

Contenido

1. TAD (TIPO ABSTRACTO DE DATOS)	2
1.1. Concepto	2
Definiciones:.....	2
1.1. Especificación de un TAD	3
1.1. Interfaz Pública del TAD:	5
1.2. Ventajas del uso de TADs	6
1.3. Abstracción.....	6
1.4. Encapsulamiento.....	9
1.5. Interfaz e Implementación.....	10
2. Bibliotecas o Librerías	11
2.1. Cómo crear una Biblioteca o Librería.....	12
2.2. USO DE UN TAD EN OTRO PROYECTO.....	15
2.3. Un Ejemplo: TAD LISTA.....	18
3. Bibliografía	19
ANEXO 1	20
Bibliotecas o Librerías estáticas	20
Bibliotecas o Librerías dinámicas	20
Diferencias entre bibliotecas Estáticas y Dinámicas	21
Bibliotecas Estándares de C	22
Archivos de cabeceras de la biblioteca ANSI C.....	22
Interfaz de una Biblioteca	24

Unidad 5

1. TAD (TIPO ABSTRACTO DE DATOS)

1.1. Concepto

Aunque los términos tipo de datos, estructuras de datos y tipos de datos abstractos parecen semejantes, su significado es bastante diferente.

Por ejemplo, sea la siguiente narrativa: “Se necesita simular la cola que realizan los alumnos de la facultad para ser atendidos en la oficina de alumnos”. Dicha atención implica poner un alumno en la cola de espera y luego de atenderlo se debe sacarlo de la misma para que pase el siguiente alumno.

Se tendría que decidir la estructura de datos a utilizar para programar la solución. Si utilizamos **tipos de datos** simples únicamente (char, integer, etc) resultará sumamente difícil implementar una solución. Se requiere una **estructura de datos** mediante los tipos definidos por el usuario, los cuales posibilitan la definición de valores de datos más cercanos al problema que se pretende resolver. Se puede usar registros (struct) para que contengan los datos de los alumnos y un array para gestionar la cola de atención, teniendo de esta forma un array de registros. Con estas decisiones se puede proceder a programar modularmente la solución. Como la estructura de datos elegida es compleja, seguramente las funciones y procedimientos necesarios para trabajar con dicha estructura, tendrán un nivel elevado de complejidad.

Sería necesario contar con un tipo de datos que permita definir la estructura de datos que se necesita (por ejemplo un array de registro) y a la vez las operaciones básicas que se realizarán en dicha estructura (mediante el uso de funciones y procedimientos. Por ejemplo: poner un alumno en la cola sería implementar un procedimiento que permita colocar un registro con los datos de un alumno, en el array). El tipo de datos que permite trabajar de esta forma se define como **tipo de datos abstracto**.

Definiciones:

Tipo de dato: es un atributo de los datos que impone ciertas restricciones sobre los mismos, como qué valores pueden tomar y qué operaciones se pueden realizar.

Estructura de datos: es una forma de organizar un conjunto de datos elementales con el objetivo de facilitar su manipulación. Un dato elemental es la mínima información que se tiene en un sistema.

Un tipo abstracto de dato (en adelante TAD) se puede pensar como un modelo matemático con una serie de operaciones definidas para ese modelo. Es muy importante apreciar que las operaciones que manipulan los datos están incluidas en las especificaciones del TAD. Por ejemplo, un TAD Conjunto puede ser definido como una colección de elementos que son tratados por operaciones como la UNION, INTERSECCIÓN y DIFERENCIA de conjuntos.

Las operaciones definidas sobre el TAD deben ser cerradas, es decir, sólo se debe acceder a los datos mediante las operaciones definidas sobre ellos.

Para representar un modelo matemático básico de un TAD se emplean las estructuras de datos, quizá de tipos distintos, conectadas entre sí de diversas formas. Los objetos tales como listas, conjuntos y grafos, así como sus operaciones, se pueden considerar como tipos abstractos de datos. No existe ninguna regla sobre qué operaciones debe manejar un TAD, ésta es una decisión propia del diseño.

La comunicación entre el programa de aplicación que utiliza el TAD y la implementación del tipo abstracto de datos debe producirse únicamente a través de la interfaz que aportan las operaciones definidas en el TAD. Esto significa que, mientras la interfaz no sea cambiada, la implementación podría ser completamente alterada sin afectar al programa de aplicación que usa el TAD.

Las estructuras de datos utilizadas pueden ser implementadas de varias formas, pero si se hace con corrección los programas que las usen no necesitarán saber qué implementación se usó.

Un TAD encapsula cierto tipo de dato en el sentido de que es posible localizar la definición del tipo y todas las operaciones de ese tipo en una sección del programa. Esto permite que si se desea modificar el comportamiento del TAD se sabe a dónde dirigirse y revisando una pequeña sección del programa se puede tener la seguridad de que no hay detalles en otras partes que puedan ocasionar errores relacionados con este TAD.

Idealmente una "muralla" se levanta alrededor de la implementación de un TAD de tal forma que un programa de aplicación sólo es capaz de manipular los datos por medio del uso de rutinas de acceso. En éste caso, la manipulación directa de los datos almacenados en la implementación del TAD no está permitida.

1.1. Especificación de un TAD

La especificación de un TAD consta de dos partes: la descripción matemática de una colección de objetos, que será representado por una estructura de datos y un conjunto de operaciones definidas para la estructura definida.

Vamos a decir que dos TADs son diferentes si tienen el mismo modelo matemático, pero distintas operaciones, ya que lo apropiado de una realización depende en gran medida de las operaciones que se van a realizar.



Figura 1. Especificación de un TAD

EJEMPLO DE ESPECIFICACIÓN DEL TAD CONJUNTO DE LETRAS MAYÚSCULAS

Este TAD lo vamos a usar para representar los conjuntos de letras mayúsculas, así como las operaciones definidas sobre conjuntos. Definimos el Modelo Matemático y las Operaciones.

Modelo matemático: Un *conjunto* se define como una colección de *elementos*. Todos los elementos de un conjunto son distintos, lo que significa que ningún conjunto puede tener dos copias del mismo elemento. Este modelo matemático debe ser representado por uno o más tipos de datos.

Operaciones: Existen muchas operaciones que se pueden definir sobre conjuntos. De acuerdo a la naturaleza del problema a resolver se deben elegir alguna de ellas:

- INICIA (C) Genera un conjunto C vacío.
- INSERTA (x, C). Agrega el elemento x al conjunto C.
- ELIMINA (x, C). Elimina el elemento x del conjunto C.

- PERTENECE (x, C). Verifica si el elemento x pertenece al conjunto C o no. Devuelve verdadero o falso.
- UNION_CONJ (C1, C2, C). Retorna un nuevo conjunto C que contiene los elementos de C1 y de C2.
- INTER_CONJ (C1, C2, C). Retorna un nuevo conjunto C que contiene solo los elementos que pertenecen tanto a C1 como a C2.
- DIFER_CONJ (C1, C2, C). Retorna un nuevo conjunto C que contiene solo los elementos que pertenecen a C1 y no a C2.
- IGUAL (C1, C2). Retorna verdadero o falso, según los conjuntos ingresados tengan los mismos elementos o no.
- MOSTRAR (C): Muestra los elementos del conjunto C

Tanto la estructura definida en el modelo matemático como los prototipos para cada una de las operaciones deben estar definidos en un archivo de cabecera (.h) de la biblioteca, como se verá más adelante. A continuación vemos el código de este archivo de cabecera.

1.1. Interfaz Pública del TAD:

/*Declaración de la estructura de datos*/

typedef short int tip_conj [27]; /*Se puede optar por usar otra estructura de datos*/

/*Declaración de las operaciones*/ /*Se pueden usar otras operaciones*/

void inicia(tip_conj);

void inserta(char, tip_conj);

short int pertenece(char, tip_conj);

void elimina(char, tip_conj);

void union_conj(tip_conj, tip_conj, tip_conj);

void inter_conj(tip_conj, tip_conj, tip_conj);

void difer_conj(tip_conj, tip_conj, tip_conj);

short int igual(tip_conj, tip_conj);

void mostrar(tip_conj);

1.2. Ventajas del uso de TADs

Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se puede resumir en los siguientes:

- a) Permite una mejor conceptualización y modelado del mundo real. Mejora la representación y facilita la comprensión. Clarifica los objetos basados en estructuras y comportamientos comunes.
- b) Mejora la robustez del sistema. Los TAD permiten la comprobación de tipos de dato para evitar errores en tiempo de ejecución.
- c) Mejora el rendimiento ya que permite la optimización de tiempo de compilación.
- d) Separa la implementación de la especificación. Permite la modificación y mejora de la implementación, sin afectar la interfaz pública del TAD, es decir las rutinas que lo usan.
- e) Permite la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.
- f) Recoge mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

1.3. Abstracción.

La abstracción es la clave para diseñar un buen software, un programa no es más que una disposición abstracta de un procedimiento o fenómeno que existe o sucede en el mundo real.

Los programadores hemos tenido que luchar con el problema de la complejidad durante mucho tiempo. Para ello hemos desarrollado una técnica excepcionalmente potente para tratar la complejidad: abstraernos de ella.

Como somos Incapaces de dominar en su totalidad a los objetos complejos, ignoramos los detalles “no esenciales”, tratando en su lugar con el modelo ideal del objeto y centrándonos sólo en el estudio de sus aspectos esenciales para resolver un problema determinado. Una buena abstracción elimina todos los detalles poco importantes y nos permite enfocarnos y concentrarnos en los detalles importantes.

En conclusión, la abstracción es la capacidad para encapsular y aislar la información del diseño y ejecución.

Con mucha frecuencia se utilizan los términos *TDA* y *Abstracción de Datos* de manera equivalente, y esto es debido a la similitud e interdependencia de ambos. Sin embargo, es importante definir por separado los dos conceptos.

La abstracción de datos consiste en ocultar las características de un objeto y obviarlas, de manera que solamente se utilice el nombre del objeto en nuestro programa. Esto es similar a una situación de la vida cotidiana. Cuando alguien menciona la palabra “perro”, los demás no necesitan que les digan lo que hace el perro. Ya saben la forma que tiene un perro y también saben que los perros ladran. De manera que se abstraen las características de todos los perros en un solo término, al cual se llamó “perro”. A esto se le llama ‘Abstracción’ y es un concepto muy útil en la programación, ya que un usuario no necesita mencionar todas las características y funciones de un objeto cada vez que éste se utiliza, sino que son declaradas por separado en el programa y simplemente se utiliza el término abstracto (“perro”) para mencionarlo.

En el ejemplo anterior, “perro” es un Tipo de Dato Abstracto y todo el proceso de definirlo, implementarlo y mencionarlo es a lo que se llama Abstracción de Datos.

A continuación se puede ver un ejemplo de cómo fue el proceso de abstracción en la historia del software:

En los primeros días de la informática, los programadores enviaban instrucciones binarias a una computadora, manipulaban directamente interrupciones en sus paneles frontales. A partir de ese momento se fue evolucionando de acuerdo a los siguientes niveles de abstracción:

- 1er nivel de abstracción: Los pnemotécnicos del lenguaje ensamblador eran abstracciones diseñadas para evitar que los programadores tuvieran que recordar las secuencias de bits que componen las instrucciones de un programa.
- 2do nivel de abstracción: El siguiente nivel de abstracción se consigue agrupando instrucciones primitivas para formar macroinstrucciones.
- 3er nivel de abstracción: Aparecen los lenguajes de programación de alto nivel. Estos permitieron a los programadores distanciarse de las arquitecturas específicas de una máquina dada. Cada instrucción en un lenguaje de alto nivel puede invocar varias instrucciones de máquina, dependiendo de la máquina específica donde se compila el programa. Esta abstracción permitía a los programadores escribir software para propósito genérico, sin preocuparse sobre qué máquina ejecutaría el programa.
- 4to nivel de abstracción: Secuencias de sentencias de lenguaje de alto nivel se pueden agrupar en funciones o procedimientos y ser invocados por una sentencia. La programación estructurada alienta el uso de abstracciones de control, tales como bucles o sentencias if-then, que se han incorporado en lenguajes de alto nivel.

Como se ve, el proceso de abstracción fue evolucionando desde la aparición de los primeros lenguajes de programación. El método más idóneo para controlar la complejidad fue aumentar los niveles de abstracción.

Los tipos de abstracción que se puede encontrar en un programa son:

a) Abstracción funcional: crear módulos e invocarlos mediante un nombre donde se destaca qué hace el módulo y se ignora cómo lo hace. El usuario sólo necesita conocer la especificación de la abstracción (el qué) y puede ignorar el resto de los detalles (el cómo).

b) Abstracción de datos:

- Tipo de datos: proporcionado por los lenguajes de alto nivel. La representación usada es invisible al programador, al cual sólo se le permite ver las operaciones predefinidas para cada tipo.
- Tipos definidos: por el programador que posibilitan la definición de valores de datos más cercanos al problema que se pretende resolver.
- TAD: para la definición y representación de tipos de datos (Estructura de datos + operaciones), junto con sus propiedades.
- Objetos: Son TADs a los que se añade propiedades de reutilización y de herencia de código. Este tipo de abstracción no es abordado en esta materia.

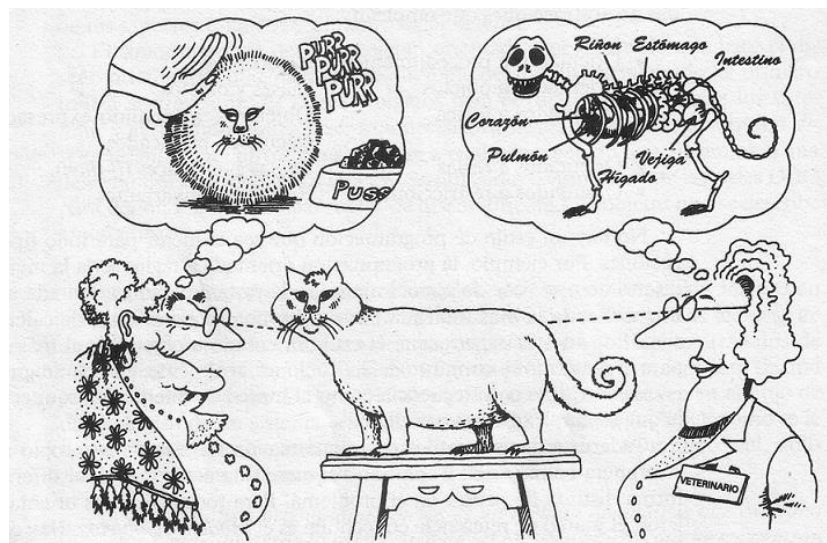


Figura 2. Ejemplo de Abstracción.

1.4. Encapsulamiento.

El encapsulamiento como la ocultación de la información son atributos internos del diseño. La estructura interna del TAD debe estar oculta al usuario del TAD, no necesita conocerla para interactuar con él. Los TAD se conciben como una cápsula cuyo interior está oculto y no puede ser alterado directamente desde el exterior.

Este concepto consiste en el ocultamiento del estado y de los datos de un TAD, de forma que sólo es posible modificar los mismos mediante las operaciones definidas para dicho TAD.

Los datos están aislados del exterior, de forma que una aplicación puede contener un conjunto de TADs que colaboran entre sí mediante el paso de mensajes invocando sus operaciones. De esta forma, los detalles de implementación permanecen "ocultos" a las personas que usan los TADs, evitando así modificaciones o accesos indebidos a los datos que almacenan dichos TADs.

Además, el usuario del TAD no se tiene que preocupar de cómo están implementados las operaciones y atributos, concentrándose sólo en cómo debe usarlos.

Esto trae ventajas asociadas:

- a) la facilidad de modificación; un tipo de datos encapsulado adecuadamente puede ser modificado sin afectar a los programas de aplicación que usan este tipo de datos.
- b) La posibilidad de reutilización en otros programas de aplicación que requieren la funcionalidad aportada por el tipo de datos, sin tener que conocer cómo fue implementado.
- c) La dificultad inherente a la modificación de la implementación de un TAD es independiente del tamaño total del sistema. Esto permite que los sistemas evolucionen con mayor facilidad.
- d) La simplificación en el uso del TAD al ocultar los detalles de su funcionamiento y presentarlo en términos de sus operaciones. Al elevar el nivel de abstracción se disminuye el nivel de complejidad de un sistema. Es posible modelar sistemas de mayor tamaño con menor esfuerzo.
- e) Constituye un mecanismo de integridad. La dispersión de un fallo a través de todo el sistema es menor, puesto que al presentar una división entre interfaz e implementación,

los fallos internos de un objeto encuentran una barrera en el encapsulamiento antes de propagarse al resto del sistema.

- f) Permite la sustitución de TAD con la misma interfaz y diferente implementación. Esto permite modelar sistemas de mayor tamaño con menor esfuerzo.



Figura 3. Encapsulamiento.

1.5. Interfaz e Implementación.

Las estructuras de los TAD se componen de dos partes: la interfaz y la implementación. Esto se debe a que las estructuras de datos reales que utilizamos para almacenar la representación de un tipo abstracto de datos son invisibles para los usuarios o clientes. Mientras que en la interfaz se declaran las operaciones y los datos, la implementación contiene el código fuente de las operaciones y lo mantiene oculto al usuario.

EL concepto formal de interfaz se refiere al límite entre dos entidades distintas. La interfaz sirve como cubierta a la correspondiente implementación. Los usuarios de un TAD tienen que preocuparse por la interfaz, pero no por la implementación, ya que esta puede cambiar en el tiempo y afectar a los programas que usan el TAD. Esto se basa en el concepto visto anteriormente de ocultación de información, una protección para el programa de decisiones de diseño que son objeto de cambio.

La solidez de un TAD reposa en la idea de que la implementación está escondida al usuario. Sólo la interfaz es pública. Esto significa que el TAD puede ser implementado de diferentes formas, pero mientras se mantenga consistente con la interfaz, los programas que lo usan no se ven afectados.

A la estructura interna de un TAD se la denomina implementación y a la parte visible, la que se presenta al exterior, interfaz. La interfaz se define por sus operaciones.

2. Bibliotecas o Librerías

El lenguaje C provee de Bibliotecas o Librerías para cumplir con las premisas del encapsulamiento y el ocultamiento de la información. En adelante cuando se haga referencia a biblioteca o librería entiéndase que estamos hablando de lo mismo.

Una biblioteca es un conjunto de implementaciones que tienen una interfaz bien definida para el comportamiento que se invoca. A diferencia de un programa ejecutable, el comportamiento que implementa una biblioteca no espera ser utilizada de forma autónoma (un programa sí ya que tiene un punto de entrada principal), sino que su fin es ser utilizada por otros programas, independientes y de forma simultánea. Por otra parte, el comportamiento de una biblioteca no tiene porqué diferenciarse en demasía del que pudiera especificarse en un programa. Es más, unas bibliotecas pueden requerir de otras para funcionar, pues el comportamiento que definen refina, o altera, el comportamiento de la biblioteca original; o bien la hace disponible para otra tecnología o lenguaje de programación. Las bibliotecas pueden vincularse a un programa (o a otra biblioteca) en distintos puntos del desarrollo o la ejecución, según el tipo de vínculo que se quiera establecer.

La idea central de una biblioteca es precisamente la de ser un módulo de software preconstruido para cuya utilización no es necesario conocer los detalles íntimos de su funcionamiento, sino su interfaz. Es decir, que respuestas nos puede dar y cómo hay que preguntar -a la librería- para obtenerlas.

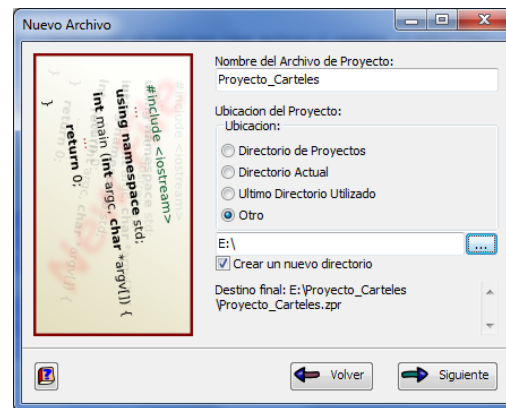
Existen dos tipos fundamentales de bibliotecas: estáticas y dinámicas, que aunque comparten el mismo nombre genérico "biblioteca", utilizan mecanismos distintos para proporcionar su funcionalidad al ejecutable.

En ambos casos es costumbre, que junto a las bibliotecas propiamente dichas los archivos .lib, .a, .dll, etc, se incluya un fichero .h denominado "de cabecera", que los incluirán en el fuente durante la fase de preproceso. Este archivo contiene las declaraciones de las entidades contenidas en la biblioteca, así como las macros y constantes predefinidas utilizadas en ella, de forma que el programador sólo tiene que incluir el correspondiente fichero .h en su aplicación para poder utilizar los recursos de la biblioteca (recuerde que en C es imprescindible incluir la declaración de cualquier función antes de su utilización). Este sistema tiene la ventaja adicional de que proporciona al

usuario la información mínima para su uso. Es decir, la "interfaz" de las funciones que utilizará. (Para profundizar biblioteca o librerías vaya al Anexo I, al final del documento)

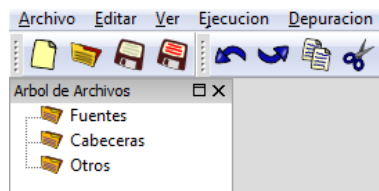
2.1. Cómo crear una Biblioteca o Librería.

Se ejecuta Zinjay, se elige Archivo→nuevo→proyecto



Se escribe el nombre del proyecto (Proyecto_carteles en este ejemplo) y se indica el directorio donde crearlo (E:\ en este caso).

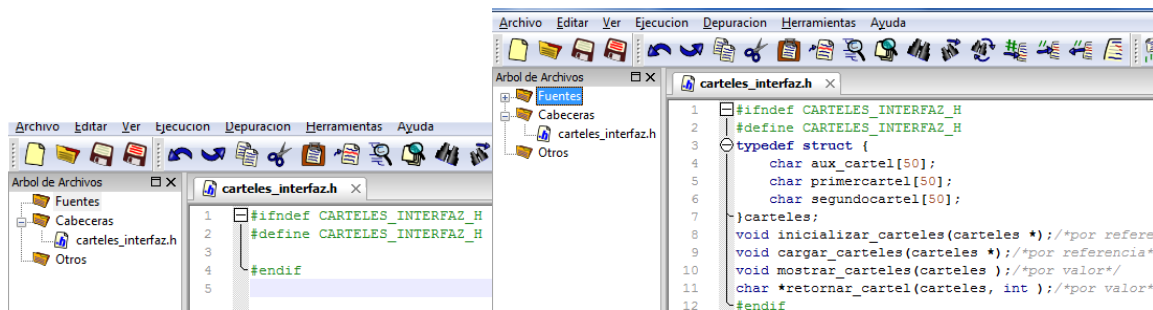
Se selecciona *Siguiente*; luego en utilizar plantilla marcamos la opción *Proyecto en blanco* y se selecciona *Crear*. Zinjay muestra lo siguiente:



Se genera el archivo Proyecto_Carteles.zpr en el directorio E:\Proyecto_Carteles

A continuación se crea la interfaz del TAD, en este caso será una estructura que contiene dos string para almacenar carteles y las operaciones de cargar cartel, inicializar y mostrar carteles.

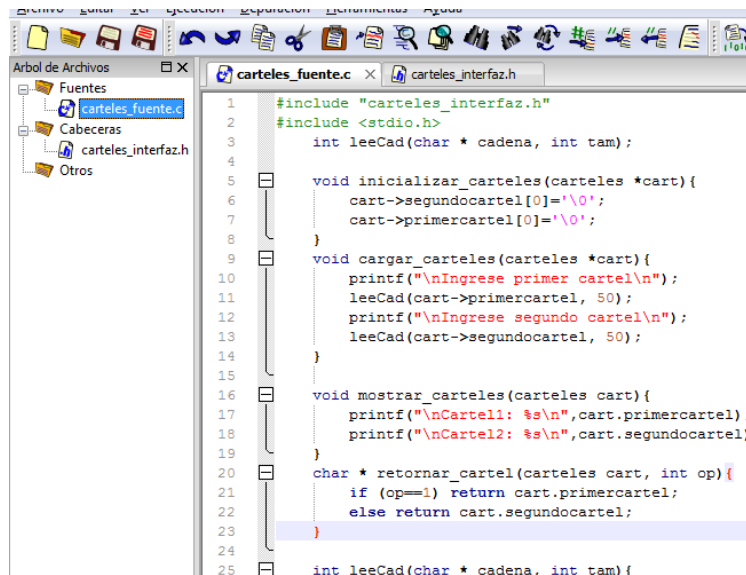
Elegir Archivo→Nuevo; en Tipo de archivo seleccionar *Archivo Cabecera* y se le da un nombre: carteles_interfaz (en este caso). Luego seleccionar *Crear*.



Aquí se define la interfaz, o sea la estructura de datos y el prototipo de las funciones. Se genera el archivo `carteles_interfaz.h` en el directorio `E:\Proyecto_carteles`. Si se quiere generar otros archivos cabeceras. (Por ejemplo si se trabaja con array de registro, conviene crear una interfaz para trabajar con la lista y otra para trabajar con los registros individualmente).

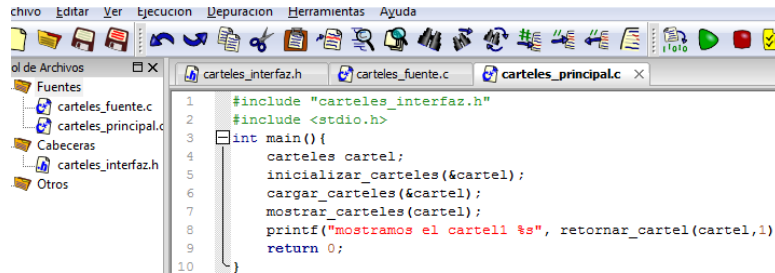
A continuación se escribe el código de la interfaz: Elegir Archivo→Nuevo; en Tipo de archivo seleccionar *Archivo Fuente* y se le da un nombre: `carteles_fuente.c` (en el ejemplo). Luego seleccionar Crear.

Aparece una ventana en blanco donde se comienza a escribir el código de los prototipos definidos en la interfaz. Si es necesario se generan nuevas funciones, las cuales no formarán parte de la interfaz. Recuerde que la primera instrucción debe ser `#include "carteles_interfaz.h"`, para que tome lo declarado en la interfaz.



Luego se crea otro archivo fuente que contenga la función `main`. Esto es necesario para poder compilar el proyecto y corregir todos los errores encontrados. En este caso se

puede escribir el código principal para resolver el problema del ejemplo. Al archivo se lo llamará `carteles_principal.c`, esto genera un archivo con igual nombre en el directorio `E:\Proyecto_Carteles`.



```
1  #include "carteles_interfaz.h"
2  #include <stdio.h>
3
4  int main() {
5      carteles cartel;
6      inicializar_carteles(&cartel);
7      cargar_carteles(&cartel);
8      mostrar_carteles(cartel);
9      printf("mostramos el cartel %s", retornar_cartel(cartel,1))
10     return 0;
11 }
```

Al compilar se observa que muestra los posibles errores en los tres archivos del proyecto. Además, luego de compilar, se genera una carpeta debug en el directorio `E:\Proyecto_Carteles`. Dentro de la carpeta debug se crean los archivos: `carteles_fuente.o`; `carteles_principal.o` y `Proyecto_Carteles.exe` (una aplicación).

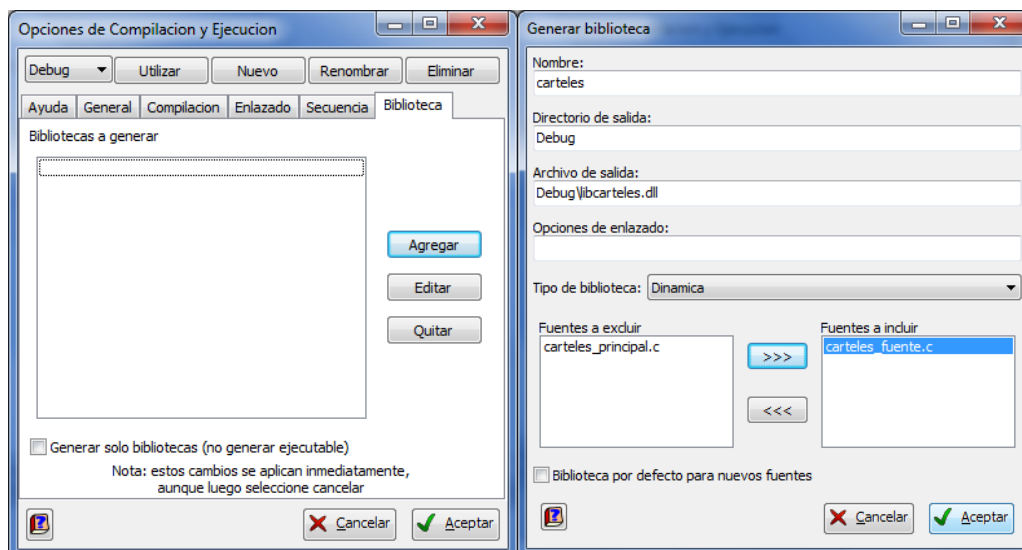
El archivo está listo para probarlo.

En el archivo `carteles_principal.c` se puede hacer uso de todo lo declarado en la interfaz (`carteles_interfaz.h`), como se ve en el ejemplo.

2.2. USO DE UN TAD EN OTRO PROYECTO

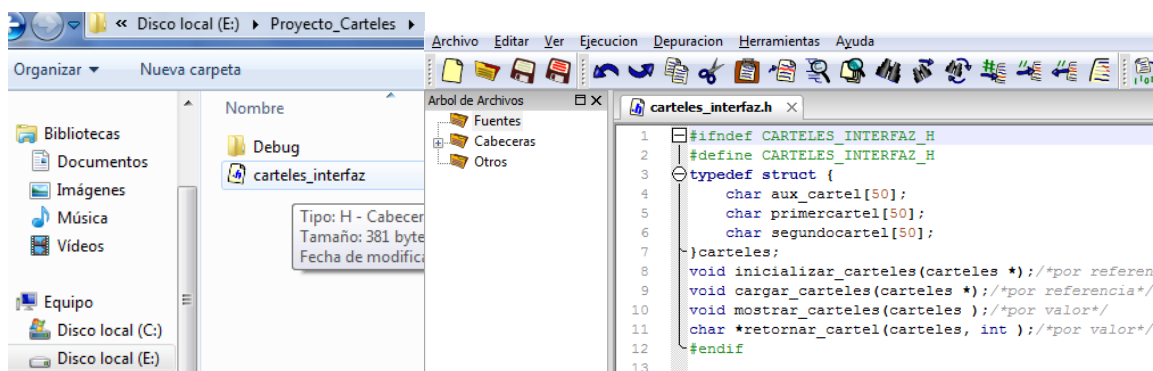
A continuación se detalla como hacer uso del TAD antes creado para utilizarlo en un nuevo proyecto.

Se abre el TAD ya creado y se elige Ejecucion→Opciones→Biblioteca; luego se selecciona el botón *Agregar*.

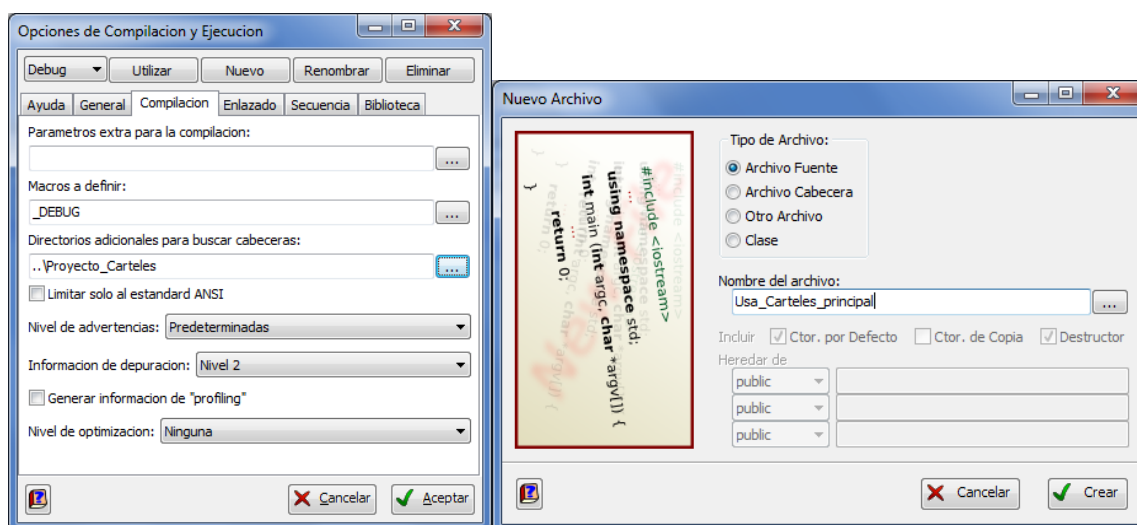


Al colocar el nombre se genera el Archivo de salida con el mismo nombre pero se le antepone 'lib'. (En el ejemplo se puso nombre=carteles y se creó automáticamente *libcarteles.dll* en archivo de salida). Con las flechas se pasan los archivos de Fuentes a incluir hacia Fuentes a excluir y viceversa. Observe que se excluyó *carteles_principal.c* ya que el mismo no forma parte del TAD. Luego de compilar el proyecto, se genera el archivo *libcarteles.dll* en la carpeta Debug). Ya se puede cerrar el proyecto Proyecto_Carteles.

Suponga que se necesita resolver lo siguiente: “Dado N alumnos, se desea darles un mensaje de bienvenida, diferenciando si el alumno es menor o mayor”. Solución: Se abre un nuevo proyecto, llamado Usa_Carteles y se lo guarda en E:\Usa_Carteles. Hacer click con el botón derecho sobre Cabeceras→Agregar Archivo... del proyecto anterior (E:\Proyecto_Carteles) seleccionar *carteles_interfaz.h*→Abrir.



Como `carteles_interfaz` se encuentra en otro proyecto, entonces se debe copiar el archivo en `E:\Usa_Carteles` (Al modificar la interfaz se deberá volver a copiar la interfaz en el proyecto). O se puede indicar donde buscar el directorio (si se mueve la ubicación de la interfaz, se debe indicar la nueva ubicación); para realizar este propósito, se procede de la siguiente forma: Elegir en el menú `Ejecucion`→`Opciones`→`Compilacion` y en *Directorios adicionales para buscar cabeceras* se hace click en los puntos suspensivos y se selecciona *reemplazar selección por directorio* luego se busca `E:\Proyectos_Carteles` luego Aceptar.



A continuación se crea un nuevo archivo fuente: `Archivo`→`nuevo`→`Archivo Fuente`. Llamarlo `Usa_Carteles_principal.c` con el siguiente código:

```
#include "carteles_interfaz.h"

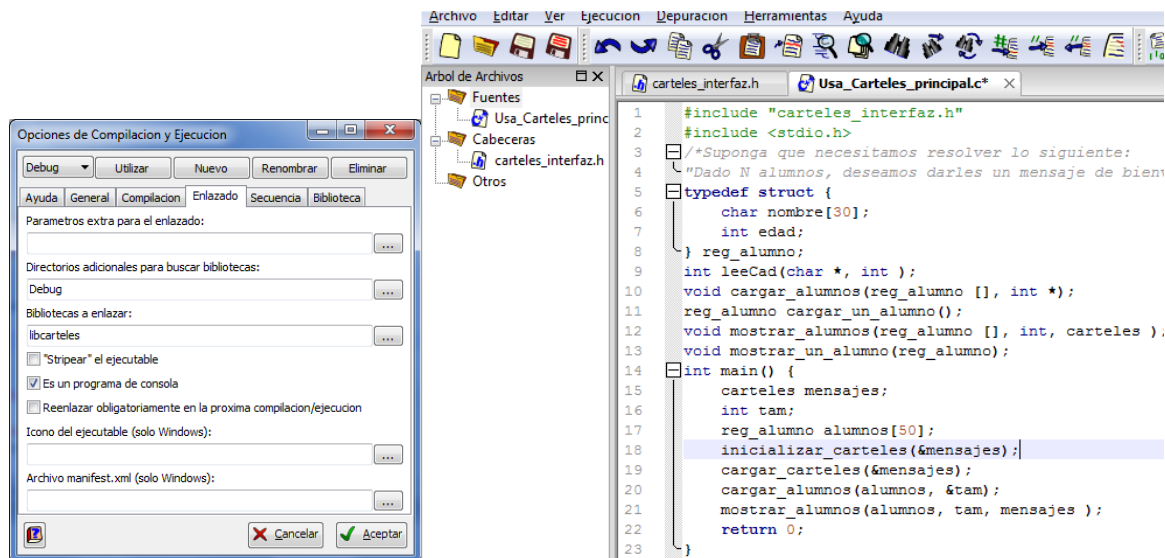
int main() {
    return 0; }
```


Al Compilar, no arroja errores. Se observa que se creó el directorio E:\Usa_Carteles\Debug; el cual contiene los archivos Usa_Carteles_principal.o y Usa_Carteles.exe

Si se quiere usar alguna función de la interfaz, en Usa_Carteles, se presentará un error de compilación ya que falta el código de la interfaz. La solución se encuentra en el archivo libcarteles.dll creado anteriormente. Solamente se tiene que copiar dicho archivo, de la carpeta E:\Proyecto_Carteles\Debug en la carpeta E:\Usa_Carteles\Debug.

Luego seleccionar Ejecucion→Opciones→Enlazado

En *Directorio adicionales para buscar bibliotecas* seleccionar de los puntos suspensivos (....) elegir *Reemplazar todo por directorio* y elegir E:\Usa_Carteles\Debug. En *Bibliotecas a enlazar* se escribe el nombre del dll, en este caso libcarteles. Luego se acepta y ya se puede compilar la nueva aplicación



```
void cargar_alumnos(reg_alumno alumno[], int *tam){
    int i;
    printf("\nIngrese la cantidad de alumnos a cargar = ");
    scanf("%i", tam);
    for (i=1; i<=*tam; i++){
        alumno[i]=cargar_un_alumno();
    }
}

reg_alumno cargar_un_alumno(){
    reg_alumno auxiliar;
    printf("\n\n Ingrese el nombre del alumno = ");
    leeCad(auxiliar.nombre, 30);
    printf("\n Ingrese la edad del alumno = ");
    scanf("%i", &auxiliar.edad);
    return auxiliar;
}

void mostrar_alumnos(reg_alumno alumno[], int tam, carteles mensajes){
    int i;
```

```
        for (i=1;i<=tam;i++){
            mostrar_un_alumno(alumno[i]);
            if (alumno[i].edad <= 16) printf ("%s",retornar_cartel(mensajes, 1));
            else printf ("%s",retornar_cartel(mensajes, 2));
        }
    }
void mostrar_un_alumno(reg_alumnoun_alumno){
    printf("\n\n Bienvenido alumno %s sos ",un_alumno.nombre );
}

void leeCad(char * cadena, int tam){
....
```

2.3. Un Ejemplo: TAD LISTA

Una lista es una secuencia de 0 a n elementos del mismo tipo. A cada elemento se lo conoce como nodo.

Especificación formal del TAD Lista

Matemáticamente, una lista es una secuencia de cero o más elementos de un determinado tipo. $(a_1, a_2, a_3, \dots, a_n)$ donde $n \geq 0$, si $n = 0$ la lista es vacía.

Los elementos de la lista tienen la propiedad de que están ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que a_i precede a a_{i+1} para $i = 1, 2, 3, \dots, n-1$ y que a_i sucede a a_{i-1} para $i = 2, 3, \dots, n$.

Para formar el **Tipo Abstracto de Datos LISTA** a partir de la noción matemática de lista, se debe definir un conjunto de operaciones con objetos de tipo lista.

La decisión de qué operaciones serán las más utilizadas depende de las características del problema que se va a resolver. También dependerá del tipo de representación elegida para las listas. Este conjunto de operaciones lo definirán en la práctica.

Para representar las listas y las correspondientes implementaciones pueden seguirse dos alternativas:

- Utilización de la estructura estática de *arreglo* para almacenar los nodos de la lista.
- Utilización de estructuras dinámicas, mediante punteros y variables dinámicas.

Implementación del TAD Lista con estructuras estáticas

En la implementación de Listas mediante arreglos, los elementos se guardan en posiciones contiguas del arreglo.

En esta realización se define el tipo *Lista* como un registro de dos campos: el primero, es el arreglo de elementos (tendrá un máximo número de elementos preestablecido); y el segundo, es un entero, que representa la posición que ocupa el último elemento de la lista (tamaño). Las posiciones que ocupan los nodos en la lista son valores enteros; la posición *i-énésima* es el entero *i*.

Definición de Listas usando arreglos:

Int Max_elem= ...(Dependerá de cada realización)

Tipo_elem.. (Tipo de los elementos de la lista) Por ejemplo int Tipo_elem;

```
Typedef struct {  
    Tipo_elem arreglo [Max_elem];  
    Int Tope;  
} TLista;
```

*/*La declaración de las operaciones dependerá de cómo se diseñe el TAD */*

Con esta realización se facilita el recorrido de la lista y la operación de añadir elementos al final.

Sin embargo, la operación de insertar un elemento en una posición intermedia de la lista obliga a desplazar en una posición a todos los elementos que siguen al nuevo elemento a insertar con el objeto de dejar un "hueco".

3. Bibliografía

Programación en C, Metodología, Algoritmos y Estructuras de Datos - Luis Joyanes Aguilar, Ignacio ZahoneroMartinez.

Curso de Programación en C/C++ - Francisco Javier Ceballos Sierra.

Estructura de Datos y Algoritmos – Aho, Hopcroft, Ullman

Unidad TAD Universida Tecnológica Nacional del CHACO.pdf. - Ing. Carolina Orcola- Ingeniería en Sistemas de Información Facultad Regional Resistencia - Año 2004

http://www.zator.com/Cpp/E1_4_4b.htm

<http://cucarachasracing.blogspot.com.ar/2012/09/compilacion-y-bibliotecas-parte-1-por.html#more>

ANEXO 1

Bibliotecas o Librerías estáticas

Denominadas también librerías-objeto, son colecciones de archivos objetos agrupados en un solo archivo de extensión .lib, o .a, junto con uno o varios archivos de cabecera (generalmente .h).

Se puede incluir toda la funcionalidad en el archivo de cabecera .h en cuyo caso no va a existir el archivo compilado .lib o .a. Esto representa un caso extremo que debe ser evitado, ya que por lo general, se incluye en la cabecera la información mínima indispensable para utilizar la biblioteca (la interfaz), incluyendo la implementación en archivos compilados.

Durante la construcción de la aplicación, el preprocesador incluye en los fuentes los archivos de cabecera. Posteriormente, durante la fase de enlazado, el *linker* incluye en el ejecutable los módulos correspondientes a las funciones de la librería que hayan sido utilizadas en el programa, de forma que el conjunto entra a formar parte del ejecutable. De ahí su nombre: Librerías enlazadas estáticamente.

Junto con los módulos .obj o .o que las componen, las bibliotecas estáticas incluyen una especie de índice o diccionario con información sobre su contenido. Este índice contiene los nombres de los recursos públicos de los distintos módulos (que pueden ser accedidos desde el exterior) y su dirección. Estos nombres deben ser distintos para evitar ambigüedades durante el enlazado, y sirven para incrementar la velocidad de enlazado cuando el "Linker" debe incluir alguno en un ejecutable.

Bibliotecas o Librerías dinámicas

Otra forma de añadir funcionalidad a un ejecutable son las denominadas bibliotecas de enlazado dinámico, generalmente conocidas como DLLs, acrónimo de su nombre en inglés ("DynamicLinked Library"). Estas bibliotecas se utilizan mucho en la programación para el SO Windows. Este Sistema contiene un gran número de tales librerías de terminación .DLL. Cualquiera que sea su terminación, de forma genérica nos referiremos a ellas como DLLs, nombre por el que son más conocidas.

Diferencias entre bibliotecas Estáticas y Dinámicas

Las diferencias más relevantes de las librerías dinámicas respecto a las estáticas son:

- Las librerías estáticas quedan incluidas en el ejecutable, mientras las dinámicas son ficheros externos, con lo que el tamaño de la aplicación (nuestro ejecutable) es mayor en el primer caso que en el segundo. Esto puede ser de capital importancia en aplicaciones muy grandes, ya que el ejecutable debe ser cargado en memoria de una sola vez.
- Las librerías dinámicas son ficheros independientes que pueden ser invocados desde cualquier ejecutable, de modo que su funcionalidad puede ser compartida por varios ejecutables. Esto significa que sólo se necesita una copia de cada fichero de librería (DLL) en el Sistema. Esta característica constituye la razón principal de su utilización, y es también origen de algunos inconvenientes, principalmente en sistemas como Windows en los que existen centenares de ellas.
- Si se realizan modificaciones en los módulos de una librería estática, es necesario recompilar todos los ejecutables que la utilizan, mientras que esto no es necesario en el caso de una librería dinámica, siempre que su interfaz se mantenga.
- Como consecuencia de lo anterior, generalmente es más difícil la depuración y mantenimiento de aplicaciones que utilizan librerías dinámicas que las estáticas, ya que en el primer caso, es necesario controlar qué versiones de los ejecutables (.EXE) son compatibles con qué versiones de las DLLs y de estas entre sí, de forma que el usuario no utilice un versiones incompatibles de los ficheros que componen la aplicación.
- Durante la ejecución de un ejecutable, las librerías estáticas que hubiesen intervenido en su construcción no necesitan estar presentes, en cambio las dinámicas deben estar en el mismo directorio o en el camino de búsqueda "Path".
- Las librerías estáticas solo se utilizan en la fase de construcción del ejecutable. Las dinámicas se utilizan durante la ejecución.
- Los ejecutables que utilizan librerías estáticas solo incorporan los módulos de aquellas que necesitan para resolver sus símbolos externos. Por contra, las librerías dinámicas deben ser cargadas en su totalidad aunque no solo se utilice una parte de su funcionalidad (no son divisibles).
- Las librerías estáticas, que entran a formar parte indivisible del ejecutable, son cargadas con el proceso de carga de este. Las librerías dinámicas no necesariamente tienen que cargarse con la carga inicial (aunque pueden serlo). De hecho, una librería dinámica

puede ser cargada bajo demanda en el momento en que se necesita su funcionalidad, e incluso puede ser descargada cuando no resulta necesaria.

- El mecanismo de enlazado estático depende del compilador. El de enlazado dinámico depende del SO, de forma que manteniendo ciertas precauciones, las DLLs construidas con un lenguaje y un compilador pueden ser utilizadas por cualquier aplicación.

Bibliotecas Estándares de C

La biblioteca estándar de C (también conocida como libc) es una recopilación de archivos de cabecera y bibliotecas con rutinas, estandarizadas por un comité de la Organización Internacional para la Estandarización (ISO), que implementan operaciones comunes, tales como las de entrada y salida o el manejo de cadenas. A diferencia de otros lenguajes como COBOL, Fortran, o PL/1, C no incluye palabras clave para estas tareas, por lo que prácticamente todo programa implementado en C se basa en la biblioteca estándar para funcionar.

El nombre y las características de cada función, el prototipo, así como la definición de algunos tipos de datos y macros, se encuentran en el archivo de cabecera (con extensión ".h"), pero la implementación real de las funciones está separada en un archivo de la biblioteca. La denominación y el ámbito de las cabeceras se han convertido en comunes, pero la organización de las bibliotecas sigue siendo diversa, ya que éstas suelen distribuirse con cada compilador. Dado que los compiladores de C, a menudo, ofrecen funcionalidades adicionales que no están especificados en el ANSI C, la biblioteca de un compilador no siempre es compatible con el estándar ni con las bibliotecas de otros compiladores.

Está demostrado que la mayor parte de la biblioteca estándar de C ha sido bien diseñada, aunque, se ha comprobado que algunas partes también son fuente de errores; funciones para entrada de cadenas como gets() o scanf(), producen desbordamientos de buffer, y muchas guías de programación recomiendan evitar su uso.

Archivos de cabeceras de la biblioteca ANSI C

<code><assert.h></code>	Contiene la macro <u>assert</u> (aserción), utilizada para detectar errores lógicos y otros tipos de fallos en la depuración de un programa.
-------------------------------	--

<complex.h>	Conjunto de funciones para manipular números complejos.
<ctype.h>	Contiene funciones para clasificar caracteres según sus tipos o para convertir entre mayúsculas y minúsculas independientemente del conjunto de caracteres (típicamente <u>ASCII</u> o alguna de sus extensiones).
<errno.h>	Para analizar los códigos de error devueltos por las funciones de biblioteca.
<fenv.h>	Para controlar entornos en coma flotante.
<float.h>	Contiene la definición de constantes que especifican ciertas propiedades de la biblioteca de coma flotante, como la diferencia mínima entre dos números en coma flotante (<u>_EPSILON</u>), el número máximo de dígitos de precisión (<u>_DIG</u>), o el rango de valores que se pueden representar (<u>_MIN</u> , <u>_MAX</u>).
<inttypes.h>	Para operaciones de conversión con precisión entre tipos enteros.
<iso646.h>	Para utilizar los conjuntos de caracteres ISO 646 (nuevo en NA1).
<limits.h>	Contiene la definición de constantes que especifican ciertas propiedades de los tipos enteros, como rango de valores que se pueden representar (<u>_MIN</u> , <u>_MAX</u>).
<locale.h>	Para la función <code>setlocale()</code> y las constantes relacionadas. Se utiliza para seleccionar el entorno <i>local</i> apropiado (configuración regional).
<math.h>	Contiene las funciones matemáticas comunes.
<setjmp.h>	Declara las macros <code>setjmp</code> y <code>longjmp</code> para proporcionar saltos de flujo de control de programa no locales.
<signal.h>	Para controlar algunas situaciones excepcionales como la división por cero.
<stdarg.h>	posibilita el acceso a una cantidad variable de argumentos pasados a una función.
<stdbool.h>	Para el tipo booleano.
<stdint.h>	Para definir varios tipos enteros.
<stddef.h>	Para definir varios tipos de macros de utilidad.
<stdio.h>	Proporciona el núcleo de las capacidades de entrada/salida del lenguaje C. Incluye <code>printf</code> .

<code><stdlib.h></code>	Para realizar ciertas operaciones como conversión de tipos, generación de números pseudo-aleatorios, gestión de memoria dinámica, control de procesos, funciones de entorno, de ordenación y búsqueda.
<code><string.h></code>	Para manipulación de cadenas de caracteres.
<code><tgmath.h></code>	Contiene funcionalidades matemáticas de tipo genérico (<i>type-generic</i>).
<code><time.h></code>	Para tratamiento y conversión entre formatos de fecha y hora.
<code><wchar.h></code>	Para manipular flujos de datos anchos y varias clases de cadenas de caracteres anchos (2 o más bytes por carácter), necesario para soportar caracteres de diferentes idiomas (nuevo en NA1).
<code><wctype.h></code>	Para clasificar caracteres anchos (nuevo en NA1).

Interfaz de una Biblioteca

Al hablar de bibliotecas, debemos reconocer la existencia de dos partes bien divididas: Los programas clientes, aquellos que usan las librerías y la implementación de las librerías, lo que determina como funciona internamente cada herramienta. En términos de bibliotecas, la interfaz se refiere al límite entre la implementación y los programas clientes. Al “llamar” a una función de la biblioteca (como cliente), se pasa información (argumentos) a la implementación a través de la interfaz. Una interfaz es entonces el medio a través del cual, estas dos partes se comunican.

Una interfaz bien diseñada, debe cumplir los siguientes requisitos:

- a) Unificación: escogidas de acuerdo a cierto tema, el enfoque se debe mantener.
- b) Simplicidad: en cuanto al número de parámetros, nombre adecuado.
- c) Generalidad: resolver un buen grupo de posibilidades.
- d) Estabilidad: se pueden realizar cambios en la librería pero no necesariamente en el cliente.

La interfaz es la cara que una librería da a un cliente. En una interfaz se pueden incluir: Prototipos de funciones, declaraciones de constantes (usando `#define`), declaraciones de nuevos tipos de datos, esto quiere decir que jamás se incluirá una implementación en una interfaz.

El formato general de una interfaz es:

```
#ifndef _nombre_h
```



```
#define _ nombre_h  
/*Líneas de #include para librerías que se vaya a usar*/  
/*prototipos de funciones o procedimientos*/  
/*definición de constantes*/  
/*declaraciones de nuevos tipos de datos*/  
#endif nombre de la librería
```

En programas complejos, una interfaz puede ser incluida en varios archivos, para evitar “confusiones” se pregunta si la interfaz no ha sido usada aun (ifndef: ifnotdefined) Si no ha sido usada, se define, caso contrario, se ignoran todas las declaraciones de la misma.

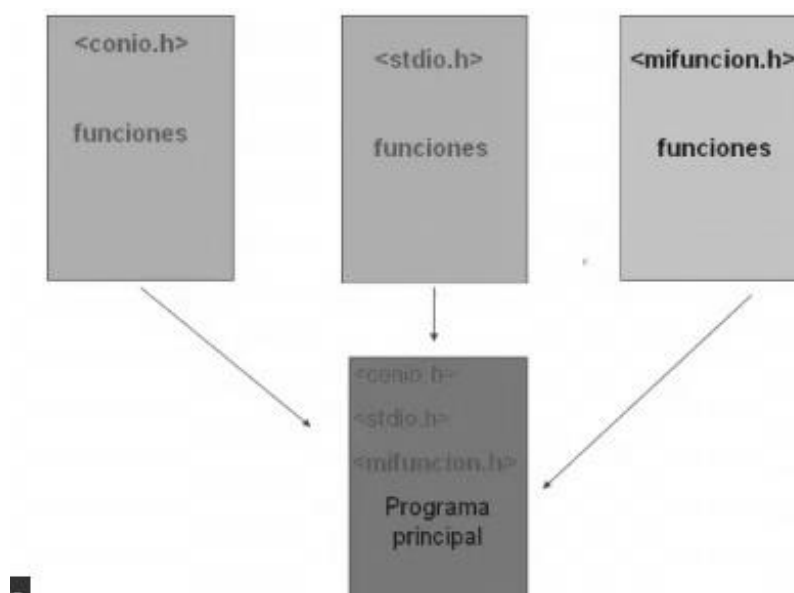


Figura 4: Bibliotecas