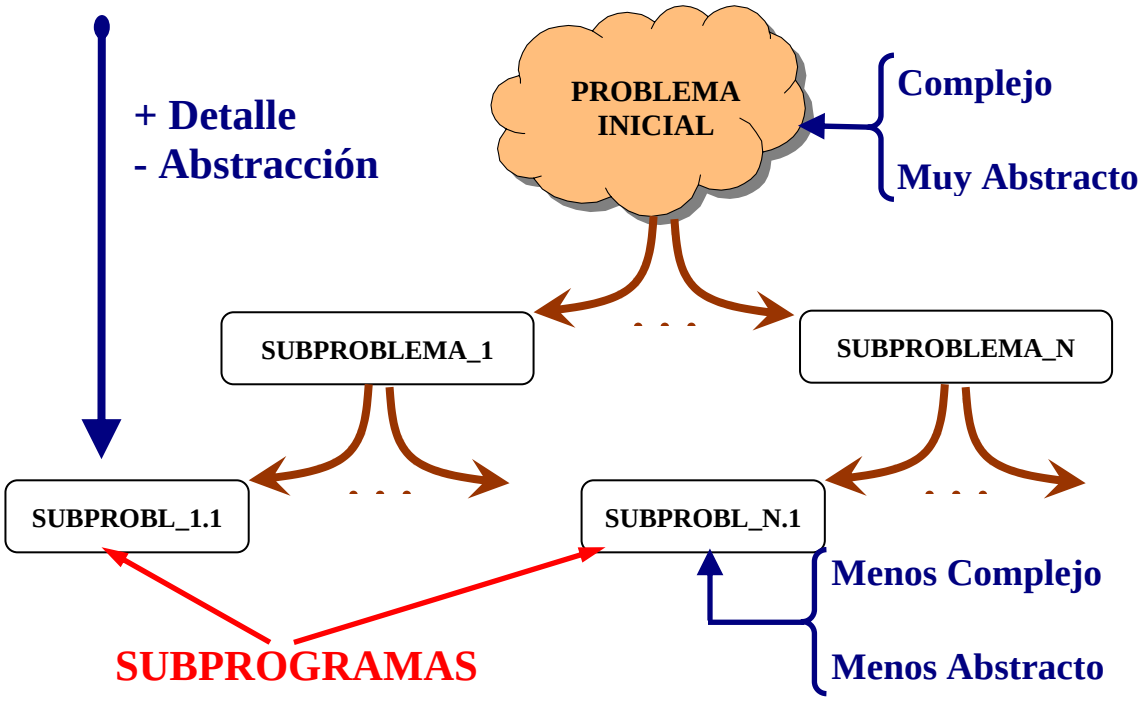
TEMA 7. Diseño Descendente: Subprogramas

7.1 Niveles de abstracción

La solución de cualquier problema puede darse en varias formas o niveles de abstracción.

Se comienza dando un enunciado más general o abstracto de la solución. Se refina esta solución elaborando los detalles que antes se han ignorado, de lo que resulta una solución nueva que es menos abstracta. Este proceso continúa, hasta que se logra un nivel de detalle apropiado. Esta es la esencia del diseño top-down.

Método de trabajo para resolver un problema:



Descomponemos el problema, en subproblemas cada vez más sencillos y concretos ➔ **DISMINUIMOS EL NIVEL DE ABSTRACCIÓN**.

Esta técnica top-down se denomina refinamiento por pasos o programación jerárquica.

7.2 Solución de problemas utilizando técnicas de diseño descendente (refinamientos sucesivos)

Idea del diseño descendente o diseño top-down:

Para solucionar un problema complejo vamos a dividirlo en problemas más simples. La solución a dichos problemas serán los **SUBPROGRAMAS** ➔ Estructura básica de un programa en C.

PROGRAMA ➔ Secuencia "corta" de **sentencias**, la mayoría de las cuales son **llamadas a subprogramas**.

Cada subprograma puede ser utilizado tantas veces como queramos dentro del programa principal ➔ **REUTILIZACIÓN DEL CÓDIGO** ➔ Evita escribir repetidamente las mismas líneas de código

Un programa en C, es un conjunto de suprogramas, siendo la función main el subprograma principal.

Es la primera función que se ejecuta ⇒ Es la encargada de llamar al resto de los subprogramas.

El programa es construido de abajo a arriba, creando 1º subprogramas que resuelvan los módulos de detalle → son usados por otros subprogramas más generales, hasta llegar a la creación del programa original.

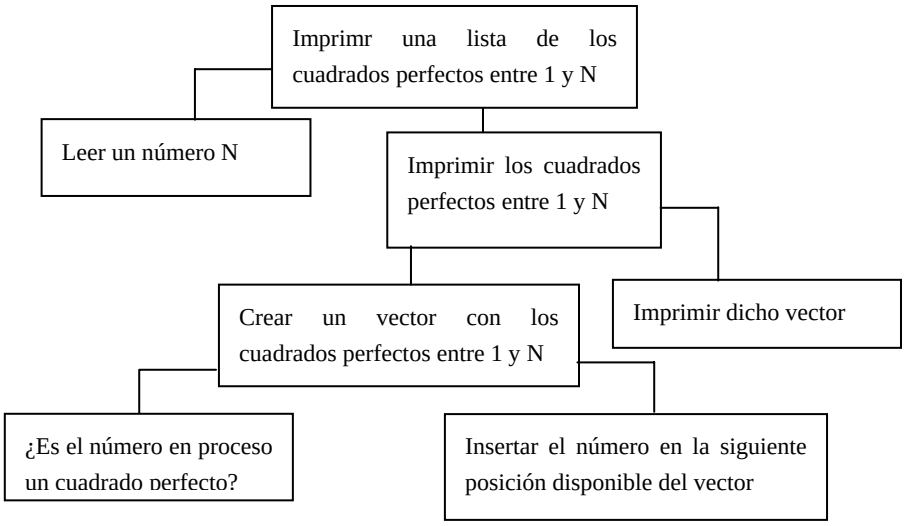


Figura 1. Solución por el método top-down del problema de los cuadrados perfectos.

7.3 Estilo en la creación de un programa

A la hora de programar se deben seguir ciertos criterios, o estilos de programación, que faciliten la elaboración del programa, su posterior modificación, la reutilización del código y su legibilidad por parte de otros programadores que no son los propios creadores.

Para conseguirlo: Programación Estructurada y Modular.

7.3.1 Programación Estructurada

Consiste en la utilización del **DISEÑO DESCENDENTE** para resolver un problema ➔ **Utilización de Subprogramas**.

7.3.2 Programación Modular

El conjunto de subprogramas utilizados para la elaboración de un programa, pueden agruparse en uno o varios ficheros ➔ **Un programa** puede estar formado por **varios archivos fuente** ➔ A estos archivos fuente se les llama MÓDULO.

> Pueden compilarse por separado, y cargarse (en el programa principal) junto con las librerías de C, usándose de forma similar. Pueden modificarse sin afectar a otros módulos.

Ventajas:

Al ser los módulos independientes, **cada módulo** puede ser **programado por una persona diferente** (siguiendo unas especificaciones), **y quien los utiliza no tiene por qué saber cómo trabajan internamente** las funciones que contiene → reducción del tiempo en realizar el programa

Cada **programa** tendrá un **módulo principal** que controla y transfiere el control al resto de submódulos (subprogramas), y éstos a su vez cuando finalizan, devuelven el control al módulo principal.

En C, **main( )** es el punto de entrada al programa, es la 1ª función que se ejecuta y desde ella se llama al resto de funciones del programa.

7.4 Subprogramas: Procedimientos y funciones

Los subprogramas ofrecen tres ventajas:

1. Mejoran la legibilidad del programa.
2. Acortan los programas, ya que evitan tener que escribir repetidamente las correspondientes instrucciones.
3. Permiten que puedan ser reutilizados en otros programas, con lo que se acorta el tiempo de realización de éstos.

Un subprograma puede ser utilizado tantas veces como se desee en distintas partes del programa o en otros programas.

Los subprogramas pueden ser de dos tipos:

- **Procedimientos:** son **subprogramas** que realizan una tarea determinada y **devuelven 0 o más de un valor**. Se utilizan para estructurar un programa y mejorar su claridad y generalidad.

  Debido a que devuelven más de un resultado, **NO UTILIZAN** la palabra reservada **return**, y los parámetros pueden ser:

  - **de ENTRADA** ➔ Sólo se utilizan para que los subprogramas que llaman al procedimiento le pasen datos al mismo.
  - **de ENTRADA/SALIDA** ➔ Se utilizan por parte de los subprogramas que llaman, para pasarle datos al procedimiento, y por parte del procedimiento para pasar los resultados obtenidos al subprograma que lo ha llamado.

- **Funciones:** son **subprogramas** que realizan una determinada tarea y **devuelven un único resultado** o valor. Se utilizan para crear operaciones nuevas no incluidas en el lenguaje.

  El resultado devuelto se indica mediante la palabra reservada **return**, y **TODOS LOS PARÁMETROS** son de **ENTRADA**.

En C no se distingue entre funciones y procedimientos: todos son funciones. Un programa C está formado por varias funciones, una de las cuales se llama **main()**, y hace de programa principal.

Formalmente, una función tiene la siguiente sintaxis general:

```
static tipo_devuelto nombre_funcion (declaracion_parametros )
{
 declaraciones;
 sentencias;
 return valor_devuelto;
}
```

cabecera o prototipo

¡¡ EN EL CÓDIGO DE LA FUNCIÓN SÓLO SE USAN LAS VARIABLES DECLARADAS DENTRO DEL MISMO, Y LOS **PARÁMETROS** !!

Vía de comunicación entre subprogramas.

donde:

- **static** (opcional): indica que la función tiene **alcance de módulo**. Si omitimos **static**, la función tiene **alcance global**.
- tipo_devuelto: indica el tipo del valor devuelto por la función, a través de **return**. Si la función no devuelve nada → **void.**
- nombre_funcion (obligatorio): identificador de la función.
- declaracion_parametros (opcional): lista variables pasadas a la función. Sintaxis:

```
static tipo funcion(tipo var1, tipo var2, . . ., tipo varN)
```

Todas las variables de la lista tienen que incluir el tipo y el nombre, incluso si hay varios parámetros del mismo tipo:

```
funcion(int x, int y, char c)   /* correcto */
funcion(int x, y, char c)       /* incorrecto */
```

Si no pasamos variables (datos) a la función (lista vacía):

```
static tipo funcion (void)     /* la funcion no tiene parametros */
```

- Las llaves de bloque '{' y '}' delimitan el cuerpo de la función.
- **return** (opcional): Permite salir de la función o devolver un valor a la sentencia que la llamó. Puede haber varios **return**.
- valor_devuelto (opcional): valor que devuelve la función. Debe ser del mismo tipo que el especificado en tipo_devuelto.

Por defecto, las funciones tienen alcance global, a menos que se declaren como **static** que indican que tienen alcance de módulo.

Es posible omitir el tipo de los argumentos en la declaración de los parámetros, pero debe indicarse en una sentencia, antes de '{'. Este forma de declaración (llamada formato clásico) está obsoleta.

```
/* Formato actual */
int func ( int a, int b, char c )
{
   . . .
}
```

```
/* Formato clásico */
int func (a, b, c )
int a, b;
char c;
{
   . . .
}
```

Las funciones:

1. Tienen tiempo de vida global.
2. Tienen alcance global, excepto las **static** (alcance de módulo)
3. No pueden anidarse (no puede definirse dentro de otra).
4. Pueden ser recursivas y una función puede llamar a otra
5. Una función no puede ser destino de una asignación.

```
ordenar(x, y)= 20; /* incorrecto */
```

Todas las funciones, excepto las **void**, devuelven un valor (en el **return)** y pueden ser usadas como operando en cualquier expresión

```
/* suponemos que max, raiz, cubo y cuadrado devuelven un valor */
x = max(y, z) - raiz(9) + 7;
if (cubo(x) > cuadrado(y))
   z = cubo(x);
```

Una función **void** no puede usarse en expresiones o asignaciones.

```
void func(int a); /* prototipo de función */
y = func(x); /* no hay ningun valor para asignar a y */
z = func(5) – 6; /* no hay valor que sumar a 6 */
```

Aunque una función devuelva un valor, no es obligatorio asignárselo a una variable en la sentencia que llama a la función:

```
#include <stdio.h>

int max(int a, int b) {  // funcion con 2 parametros que devuelve un entero
    if (a > b)   return a;
    else return b;
}
void pedir(void) {   // funcion que no tiene parámetros y no devuelve nada
   printf("Dame dos enteros: ");
}

int main(void) {
    int x, y, mayor;
    pedir( );          // llamada a una función que no tiene parametros
    scanf("%d%d", &x, &y);
    mayor = max(x, y);
    printf("El valor mayor de %i y 50 es %i", mayor, max(mayor, 50) );
    max(5,7);          // el valor devuelto por max se pierde al no asignarse
}
```

7.5 Parámetros y variables locales. Variables globales

Dependiendo del alcance, visibilidad y tiempo de vida, distinguimos:

Variables locales, parámetros y variables globales.

7.5.1 Ambito o alcance de las variables

El Ámbito de las variables indica:

- **desde dónde** es **VISIBLE** una variable, y
- **dónde** se puede **UTILIZAR** una variable.

Existen dos tipos de variables (según su ámbito)

Vbles. de ÁMBITO GLOBAL, es decir **Variables Globales** ➔ Visibles y utilizables en **CUALQUIER** parte del programa Son declaradas a nivel externo al principio del programa..

Vbles. de ÁMBITO LOCAL, es decir **Variables Locales** ➔ Visibles y utilizables **SÓLO** dentro del bloque o función donde están definidas.

¡¡ NO RECOMENDAMOS EL USO DE VARIABLES GLOBALES !!

7.5.2 Variables locales

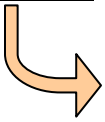
- Son aquellas que están definidas dentro de un bloque o función
- No pueden usarse fuera del bloque o función donde están definidos
- Tiempo de vida local (sólo ocupan memoria mientras existen):
  - Se crean cuando se ejecuta el bloque en el que están definidas
  - Se destruyen al salir del bloque o función donde están
- Una función puede tener variables locales con el mismo nombre que variables globales. La variable local tiene preferencia.

En C++ podemos declarar variables en cualquier parte; sólo pueden ser usadas una vez declaradas y no antes. Una variable no es creada hasta que se ejecuta la sentencia donde se declara:

```
#include <stdio.h>
#include <conio.h>

int a = 1, b = 1, c = 1;  // variables globales

int main(void) {
 printf("A: %i B: %d C: %i \n",a, b, c);   // a, b, c globales
 ver( );
 printf("A: %i B: %d C: %i \n",a, b, c);   // a, b, c globales
 getch( );
 return 0;
}

void ver(void) {
 int a = 5;   // a local
 c++;    // incrementa c global
 printf("a: %i b: %i c: %i \n", a, b, c);   // a local, b y c global
 int c = 50;    // c local, a partir de aquí, las referencias a c son a la local
 c++;    // incrementa c local
 printf("a: %i b: %i c: %i \n", a, b, c);   // a y c local, b global
}
```

7.5.3 Parámetros (por valor y por referencia)

Comunican a los subprogramas con el resto del programa.



Significa que: Reciben los valores con los que trabajarán los subprogramas.

Los parámetros son los valores que se pasan a la función al ser llamada. Dentro de una función, funcionan y se tratan igual que una variable local ➔ se crean al comenzar la ejecución de la función y se destruyen al finalizar ésta (tiempo de vida y ámbito local).

En la llamada a una función, los **ARGUMENTOS** deben coincidir en **NÚMERO** y **TIPO** (*no es necesario que sus nombres coincidan*) con los **PARÁMETROS**, ya que cada parámetro es sustituido por el argumento que en la llamada a la función ocupa su misma posición.

De no ser así se producirán errores o resultados inesperados:

```
#include <stdio.h>
int suma(unsigned char a, unsigned char b) {
 return ( a + b );
}
int main(void) {
 printf("La suma es %i \n",  suma(258, 3)  );
 getch( );
 return 0;
}
```

Al compilar no hay errores, pero el resultado devuelto es:

```
La suma es 5
```

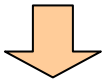
El 258 en binario es: 00000001 00000010

El 3 en binario es: 00000011

Como el compilador esperaba un **char** como primer argumento, del 258 coge solo el primero de los dos bytes que lo forma y lo interpreta como un 2. Por tanto suma 3 y 2, devolviendo un 5.

Los parámetros pueden ser de dos tipos:

- **Parámetros por valor o copia:** Son parámetros **sólo de ENTRADA**. *Se usan para pasar datos a la función* A la función se le pasa una **COPIA** del valor del **ARGUMENTO**;

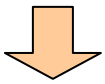
  

  Cualquier modificación sufrida por el **PARÁMETRO**, dentro de la función, **NO afectará al ARGUMENTO**, permaneciendo su valor inalterado cuando el subprograma finaliza.

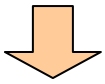
  

  La función trabaja con una copia del argumento.

- **Parámetros por variable o referencia:** Son parámetros de **E/S**. *Se usan para pasar datos a la función y/o para que la función devuelva en ellos los resultados obtenidos* A la función se le pasa **LA DIRECCIÓN DE MEMORIA DEL ARGUMENTO** ➔ Cualquier modificación en el PARÁMETRO, afecta al ARGUMENTO,

  El subprograma trabaja con EL PROPIO ARGUMENTO, ya que accede a su valor a través de su **DIRECCIÓN** (y no a partir de una copia del mismo).

¡ Un parámetro por referencia es un PUNTERO !

C usa el método de llamada por valor para pasar argumentos. Para forzar una llamada por referencia es necesario usar punteros:

```
#include <stdio.h>
int x = 5; /* variable global */

int triple(int x) {
 x = x * 3; return(x);
}

void cambia(int *x) {
 *x=*x + 2;
}

int main(void) {
 printf("El valor de x es %i \n", x);
 printf("El triple de x es %i \n", triple(x) );
 printf("El valor de x es %i \n", x);
 cambia(&x);
 printf("El valor de x es %i \n", x);
 getch( );
 return 0;
}
```

Tras ejecutar el programa, el resultado mostrado es el siguiente:

El valor de x es 5. El triple de x es 15. El valor de x es 5. El valor de x es 7

En la llamada a la función triple sólo se transfiere el valor de la x (5). La variable global x, no se ve afectada y conserva su valor.

En la llamada a la función cambia, se transfiere la dirección de memoria de la variable x, no su valor. Al modificar el valor de dicha dirección de memoria (*x = *x + 2) modificamos la variable global x.

Recomendación: En una función, la declaración de parámetros tendrá:

- **1º** ➔ lista de Parámetros de Entrada.
- **2º** ➔ lista de Parámetros de E/S.

y la **sintaxis** será la siguiente:

```
tipo Nombre (tipo E1, …, tipo En, tipo *ES1, …, tipo *Esn)
```

Parámetros por **VALOR** Parámetros por **REFERENCIA**

Ejemplo: función que devuelve las 2 raíces de una ecuación de 2º grado

```
void ECUACION (int a, int b, int c, float *R1, float *R2)
```

Coeficientes de la ecuación:

$$ax^2 + bx + c = 0$$

Son los parámetros de Entrada.

Raices solución ➔ **Param. de Salida**:

$$R1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad R2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

La llamada sera así:

```
int a, b, c; float r1, r2;
. . .
ECUACION(a, b, c, &r1, &r2);
```

7.5.4 Variables globales

- Son accesibles y visibles en todo el programa
- Se pueden usar en cualquier lugar (y función) del programa.
- Tienen tiempo de vida global. Se declaran a nivel externo.
- Cualquier función puede acceder y modificar su valor. No es necesario pasarla como parámetro a una función.

Las variables globales deben evitarse ya que:

- Ocupan memoria durante toda la ejecución del programa
- Pueden ser modificadas por cualquier función.

7.6 Prototipos de funciones

Informan al compilador del:

- Número y tipos de datos de los argumentos de una función
- Tipo de dato devuelto por una función.

Sirven para:

1. Evitar las conversiones erróneas de los valores cuando los argumentos y parámetros no son del mismo tipo.
2. Detectar errores en el número y tipo de argumentos usados al llamar y/o definir una función.

Los prototipos van al principio del programa. Su sintaxis es:

```
static tipo_devuelto nombre_funcion(tipo var1, . . ., tipo varN);
```

Es como la 1ª línea de la definición de función, añadiendo ';'.

```
int min(int x, int y);   /* prototipo de funcion */
int main(void)
{  float a = 29.57 , b;
   printf("Dame un número");
   scanf(" %f ", &b);
   printf("El número mas pequeño es %i \n", min( a, b ) );
}
int min(int x, int y)
{  if (x < y) return x;
   else  return y;
}
```

Al llamar a **min( )**, pasamos datos **float**, cuando la función espera **int**. Gracias al prototipo de la función, se realiza previamente una conversión de los valores **float** a **int**; el valor de **a** (29.57) se convierte en 30, idem con el de **b**.

De no existir el prototipo, los valores **a** y **b** serian enviados como **float** (4 bytes), y **min( )** los hubiera interpretado como **int** (2 bytes): los valores de **x** e **y** serian absurdos, así como su resultado.

7.7 Módulos

Un **Módulo** (archivo fuente) es una **colección** de **constantes, tipos, funciones**, etc. agrupados en un **mismo fichero**.

El lenguaje C, no sólo **permite** dividir un programa en **distintas funciones**, además permite que dichas funciones puedan a su vez estar **agrupadas en múltiples ficheros o módulos** (en un mismo módulo se suelen agrupar funciones y variables relacionadas entre si).

Un programa puede estar formado por uno o más módulos.

Las funciones existentes en un módulo son visibles y pueden ser usadas en el resto de módulos, a menos que se declaren como **static**.

- Las variables globales de un módulo son visibles y pueden ser usadas en el resto de módulos: Para ello deben ser redeclaradas como **extern** en el resto de módulos donde se quieran usar.
- Las globales **static** sólo pueden ser usadas en el módulo en el que están definidas: no pueden ser usadas en el resto de módulos.

Cuando un módulo usa funciones definidas en otros módulos, debe:

1. Conocer cuál es su prototipo.
2. Conocer para qué sirve cada función.

no le hace falta saber **cómo** están implementados (sólo debe saber usarlas y como llamarlas).

Ventajas de dividir un programa en módulos:

1. **Facilitan la edición**, al trabajar con ficheros más pequeños.
2. **Compilación más rápida**: sólo se compilan los módulos en los que se han producido cambios y no todo el código.
3. **Programación más estructurada**: al tener las funciones y variables relacionadas en un mismo fichero (y no todo mezclado) se entiende todo mejor y disminuimos los errores.
4. **Aumentamos la reutilización del software**: El módulo puede ser utilizado en otros programas, haciendo la programación más rápida y eficiente.

7.8 Ambito de las funciones

Las funciones tienen dos ámbitos: **ámbito global** y **de módulo**.

Por defecto, las funciones tienen alcance global, a menos que se declaren como **static** que indican que tienen alcance de módulo.

- **Ámbito Global** ➔ Son las **funciones exportadas** por un módulo: pueden ser usadas en cualquiera de los módulos que componen el programa simplemente incluyendo sus prototipos

Sintaxis: **#include "modulo.h"** ← **Para importar un módulo.**

- **ÁMBITO DE MÓDULO** ➔ Son **funciones static** (privadas): sólo pueden ser utilizados dentro del módulo en el que han sido creados; no se exportan y no son visibles al resto de módulos.

7.9 Proyectos

En C, los programas de archivos o módulos múltiples se llaman proyectos. Cada proyecto se asocia a un fichero de proyecto, que determina que ficheros son parte del proyecto.

*(En el IDE DEV-C++ los ficheros de proyectos tienen la extensión **.dev**)*

Este fichero (**.dev**) **contendrá** los **nombres** de los ficheros o **módulos** (**.cpp** ó **.c**), que intervienen en la aplicación, de los cuales **sólo uno** podrá contener a la función **main( )**.

Para **cada módulo se define un fichero cabecera (.h)**, el cual contiene los prototipos de las funciones y las variables que se quieren exportar, y **un fichero de implementación (.cpp ó .c)** con la codificación de las funciones tanto públicas como privadas.

Cada módulo debe incluir todos los ficheros cabeceras de los módulos que necesite para su funcionamiento

Un proyecto comprende un fichero proyecto **.dev**, uno o más ficheros cabecera **.h** y dos o más módulos **.c**. Cuando hay un fichero de proyecto **.dev**, el compilador lee el archivo **.dev**, compila cada archivo allí indicado, los enlaza y crea el ejecutable **.exe**.

7.10 Sección de includes: Ficheros cabecera

Los **prototipos** de las funciones de la Biblioteca Estándar de C, están en los **Ficheros Cabecera .h** suministrados por el compilador

Para poder utilizar las funciones de librería de la biblioteca de C es necesario insertar en el programa el archivo cabecera donde estén definidos los prototipos de dichas funciones.

Estos archivos contienen, además de prototipos de funciones:

1. Definiciones, macros y ctes que usan las funciones (**#define**).
2. Referencias externas (variables **extern** de otros módulos).
3. Enumeraciones (**enum**) y declaración de estructuras (**struct**).

Los archivos cabecera se insertan al principio del programa fuente con la directiva **#include** seguida del nombre o ruta completa del fichero **.h**. Un fichero fuente puede tener varios **#include**.

Además de los ficheros cabecera suministrados por el compilador, el usuario puede crear los suyos propios con los prototipos y funciones que el mismo haya implementado.

La directiva **#include** tiene dos tipos de sintaxis.

```
#include <nombre_fichero.h>
```

Los símbolos '<' y '>' indica al compilador que busque el fichero en el directorio include indicado en el entorno integrado de C. En dicho directorio están las librerías ofrecidas por el compilador

```
#include "nombre_con_ruta_acceso.h"
```

Cuando va entre comillas (" ") el compilador busca el fichero en la ruta de acceso especificada, si ésta se indica, o en el directorio de trabajo actual, si no se indica ninguna ruta.

El uso de ficheros cabecera ofrece las siguientes ventajas:

1. Cuando es necesaria una nueva declaración o redefinición, ésta sólo se hace en el fichero **.h**. Los módulos, al contener la directiva **#include** automáticamente quedan actualizados.
2. Cuando un fichero objeto **.o** es utilizado en mas de un programa, sólo es necesario enlazar el modulo objeto **.o** a cada programa e incluir en cada programa el fichero **.h** que contiene las declaraciones y prototipos.

Ejemplo: supongamos un programa compuesto de dos módulos (modulo1 y modulo2) que hacen uso de una serie de constantes definidas en un fichero cabecera llamado **cab.h**. (ambos módulos contendrán una sentencia **#include cab.h** al principio del fichero).

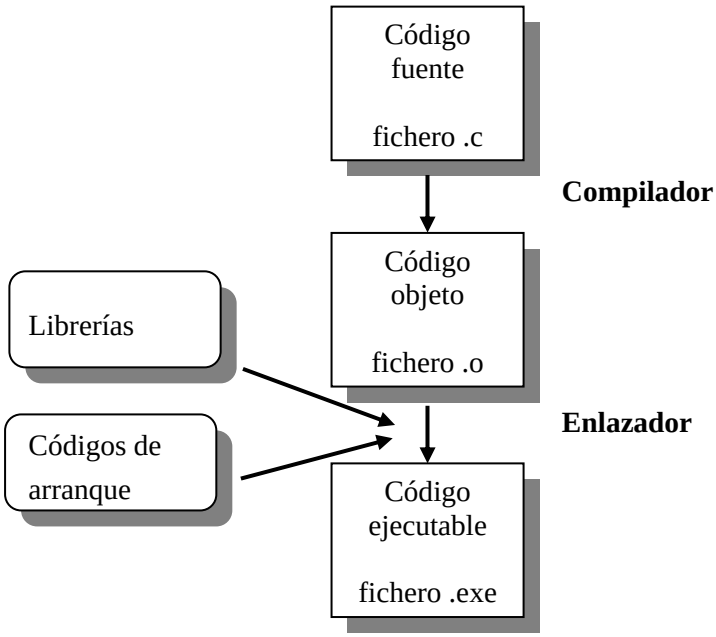
Para realizar el ejecutable, creamos un fichero de proyecto **.dev** con los nombres de los dos módulos que componen el programa.

El fichero proyecto es usado por el compilador como guía: cada archivo especificado en el archivo **.dev** es compilado y enlazado para formar el ejecutable **.exe**

7.11 Compilación y enlazado (link)

La compilación es el proceso mediante el cual el código fuente, escrito en un determinado lenguaje de programación, es convertido a un código objeto, directamente entendible por la computadora.

Durante la compilación, el compilador chequea que no se produzcan errores. De producirse alguno no se generará el correspondiente fichero objeto **.o**.



Una vez compilados y generados todos los ficheros objetos **.o** entra en acción el enlazador o linkador.

El enlazador (linker) realiza las siguientes funciones:

- Combina los archivos objetos **.o** de la lista de enlace, con los ficheros objetos del sistema (códigos de arranque) y con las librerías del sistema **(.lib**) y genera un archivo ejecutable.
- Resuelve las referencias externas y direcciones de memoria.

Las referencias externas son producidas cuando el código de un archivo hace referencia al código situado en otro (bien porque llame a una función definida en otro módulo, o bien porque haga referencia a una variable **extern**).

Durante el proceso de enlace (link) pueden producirse errores si el linkador no es capaz de encontrar algún fichero **.o** y **.lib** o si se producen inconsistencia entre los diferentes ficheros objetos (por ejemplo, una variable **extern** definida en un fichero no está definida en ninguno de los demás módulos, o no coinciden sus tipos).

Examinemos el siguiente programa, compuesto de dos módulos:

```
/* MODULO 1 */
#include <stdio.h>

static int a = 6;
int b;
extern int c;

/* prototipo de funciones */
void ver(void);
int incr( );

int main( void) {
 printf("a: %d; b: %d; c: %d \n", a, b, c);
 incr( );
 ver( );
 return 0;
}
```

```
/* MODULO 2 */
#include<stdio.h>
extern int a;
int b = 1;

void ver(void) {
 printf("a: %d; b: %d \n", a, b);
}
```

Al compilar cada uno de estos módulos no se produce ningún error de compilación. El compilador se tiene que 'fiar' de lo indicamos en cada módulos ya que no puede saber a priori si las funciones y variables externas especificadas en cada módulo están declaradas e implementadas en el resto de módulos.

Esto sólo puede ser resuelto en el momento del enlazado del programa. Cuando el enlazador entra en acción y trata de resolver las diferentes referencias externas se producen estos errores:

```
multiple definition of 'b'
[Linker Error] undefined reference to 'incr( )'
[Linker Error] undefined reference to 'c'
[Linker Error] undefined reference to 'a'
```

1. En ambos módulos está duplicada la variable b: En ambos hemos usado una variable global con el mismo nombre (b).
2. La función incr() no está definida en ningún módulo: En la compilación no se produjo error por que el compilador 'esperaba' que dicha función estuviera definida en el otro módulo o en la biblioteca de funciones del C.
3. La variable c no está definida en ningún sitio: En MODULO1 indicamos que existe una variable global externa c definida en otro módulo, pero en MODULO2 no aparece definida.
4. La variable a no está definida en ningún sitio: En MODULO2 indicamos que existe una variable global a externa definida en otro módulo, pero en MODULO1 la variable global a es estática, por tanto sólo es visible en dicho módulo.

Recuerda: además de errores en tiempo de compilación y de ejecución pueden producirse errores en tiempo de linkado o enlace.

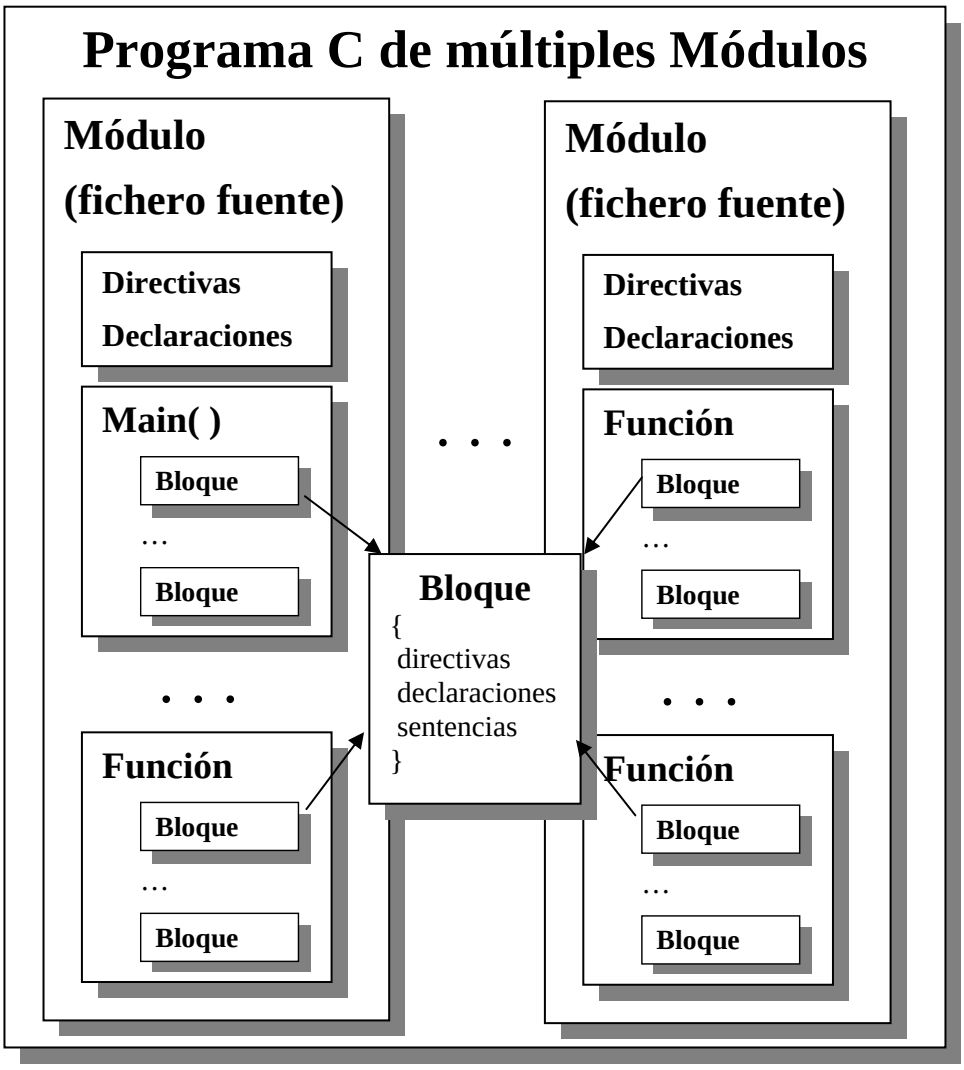
Una vez compilado los diferentes módulos y linkados correctamente se generará el correspondiente fichero ejecutable.

7.12 APENDICE: Visión general de un programa en C

Un programa C puede estar compuesto por uno o más módulos.

Cada módulo puede contener:

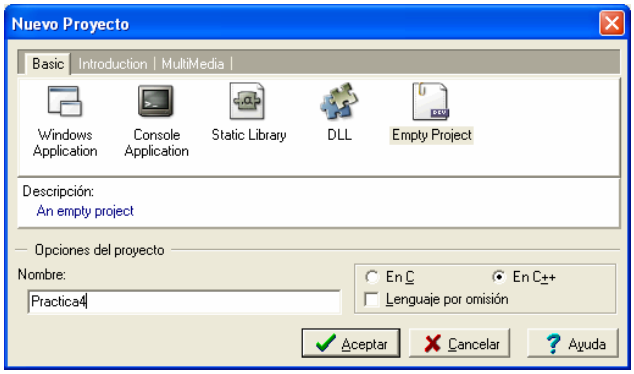
- Directivas: incluir ficheros (**#include**), definir constantes (**#define**)
- Declaraciones (prototipos de funciones, variables, etc).
- Funciones. Cada función puede contener: directivas, declaraciones y uno o más bloques de sentencias.
- Cada bloque pueden contener: directivas, declaraciones y una o más sentencias.
- Un programa debe tener al menos una función (**main( )**). En un programa con múltiples módulos sólo existe un **main( )**.



La estructura de cada módulo debe ser la siguiente:

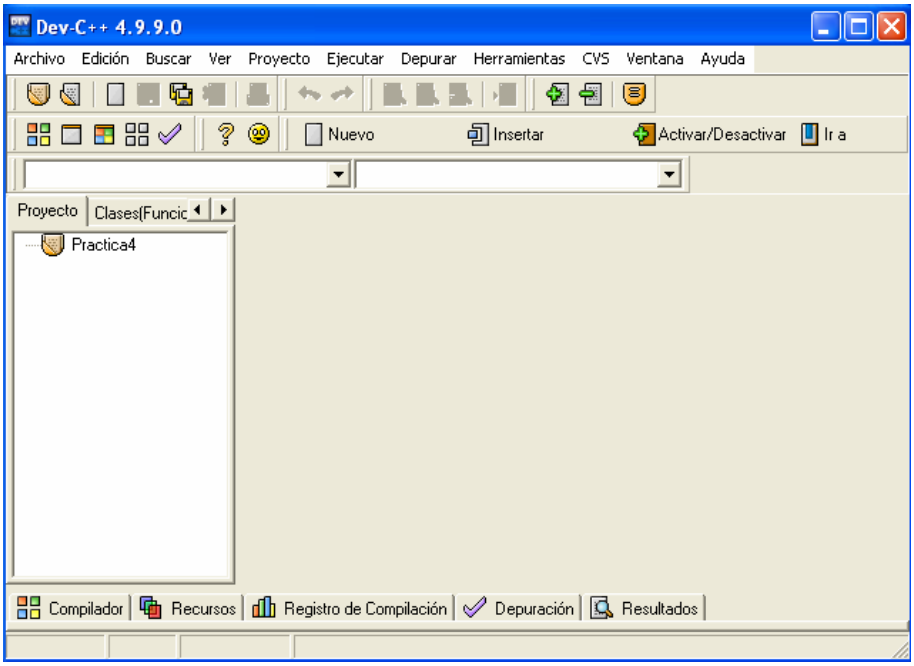
1. Cada módulo debe comenzar con un comentario que describa que hace el módulo. Debe incluirse el nombre del módulo, versión, fecha de 1ª realización y última modificación
2. A continuación debemos poner las diferentes directivas **#include** para incluir los prototipos, macros y constantes de los ficheros de librería y del propio usuario que usamos, así como los prototipos que no vienen allí especificados.
3. A continuación debemos poner las diferentes directivas **#define** de las constantes y macros usadas en el módulo.
4. Posteriormente las definiciones de las variables globales, globales **static**, y variables **extern** definidas en otros módulos (si no viene incluidas en ningún fichero cabecera).
5. Por último vendrán las diferentes funciones que componen el módulo. Éstas deben tener un comentario previo que describan que tarea realizan y que parámetros recibe, si la función modifica alguna variable global, así como la índole y el tipo de valor devuelto por la función (si devuelve alguno).

7.13 APENDICE: Creación de proyectos en Dev-C++



Creación de un proyecto:

1. Archivo – Nuevo Proyecto ➔ Aparece una ventana como la anterior.
2. En la pestaña Basic, seleccionar Proyecto Vacío (Empty Project), seleccionar como lenguaje base C o C++, asignar un nombre al proyecto y clic en Aceptar.
3. Se abrirá una ventana para seleccionar la carpeta donde se almacenará el proyecto.
4. Una vez realizado se creará un fichero **nombreProyecto.dev** en la carpeta seleccionada con la información sobre el proyecto y aparecerá el nombre del proyecto en la ventana de "Navegación de Proyectos" situada a la izquierda del entorno.



Añadir ficheros fuentes a un proyecto:

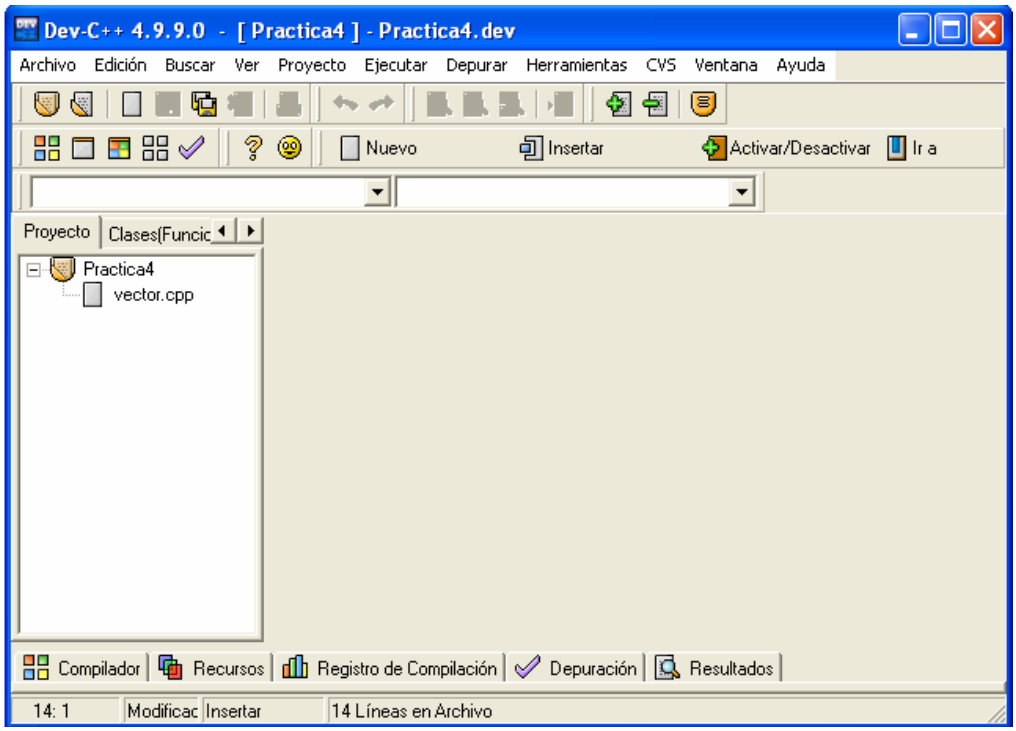
Para incluir un fichero:

- Proyecto – Nuevo Archivo Fuente ← Si el fichero fuente no existe
- Proyecto – Añadir Fichero Fuente ← Si el fichero existe.

En el primer caso, se creará un nuevo fichero para implementar el módulo deseado. Una vez realizado se guardará con el nombre .h ó .cpp.

En el segundo caso, aparecerá una ventana de navegación para seleccionar el fichero fuente, que se asignará al proyecto en la ventana de navegación de proyectos. Si deseamos abrir el fichero bastará hacer clic sobre él.

Si utilizamos Archivo – Nuevo Fuente, aparecerá un mensaje por si deseamos incluirlo al proyecto.



Después de añadir varios ficheros, aparecera en el entorno DEV C++ una pestaña abierta por cada fichero que intervenga en el proyecto.

Compilación y Ejecución.

- Ejecutar – Compilar ➔ Para compilar todos los ficheros modificados del proyecto desde la última compilación. Antes de compilar se guarda el contenido de todos los ficheros fuentes. Si no existen errores, se genera el ejecutable.
- Ejecutar – Compilar Archivo Actual ➔ Compila sólo el fichero acual.
- Ejecutar – Ejecutar ➔ Ejecuta el proyecto una vez haya sido compilado sin errores.
- Ejecutar – Compilar y Ejecutar ➔ Compila el proyecto y si no existen errores los ejecuta.
- Ejecutar – Reconstruir todo ➔ Realiza una compilación completa del proyecto, esto es compila los ficheros modificados y los no modificados.