tipos registro(**struct**).

**DEFINICIÓN DE TIPOS ABSTRACTOS COMO TIPOS REGISTRO(STRUCT)**

En los tipos registro(**struct**) además de permitir definir estructuras con varios campos con nombre y tipo individual, también permite definir sub-programas y se puede distinguir entre elementos públicos y privados. Así con estas nuevas características añadidas podemos definir tipos abstractos de datos.

En **notación BFN** podemos designar las reglas de sintaxis de la siguiente manera:

Tipo\_struct::= **typedef struct** identificador

{Lista\_de\_items[private: Lista\_de\_items]};

Lista\_De\_items ::= Item;{Item;}

Ítem::= Campo | Cabecera\_Subprograma

Campo::= Campos\_igual\_tipo | Campo\_puntero | Campo\_array

Campos\_igual\_tipo ::= Identificar\_tipo identificadores;

Campo\_puntero::= Identificador\_tipo\* Idenficador;

Campo\_array ::= Identificador\_de\_tipo Identificador[Dimensiones];

Como ejemplo práctico:

**typedef** **struct** TipoPunto {

**float** x;

**float** y;

/\*\* Leer un punto con formato”(x,y)”\*/

**void** Leer();

**void** Escribir();

**float** Distancia(TipoPunto p);

};

Los sub-programas dentro de un TAD se declaran únicamente poniendo la cabecera. Para referirse a las funciones declaradas dentro del TAD, basta con hacerlo de manera similar a como se invocaban los campos de datos. Con el nombre de la estructura, seguida de un ‘.’ y con el nombre de la función que se quiera invocar. Por ejemplo, siguiendo en el código anterior:

Variable.campo 🡪 Referencia a campo de datos.

Variable.operacion(argumentos); 🡪 Referencia a operación.

TipoPunto p,g; 🡪 Inicializamos dos TADs tipo punto.

p.x = 3.3;

p.y = 4.4;

p.Leer();

p.Escribir();

printf(“%f”, p.Distancia ( q ));

Además de declarar la interfaz de cada función dentro del registro(struct), también hay que definir la implementación de dicha función, haciéndola fuera de la declaración del Registro, usando la notación:

**void** Nombre-Registro::Nombre-Funcion (argumentos){

codigo;

}

También cabe destacar que si trabajamos en la implementación de una función dentro del TAD, los campos de datos dentro del TAD no hace falta enviarlos a las funciones dentro del TAD como argumento, basta con mencionarlas en las sentencias que se quieran utilizar dichos campos.

**OCULTACIÓN**

Para que un tipo de dato sea totalmente abstracto, la implementación de las funciones no tendría que ser visible, los subprogramas, como mecanismo de abstracción, ya ocultan los detalles de la realización de las operaciones.

Si se quieren ocultar los detalles de representación interna de los valores de un TAD, se pueden definir ciertos elementos como privados, usando la palabra clave **private** para delimitar una zona de declaraciones privadas dentro de la estructura.

Podemos verlo como que los campos de datos o funciones declaradas en la zona (**private**) no podrán ser invocados en programas o sub-programas externos al TAD. Como ejemplo :

**typedef** **struct** MiTadFecha {

**int** dia,mes,anno;

**void** LeerFecha();

**void** ImprimirFecha();

**private**:

**bool** ComprobarFecha(); 🡪 Esta función solo la puede invocar los subprogramas LeerFecha o ImprimirFecha dentro del TAD.

**bool** Variable; 🡪 Esta variable solo podrá ser utiliza por funciones dentro del TAD

}

1. **Metodología Basada en abstracciones.**

La técnica de programación estructurada de refinamientos sucesivos puede ampliarse para contemplar la descomposición modular de un programa. Ahora disponemos de un nuevo tipo de abstracción, que son los tipos abstractos de datos. Habrá que ir realizando simultáneamente las siguientes actividades:

* Identificar las operaciones a identificar y descomponerlas en operaciones simples. (refinarlas)
* Identificar las estructuras de información y simplificarlas.

**DESARROLLO DE REFINAMIENTO BASADO EN ABSTRACCIONES**

El desarrollo descendente con **abstracciones** se basa en que cada vez que decidamos refinar (descomponer al complejo en algo simple) tenemos que tomar una de las siguientes decisiones:

# ABSTRACCIONES FUNCIONALES

* Considerar una operación como terminal (operación simple) y codificarla mediante sentencias del lenguaje de programación.
* Considerar la operación como una operación compleja, y descomponerla en otras más simples (operaciones terminales).
* Considerarla como una operación abstracta, y especificarla, escribiendo más adelante al subprograma que la realiza.

# ABSTRACCIONES ESTRUCTURA DE DATOS

* Considerar un dato como elemental y usar directamente un tipo predefinido.
* Considerar un dato como un dato complejo y descomponer en otros más sencillos (registro, unión o formación).
* Considerar un dato como un dato abstracto y especificar su interfaz, dejando para más adelante los detalles de su implementación.