
UNIVERSIDAD
DE GRANADA

# METODOLOGÍA DE LA PROGRAMACIÓN

## CUADERNO DE PRÁCTICAS


**Grado en Ingeniería Informática**
**GRUPO B**

**Francisco José Cortijo Bon**
cb@decsai.ugr.es

**2021/2022**

Departamento de
Ciencias de la Computación
e Inteligencia Artificial

Autor: **Francisco José Cortijo Bon** (cb@decsai.ugr.es)

*"Lo que tenemos que aprender a hacer, lo aprendemos haciéndolo".*
*Aristóteles*

*"In theory, there is no difference between theory and practice. But, in practice, there is".*
*Jan L. A. van de Snepscheut*

*"The gap between theory and practice is not as wide in theory as it is in practice".*

*"Theory is when you know something, but it doesn't work. Practice is when something works, but you don't know why. Programmers combine theory and practice: Nothing works and they don't know why".*

# Índice general

## SESIONES DE PRÁCTICAS



## RELACIONES DE PROBLEMAS



## GUIONES DETALLADOS

# Sesión 1

# El modelo de compilación en C++. El programa g++

## ► *Objetivos*

1. Conocer los distintos tipos de ficheros que intervienen en el proceso de compilación de programas en C++.
2. Conocer cómo se relacionan los diferentes tipos de ficheros que intervienen en el proceso de compilación de programas en C++.
3. Conocer el programa gcc/g++ y saber cómo trabaja en las distintas etapas del proceso de generación de un archivo ejecutable a partir de uno o más ficheros fuente.

## ► *Actividades a realizar en casa*

***Actividad previa: Lectura de la documentación de la primera práctica.***

Lea el **guión detallado** (página GD. 1) de la práctica:

***El modelo de compilación en C++. El programa*** g++

sobre el que se va a trabajar en esta sesión de prácticas.

---

***Actividad posterior: Ejercicio de modularización.***

Lea el documento:

***Ejercicio de moduralización. Parte 1. Modularización de funciones***

que encontrará en PRADO, en la sección **PRÁCTICA 1**, apartado **EJERCICIO DE MODULARIZACIÓN** y realice las actividades propuestas.

**Nota:** *No hay que entregar estas actividades*.

## ► Actividades a realizar en las aulas de ordenadores

En esta sesión se trabajará sobre el uso del programa g++ para compilar y enlazar, siguiendo las indicaciones del profesor.

## ► Recomendaciones

Es muy importante la lectura de los documentos indicados antes de la asistencia a la sesión práctica y la realización de la actividad de modularización después de celebrarse esta sesión práctica.

# Sesión 2

# Gestión de un proyecto software. El programa `make`

## ► *Objetivos*

1. Ser conscientes de la dificultad de mantener proyectos complejos (con múltiples ficheros y dependencias) si no se utilizan herramientas específicas.
2. Conocer el funcionamiento de la orden `make`.
3. Conocer la sintaxis de los ficheros *makefile* y cómo son interpretados por `make`.

## ► *Actividades a realizar en casa*

***Actividad previa: Lectura de la documentación de la segunda práctica.***

Lea el **guión detallado** (página GD. 27) de la práctica:

***Gestión de un proyecto software. El programa*** `make`

sobre el que se va a trabajar en esta sesión de prácticas.

---

***Actividad previa: Lectura de soluciones de ejercicios resueltos.***

En esta sesión no tendrá que entregar ningún ejercicio. No obstante deberá realizar las siguientes tareas relativas a ejercicios de la **Relación de Problemas I** (página RP. 1):

1. Resuelva el ejercicio 1. Intente resolverlo sin ejecutarlo. Después, compruebe si su resultado es correcto.
2. Lea la solución de los ejercicios 2, 3, 4 y 5.

   Las soluciones están disponibles en PRADO, en la sección **PRÁCTICA 2**, apartado **EJERCICIOS RESUELTOS DE LA RELACIÓN DE PROBLEMAS I**.

   Para desempaquetar el proyecto consulte en PRADO, sección **GNU/Linux**, apartado **Sobre el programa** `tar`.

***Actividad posterior: Ejercicio de modularización.***

Lea el documento:

- Ejercicio de moduralización. Parte 2. `make` **y** `makefile` **para gestión de proyectos**.

que encontrará en PRADO, en la sección **PRÁCTICA 2**, apartado **EJERCICIO DE MODULARIZACIÓN** y realice las actividades propuestas.

**Nota:** *No hay que entregar estas actividades*.

## ► *Actividades a realizar en las aulas de ordenadores*

En esta sesión se trabajará sobre el uso del programa `make` para gestionar proyectos, siguiendo las indicaciones del profesor.

## ► *Recomendaciones*

Es muy importante la lectura de los documentos y ejercicios resueltos indicados antes de la asistencia a la sesión práctica y la realización de la actividad de modularización después de celebrarse esta sesión práctica.

# Sesión 3

# Modularización. Bibliotecas. El programa `ar`. Punteros (1)

## ► *Objetivos*

Respecto a la modularización y bibliotecas los objetivos de esta sesión práctica son:

1. Conocer las ventajas de la modularización.
2. Saber cómo organizar un proyecto software en distintos ficheros y cómo se gestionan usando un fichero *makefile*.
3. Entender el concepto de *biblioteca* en programación.
4. Conocer el funcionamiento de la orden `ar`.
5. Saber cómo enlazar ficheros de biblioteca.
6. Apender a gestionar y enlazar bibliotecas en ficheros *makefile*.

Los ejercicios a resolver en esta práctica tienen como objetivo practicar con el paso de *arrays* a funciones y su gestión mediante punteros. Concretamente:

1. Escribir funciones que reciben/devuelven punteros. En particular, punteros que contienen la dirección de memoria de algún elemento de un vector.
2. Modularizar en diferentes ficheros la solución a un problema y gestionar el proyecto empleando un fichero *makefile*.

## ► Actividades a realizar en casa

### *Actividad previa: Resolución de problemas obligatorios. Punteros (I)*

Se trabajará sobre la **Relación de Problemas I** (página RP. 1). Concretamente, se trata de resolver los ejercicios **obligatorios** siguientes.

6 (Posición del mayor) `I_PosMayor_Basico.cpp`
7 (Ordenar un vector) `I_OrdenaVector.cpp`
8 (Ordena y mezcla vectores) `I_OrdenayMezclaVectores.cpp`

Modularice las funciones frecuentes que gestionan arrays de enteros en los ficheros fuente `ProcesamientoArrayInt.cpp` y `ProcesamientoArrayInt.h`. Por ejemplo, las funciones siguientes son buenas candidatas a estar en este módulo:

```
void MuestraVector (const char *msg, int *p, int n_datos,
                    int datos_linea);
void RellenaVector (int *p, int num_datos, int min_aleat,
                    int max_aleat);
void OrdenaSeleccion   (int *v, int pos_inic, int pos_fin);
void OrdenaInsercion   (int *v, int pos_inic, int pos_fin);
void OrdenaIntercambio (int *v, int pos_inic, int pos_fin);
```

Esta lista no es exhaustiva, sólo orientativa. Este módulo irá actualizándose.

Para poder generar correctamente los tres ejecutables pedidos será preciso escribir un fichero makefile, al que llamarán `makefile_03.mak`.

**Nota:** *Hay que entregar estas actividades. Se habilitará una entrega en PRADO.*

---

### *Actividad previa: Lectura de la documentación de la tercera práctica.*

Lea el **guión detallado** (página GD. 47) de la práctica:

***Modularización de software. Bibliotecas. El programa*** `ar`

sobre el que se va a trabajar en esta sesión de prácticas.

---

***Actividad posterior: Ejercicio de modularización.***

Lea los documentos:

- Ejercicio de moduralización. Parte 3. **BIBLIOTECAS. EL PROGRAMA** `ar` .
- Ejercicio de modularización. Parte 4. **MODULARIZACIÓN DE CLASES**.

que encontrará en PRADO, en la sección **PRÁCTICA 3**, apartado **EJERCICIO DE MODULARIZACIÓN** y realice las actividades propuestas en ellos.

**Nota:** *No hay que entregar estas actividades*.

---

***Actividad posterior: Resolución de problemas optativos. Punteros (I)***

Se trabajará sobre la **Relación de Problemas I** (página RP. 1).
Concretamente, se trata de modificar la solución de los ejercicios 2, 3, 4 y 5.

Modularice la **clase** `GeneradorAleatorioEnteros` en los ficheros:

- `GeneradorAleatorioEnteros.cpp`, y
- `GeneradorAleatorioEnteros.h`

y reescriba las soluciones proporcionadas de esos ejercicios con la clase `GeneradorAleatorioEnteros` modularizada en una biblioteca.

**Nota:** *No hay que entregar estas actividades*.

## ► *Actividades a realizar en las aulas de ordenadores*

En esta sesión se trabajará sobre la creación de bibliotecas con el programa `ar` y se construirán programas que usan bibliotecas. El trabajo se realizará siguiendo las indicaciones del profesor.

## ► *Recomendaciones*

Todas las soluciones deben estar correctamente escritas, **comentadas** y modularizadas.

Es muy importante la lectura de los documentos indicados antes de la asistencia a la sesión práctica y la realización de las actividades propuestas después de celebrarse esta sesión práctica.

## ► *Normas detalladas*

1. Deberá entregar únicamente el fichero `MP_03.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_03.mak`.
2. **MUY IMPORTANTE:** También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.
3. **No** incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter ~
4. No se admitirán otras soluciones que no cumplan escrupulosamente las normas indicadas.

# Sesión 4

# Punteros (2)

## ► *Objetivos*

Los ejercicios a resolver en esta práctica tienen como objetivo fundamental practicar con el paso de vectores y matrices a funciones, y su gestión con punteros. Trabajaremos con cadenas de caracteres clásicas (*cadenas tipo C*). Los objetivos detallados son:

1. Escribir funciones que reciben/devuelven punteros. En particular, punteros que contienen la dirección de algún elemento del vector (no necesariamente el primero).
2. Escribir funciones que procesan vectores y matrices.
3. Entender cómo se declaran y definen cadenas clásicas y la necesidad de reservar suficiente espacio (en esta práctica en tiempo de compilación) para su procesamiento correcto.
4. Entender la importancia de gestionar correctamente el carácter `'\0'` y tener claro que debe estar presente en toda cadena clásica.

Finalmente, como objetivo general del curso continuamos practicando la modularización en ficheros y la gestión de proyectos empleando un fichero `makefile`.

## ► *Actividades a realizar en casa*

***Actividad: Lectura de soluciones de ejercicios resueltos.***

Lea la solución de los ejercicios resueltos que podrá encontrar en PRADO.

***Actividad: Resolución de problemas.***

Se trabajará sobre la **Relación de Problemas I** (página RP. 1). Concretamente, se trata de resolver los ejercicios **obligatorios** siguientes.

9 (Elecciones - Ley D'Hont) `I_Ley_DHont.cpp`
10 (Descomposición en factores primos) `I_EratostenesFactoresPrimos.cpp`
11 (Ordenar matrices) `OrdenaMatrizEnteros.cpp`
20 (Eliminar espacios en una cadena clásica) `EliminarBlancos.cpp`
23 (Delimitar palabras en cadenas) `I_EncuentraPalabras.cpp`

Para poder generar correctamente los tres ejecutables pedidos será preciso escribir un fichero makefile, al que llamarán `makefile_sesion04.mak`.

## ► *Actividades a realizar en las aulas de ordenadores*

En esta sesión se trabajará sobre los ejercicios propuestos. Los estudiantes defenderán las soluciones entregadas.

## ► *Recomendaciones*

Todas las soluciones deben estar correctamente escritas, **comentadas** y modularizadas.

## ► *Normas detalladas*

1. Deberá entregar únicamente el fichero `MP_04.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_04.mak`.
2. **MUY IMPORTANTE:** También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.
3. **No** incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter ~
4. No se admitirán otras soluciones que no cumplan escrupulosamente las normas indicadas.

# Ficheros de texto (aproximación básica)

Un fichero de *texto* está formado por una serie indeterminada de caracteres, delimitado en su final por un *indicador de fin de archivo*. Entre los caracteres del fichero puede aparecer el carácter *nueva línea* que hace que al visualizar el contenido del fichero éste aparezca dispuesto en líneas separadas. Evidentemente, no se requiere que todas las líneas tengan la misma longitud. De hecho, algunos editores de texto nos permiten "visualizar" los saltos de línea: hágalo y verá cómo las líneas tienen longitudes diferentes.

Como sabemos, cuando se desea mostrar una línea en la consola, debemos marcar el final de la línea enviando el carácter `\n` (nueva línea) o usando el manipulador `endl`:

```
cout << "¡Hola, mundo!\n";    cout << "¡Hola, mundo!" << endl;
```

Así, podemos entender que cada *línea* de un fichero de texto está delimitada en su final por el carácter `\n`. Este mismo carácter es el que se genera al pulsar la tecla `ENTER` cuando se están leyendo caracteres. Las funciones de lectura, no obstante, no suelen incluir ese carácter en el valor leído sino que entienden que ese carácter marca (sin formar parte de él) el valor que se lee. Así, el método de la clase `istream`:

```
istream & getline (char *p, int n);
```

Lee, como mucho $\mathtt{n}-1$ caracteres de la entrada estándar. Lo habitual es que se termine cuando se pulsa ENTER (`'\n'`) y entonces los caracteres leidos se copian (por orden) a partir de la dirección de memoria guardada en `p`, salvo el `'\n'`, que lo sustituye por `'\0'`. La memoria referenciada por `p` debe estar reservada y ser suficiente.

Así, el siguiente código lee líneas de la entrada estándar y las muestra. Esto se repite hasta que se encuentra la marca de *fín de fichero* (manualmente -teclado- se introduce con `Ctrl+D`)

```
// El array "cadena" tiene "TOPE_LINEA" casillas disponibles
const int TOPE_LINEA = 200;
char cadena[TOPE_LINEA];

// Leer líneas y mostrarlas
while (cin.getline(cadena, TOPE_LINEA)) {
   cout << cadena << endl;
}
```

Un programa que contenga el código anterior y se ejecute **redirigiendo la entrada estándar** (`cin`) tomando los datos desde un fichero, mostrará el contenido de dicho fichero, línea a línea. O sea, en cada iteración del ciclo `while` se lee una línea del fichero y se muestra. Si la lectura "fracasa" (encuentra el indicador de fin de fichero) **no** se vuelve a entrar al ciclo `while`.

UNIVERSIDAD DE GRANADA

# METODOLOGÍA DE LA PROGRAMACIÓN

## RELACIONES DE PROBLEMAS

**Grado en Ingeniería Informática**
**GRUPO B**

**Francisco José Cortijo Bon**
cb@decsai.ugr.es

**2021/2022**

Departamento de Ciencias de la Computación e Inteligencia Artificial

# PROBLEMAS I. *Punteros. Funciones y punteros*

1. Describir la salida de los siguientes programas:

*a)*

```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p;

  a = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

*b)*

```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p;

  *p = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

*c)*

```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p = &a;

  *p = *p * a;
  if (a == *p)
    cout << "a es igual a *p" << endl;;
  else
    cout << "a es diferente a *p" << endl;
  return 0;
}
```

*d)*

```
#include <iostream>
using namespace std;

int main (){
  int a = 5, *p = &a, **p2 = &p;

  **p2 = *p + (**p2 / a);
  *p = a+1;
  a = **p2 / 2;
  cout << "a es igual a: " << a << endl;
  return 0;

}
```

2. Declare una variable `v` como un vector de 1000 enteros. Escriba un programa que recorra el vector y modifique todos los enteros *negativos* cambiándolos de signo.

   No se permite usar el operador `[]`, es decir, el recorrido se efectuará usando aritmética de punteros y el bucle se controlará mediante un contador entero.

   **Nota**: Para inicializar aleatoriamente el vector puede emplear la clase `GeneradorAleatorioEnteros` que puede encontrar en PRADO.

3. Modifique el código del problema 2 para controlar el final del bucle con un puntero a la posición siguiente a la última.

4. Con estas declaraciones:

```
const int TOPE = 100;
float v1 [TOPE] = {2,3,8,22,44,88,99,100,101,255,665};
float v2 [TOPE] = {1,3,4,5,6,25,87,89,99,100,500,1000};
float res [2*TOPE];

int tam_v1=11, tam_v2=12;      // 0 <= tam_v1, tam_v2 < TOPE
int tam_res = tam_v1+tam_v2;   // 0 <= tam_res < 2*TOPE
```

   Escribir un programa para mezclar, *ordenadamente*, los valores de `v1` y `v2` en el vector `res`.

   **Nota:** Observad que `v1` y `v2` almacenan valores *ordenados* de menor a mayor.

   **No se puede usar el operador `[]`. En definitiva, debe usar aritmética de punteros**.

5. Consideremos un vector `v` de números reales de tamaño `TOPE`. Supongamos que se desea dividir el vector en dos secciones: la primera contendrá a todos los elementos menores o iguales al primero y la otra, los mayores.

   Para ello proponemos un algoritmo que consiste en:

   - Colocamos un puntero al principio del vector y lo adelantamos mientras el elemento apuntado sea menor o igual que el primero.
   - Colocamos un puntero al final del vector y lo atrasamos mientras el elemento apuntado sea mayor que el primero.
   - Si los punteros no se han cruzado, es que se han encontrado dos elementos "mal colocados". Los intercambiamos y volvemos a empezar.
   - Este algoritmo acabará cuando los dos punteros "se crucen", habiendo quedado todos los elementos ordenados según el criterio inicial.

   Escriba un programa que declare una constante (`TOPE`) con valor 20 y un vector de reales con ese tamaño, lo rellene con números aleatorios entre 0 y 100 y lo reorganice usando el algoritmo antes descrito.

6. Escribir una función que recibe un vector de números enteros y dos valores enteros (que indican las posiciones de los extremos de un intervalo sobre ese vector). La función devuelve **un puntero** al elemento mayor dentro de ese intervalo.

   La función tendrá como prototipo:

```
int * PosMayor (int *pv, int izda, int dcha);
```

   donde `pv` contiene la dirección de memoria de una casilla del vector e `izda` y `dcha` son los extremos del intervalo entre los que se realiza la búsqueda del elemento mayor.

   Considere la siguiente declaración:

```
const int TOPE = 100;
int vector [TOPE];
```

Escriba un programa que rellene aleatoriamente el vector (completamente o una parte, pero siempre desde el principio) y que calcule el mayor valor entre dos posiciones dadas. El programa pedirá:

*a*) el número de casillas que se van a rellenear con números aleatorios,

*b*) las posiciones entre las que se va a calcular el mayor valor.

Considere todas las situaciones de error que puedan ocurrir y busque soluciones razonables antes que abortar la ejecución del programa.

**Nota:** Modularice la solución con funciones.

7. Escriba un programa que rellene aleatoriamente un vector (completamente o una parte, pero siempre desde el principio) y lo **ordene** (usando los tres algoritmos conocidos) entre dos posiciones dadas. El programa pedirá:

   *a*) el número de casillas que se van a rellenear con números aleatorios,

   *b*) las posiciones entre las que se va a ordenar el vector.

   Considere todas las situaciones de error que puedan ocurrir y busque soluciones razonables antes que abortar la ejecución del programa.

   Implemente tres funciones para la ordenación:

```
void OrdenaSeleccion (int *v, int pos_inic, int pos_fin);
void OrdenaInsercion (int *v, int pos_inic, int pos_fin);
void OrdenaIntercambio (int *v, int pos_inic, int pos_fin);
```

8. En este ejercicio se combinan las soluciones de los ejercicios 4 y 7.

   Con estas declaraciones:

```
const int TOPE = 100;
int v1 [TOPE],  int v2 [TOPE], int res [2*TOPE];

int tam_v1, tam_v2;  // 0 <= tam_v1, tam_v2 < TOPE
int tam_res; // 0 <= tam_res < 2*TOPE
```

   Escriba un programa que pida el número de casillas que se van a ocupar de los vectores `v1` y `v2`, los rellene aleatoriamente (completamente o una parte, pero siempre desde el principio), los **ordene** y finalmente los **mezcle** sobre el vector `res`.

   Considere todas las situaciones de error que puedan ocurrir y busque soluciones razonables antes que abortar la ejecución del programa.

   Use **funciones** (al menos para mostrar, ordenar y mezclar los vectores).

9. *(Basado en el Examen de FP - Febrero de 2017)*

   El sistema (o ley) **D'Hondt** es el método que se utiliza en España para asignar los escaños asignados a una circunscripción electoral.

   Se quiere construir un programa que lea el número total de escaños a distribuir, el número de partidos que han participado en las elecciones y los votos obtenidos por cada uno de ellos. El programa mostrará cuántos escaños obtuvo cada partido.

   La asignación de los escaños se hace a través de un proceso iterativo en el que en cada iteración se asigna un escaño a un partido y así hasta llegar al número total de escaños a repartir. Se calculan cocientes sucesivos para cada partido político. La fórmula de los cocientes es: $cociente = V_i/(S_i + 1)$ donde $V_i$ es el número total de votos obtenidos por el partido $i$ y $S_i$ es el número de escaños que ha obtenido hasta el momento el partido $i$.

   El número de votos recibidos por cada partido se divide sucesivamente por cada uno de los divisores, desde 1 hasta el número total de escaños a repartir. La asignación de cada escaño se determina hallando el **máximo** de los cocientes y asignando a cada uno un escaño, hasta que estos se agoten.

   Consulte aquí para más detalles.

   **Ejemplo.** Supongamos unas elecciones a las que se presentan $n = 4$ partidos, entre los que deben repartirse 5 escaños. La asignación de escaños se hace como sigue: Cada escaño se asigna al partido cuyo cociente sea máximo (en el ejemplo, se indica con * y en negrita).

| | Partido A | Partido B | Partido C | Partido D |
|---|---|---|---|---|
| Votos | $V_A = 340000$ | $V_B = 280000$ | $V_C = 160000$ | $V_D = 60000$ |
| Escaño 1 | **(340000/1 =) 340000*** | (280000/1 =) 280000 | (160000/1 =) 160000 | (60000/1 =) 60000 |
| Escaño 2 | (340000/2 =) 170000 | **(280000/1 =) 280000*** | (160000/1 =) 160000 | (60000/1 =) 60000 |
| Escaño 3 | **(340000/2 =) 170000*** | (280000/2 =) 140000 | (160000/1 =) 160000 | (60000/1 =) 60000 |
| Escaño 4 | (340000/3 =) 113333 | (280000/2 =) 140000 | **(160000/1 =) 160000*** | (60000/1 =) 60000 |
| Escaño 5 | (340000/3 =) 113333 | **(280000/2 =) 140000*** | (160000/2 =) 80000 | (60000/1 =) 60000 |
| E. asignados | 2 | 2 | 1 | 0 |

   **Sugerencia**: represente la información de cada partido con un `struct`. Por ejemplo:

```
struct Partido {
   char sigla; // "Nombre" del partido
   long votos; // Votos recibidos
   int escagnos; // Escaños conseguidos
};
```

de manera que la información de todos los partidos podría almacenarse en un array, por ejemplo:

```
// La información de los partidos se guarda en un
// array de datos "Partido"

const int  CAPACIDAD = 20;
Partido resultado[CAPACIDAD];
```

Dispone de una solución a este problema publicada en PRADO. Esta solución, no obstante, no usa funciones. Tome esa solución como base y escriba la nueva solución **usando funciones** para:

- Inicializar un array de datos de tipo `Partido` con las siglas de los partidos (por simplicidad `A`, `B`, etc.) y leer los votos obtenidos por cada partido.
- Calcular los escaños que se asignan a cada partido con la Ley D'Hont.
- Mostrar el resultado.

10. *(Basado en el Examen de FP - Septiembre 2012)*

    La ***criba de Eratóstenes*** (Cirene, 276 a. C. Alejandría, 194 a. C.) es un algoritmo que permite hallar todos los números primos menores que un número natural dado $n$.

    El procedimiento "manual" consiste en escribir todos los números naturales comprendidos entre 2 y $n$ y *tachar* los números que *no* son primos de la siguiente manera: el primero (el 2) se declara primo y se tachan todos sus múltiplos; se busca el siguiente número entero que no ha sido tachado, $p$, se declara primo y se procede a tachar todos sus múltiplos, y así sucesivamente. El proceso de criba termina cuando el cuadrado de $p$ es mayor que $n$.

    Si `MAX_PRIMO` es el mayor número que se va a considerar (establezca su valor a 5000, p.e.) escribir un programa que lea un valor $n$ ($1 \leq n \leq$`MAX_PRIMO`) y calcule y muestre todos los primos menores o iguales que $n$.

    Recomendaciones:

    *a*) Para realizar la criba use el vector `es_primo` de datos `bool` de manera que si `es_primo[i]` es `true` entonces, $i$ es primo.

    *b*) El resultado (los números primos calculados) se guardarán en el vector `primos`, de datos `int`, almacenados de manera consecutiva, sin huecos. Se establece un tamaño máximo para el vector de `MAX_DATOS` casillas (piense en un valor sensato para `MAX_DATOS` basándose en `MAX_PRIMO`).

    *c*) Tenga en cuenta las posibles situaciones de error, detéctelas y actúe correctamente.

**Aplicación: Descomposición en factores primos de un número.**

El procedimiento descrito anteriormente proporciona, simplemente, una colección de números primos.

Ahora deberá realizar un programa que lea un número indeterminado de enteros positivos (finalizar cuando se introduce un cero) y exprese cada uno de ellos en base al producto de sus factores primos.

Exprese el resultado como el producto de potencias. Por ejemplo:

```
360 = 2^3 * 3^2 * 5
121 = 11^2
11 = 11
```

Para la solución **deberá considerar únicamente los factores primos**. Para ello, en primer lugar deberá calcular (y guardar en un array) los números primos que va a utilizar para calcular la descomposición. Después ya podrá leer y valores y calcular y mostrar su descomposición.

Recomendamos usar un struct del tipo:

```
struct Pareja {
   int primo;
   int potencia;
};
```

para guardar cada pareja primo-potencia. Cada una de estas parejas se guardará en una casilla del array `descomposicion`:

```
const int CASILLAS_DESCOMPOSICION = 50;
Pareja resultado[CASILLAS_DESCOMPOSICION] = {{0,0}};
int utilizados_descomposicion = 0;
```

Por ejemplo, para el valor 360 (`360 = 2^3 * 3^2 * 5`) el array `resultado` contendría 3 casillas útiles (`utilizados_descomposicion` vale 3) y contendría:

```
{{2,3}, {3,2}, {5,1}}
```

Dispone de una solución a este problema publicada en PRADO. Esta solución, no obstante, no usa funciones. Tome esa solución como base y escriba la nueva solución **usando funciones**.

11. Escriba un programa que rellene aleatoriamente una matriz de enteros y sea capaz de ordenar sus filas y columnas. Deberá **reutilizar** funciones ya implementadas.

    Trabaje sobre una matriz declarada de la siguiente manera:

    ```
    const int NUM_FILAS = 40;
    const int NUM_COLUMNAS = 20;

    int matriz_base [NUM_FILAS][NUM_COLUMNAS] = {0};
    int util_filas_base = 0;
    int util_columnas_base = 0;
    ```

    Realmente trabajará sobre una **zona rectangular** de la matriz `matriz_base`.

    En primer lugar se rellenará la matriz completa con números aleatorios comprendidos entre 1 y 100.

    Para indicar la *submatriz* o zona rectangular de `matriz_base` el programa solicitará los números de las filas y columnas inicial y final que delimitan la zona rectangular de `matriz_base`.

    La matriz original no se modificará por lo que deberá usar sendas **copias** de `matriz_base` para poder ordenar las copias. Por ejemplo:

    ```
    int orden_filas [NUM_FILAS][NUM_COLUMNAS] = {0};
    int util_filas_orden_filas = 0;
    int util_columnas_orden_filas = 0;

    int orden_columnas [NUM_FILAS][NUM_COLUMNAS] = {0};
    int util_filas_orden_columnas = 0;
    int util_columnas_orden_columnas = 0;
    ```

    Los datos de la zona rectangular de `matriz_base` sobre la que vamos a trabajar se copiarán en `orden_filas` (resp. `orden_columnas`) a partir de su casilla $(0, 0)$, o sea, las casillas no usadas de `orden_filas` y `orden_columnas` estarán en la parte final de las filas y columnas (índices altos de filas y columnas).

    A continuación se ordenará `orden_filas`. Cada fila se ordenará independientemente. Después se ordenará `orden_columnas`. Cada columna se ordenará independientemente.

    Finalmente mostrará el contenido de las tres matrices. Organice la presentación de resultados para que se muestre únicamente las zonas de interés: 1) para la matriz `matriz_base` el contenido de la matriz completa y 2) para las matrices `orden_filas` y `orden_columnas` el contenido de la zona ordenada.

12. Las cadenas de caracteres (tipo "C", o cadenas "clásicas") son una buena fuente para ejercitarse en el uso de punteros. Una cadena de este tipo almacena un número indeterminado de caracteres (para los ejercicios basatará un valor siempre menor que 100) delimitados al final por el *carácter nulo* (`'\0'`).

    Escriba un programa que lea una cadena y localice la posición del primer *carácter espacio* (`' '`) en una cadena de caracteres "clásica". El programa debe indicar su posición (0: primer carácter, 1: segundo carácter, etc.).

    **Notas:**

    - La cadena debe recorrerse usando aritmética de punteros y sin usar ningún entero.
    - Usar la función `getline()` para la lectura de la cadena (Cuidado: usar el método público de `istream` sobre `cin`, o sea `cin.getline()`). Ver `http://www.cplusplus.com/reference/iostream/istream/getline/`

13. Consideremos una cadena de caracteres "clásica". Escriba un programa que lea una cadena y la imprima pero saltándose la primera palabra, *evitando escribirla carácter a carácter*.

    Considere que puede haber ninguna, una o más palabras, y si hay más de una palabra, están separadas por caracteres separadores (ver función `isspace`).

14. Considere una cadena de caracteres "clásica". Escriba la función `longitud_cadena`, que devuelva un *entero* cuyo valor indica la longitud de la cadena: el número de caracteres desde el inicio hasta el carácter nulo (no incluído).

    Tome como modelo la función `strlen` de `cstring`: la función accede a la cadena a través de un prámetro formal de tipo `const char *`

    ```
    int longitud_cadena (const char * cad);
    ```

    **Nota:** No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.

15. Escriba una función a la que le damos una cadena de caracteres y calcule si ésta es un palíndromo.

    **Nota:** No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.

16. Considere dos cadenas de caracteres "clásicas". Escriba la función `comparar_cadenas`, que devuelve un valor *entero* que se interpretará como sigue: si es *negativo*, la primera cadena es más "pequeña"; si es *positivo*, será más "grande"; y si es *cero*, las dos cadenas son "iguales".

    **Notas:**

    - Emplead como criterio para determinar el orden el código ASCII de los caracteres que se están comparando. Tome como modelo la función `strcmp` de `cstring`.
    - No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.

17. Considere dos cadenas de caracteres "clásicas". Escriba la función `copiar_cadena`, que copiará una cadena de caracteres en otra. El resultado de la copia será el primer argumento de la función. La cadena original (segundo argumento) **no** se modifica.

    Tome como modelo la función `strcpy` de `cstring`.

    **Notas:**

    - Se supone que hay suficiente memoria en la cadena de destino.
    - No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.

18. Considere dos cadenas de caracteres "clásicas". Escriba la función `encadenar_cadena`, que añadirá una cadena de caracteres al final de otra. El resultado se dejará en el primer argumento de la función. La cadena que se añade (segundo argumento) **no** se modifica.

    Tome como modelo la función `strcat` de `cstring`.

    **Nota:** Se supone que hay suficiente memoria en la cadena de destino.

19. Escriba la función `extraer_subcadena` a la que le damos una cadena de caracteres, una posición de inicio `p` y una longitud `l` sobre esa cadena. Queremos obtener una *subcadena* de ésta, que comienza en `p` y que tiene longitud `l`.

    ```
    char * extraer_subcadena (char * resultado, const char * origen,
                              int p, int l);
    ```

    **Notas:**

    - La cadena original **no** se modifica.
    - Se supone que hay suficiente memoria en la cadena resultado.
    - Si la longitud es demasiado grande (se sale de la cadena original), se devolverá una cadena de menor tamaño (la que empieza en `p` y llega hasta el final de la cadena).
    - No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.

20. Escriba funciones para eliminar separadores en una cadena:

    - `eliminar_blancos_iniciales`: los elimina del inicio de la cadena,
    - `eliminar_blancos_finales`: los elimina del final de la cadena,
    - `eliminar_blancos_extremos`: los elimina del principio y del final,
    - `eliminar_blancos_intermedios`: elimina los separadores internos, y
    - `eliminar_todos_blancos`: los elimina todos.

    Todas las funciones tienen un esquema común:

    ```
    char * eliminar_... (char * resultado, const char * origen);
    ```

    **Notas:**

    - La cadena original **no** se modifica.
    - Se supone que hay suficiente memoria en la cadena resultado.
    - No se puede usar el operador `[]`: use aritmética de punteros.
    - No puede usar ninguna función declarada en `cstring`.

    Escriba un programa que lea una serie indeterminada de *líneas* (finaliza cuando lee una línea que contiene **únicamente** `FIN`). Mostrará cada línea leida y el resultado de aplicar las funciones indicadas anteriormente.

    Finalmente indicará el porcentaje de separadores presentes en el texto original completo. Realice este cálculo en la función `main` sin recorrer, casilla a casilla, las cadenas leidas.

21. Escriba la función `inviertir_cadena`. Recibe una cadena de caracteres y devuelve una nueva cadena, resultado de invertir la primera.

    ```
    char * inviertir_cadena (char * resultado, const char * origen);
    ```

    **Notas:**

    - La cadena original **no** se modifica.
    - No se puede usar el operador `[]`: use aritmética de punteros.

    Escriba un programa que lea una serie indeterminada de *líneas* (finaliza cuando lee una línea que contiene **únicamente** `FIN`). Mostrará cada línea leida y el resultado de invertirla.

22. Escribir un programa que lea una cadena de caracteres, encuentre y registre el inicio de cada palabra, y finalmente muestre el primer carácter de cada palabra.

    Para registrar el inicio de cada palabra, usen un *array* de punteros a carácter (cada puntero contendrá la dirección del primer carácter de una palabra). En la figura 1.A mostramos el estado de la memoria para la función `main` justo antes de llamar a la función `encuentra_palabras` (para calcular el inicio de cada palabra). En la figura 1.B mostramos el estado de la memoria al empezar la ejecución de la función `encuentra_palabras`.

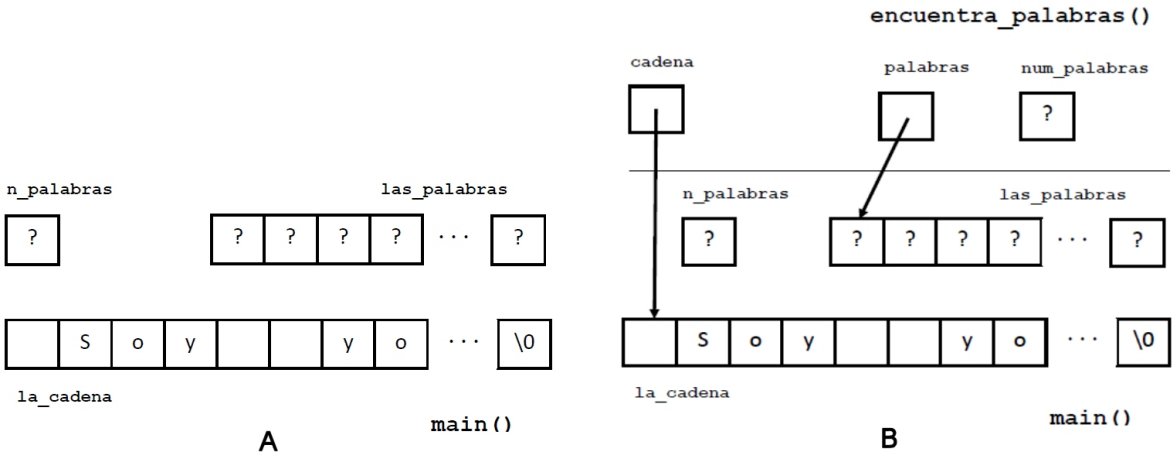



Figura 1: Estado de la pila A) antes de llamar a la función, `encuentra_palabras` y B) inmediatamente después de empezar la función

Los datos de la función `main` se han declarado:

```
const int MAX_CARACTERES = 100;
const int MAX_PALABRAS = 20;
char la_cadena[MAX_CARACTERES];
char * las_palabras[MAX_PALABRAS];
int n_palabras;
```

Escriba la función `encuentra_palabras` para calcular el principio de cada palabra.

```
int encuentra_palabras (char ** palabras, const char * cadena);
```

donde (ver figura 1.B):

- La función devuelve el número de palabras que hay en `cadena`.
- `cadena` es un puntero a la cadena donde se van a buscar las palabras, y

- `palabras` es un puntero a un *array* de datos `char *` de manera que `palabras[0]` contendrá la dirección de la primera letra de la primera palabra de `cadena`, `palabras[1]` contendrá la dirección de la primera letra de la segunda palabra de `cadena`, ...
- `num_palabras` es una variable local de la función.

En este ejemplo la función devolverá 2 y se llamó:

```
int num_palabras = encuentra_palabras(las_palabras, la_cadena);
```

En la figura 2.A mostramos el estado de la memoria al finalizar la función `encuentra_palabras`, justo antes de devolver el control a la la función `main` y en la la figura 2.B mostramos el estado de la memoria al volver el control a `main`.

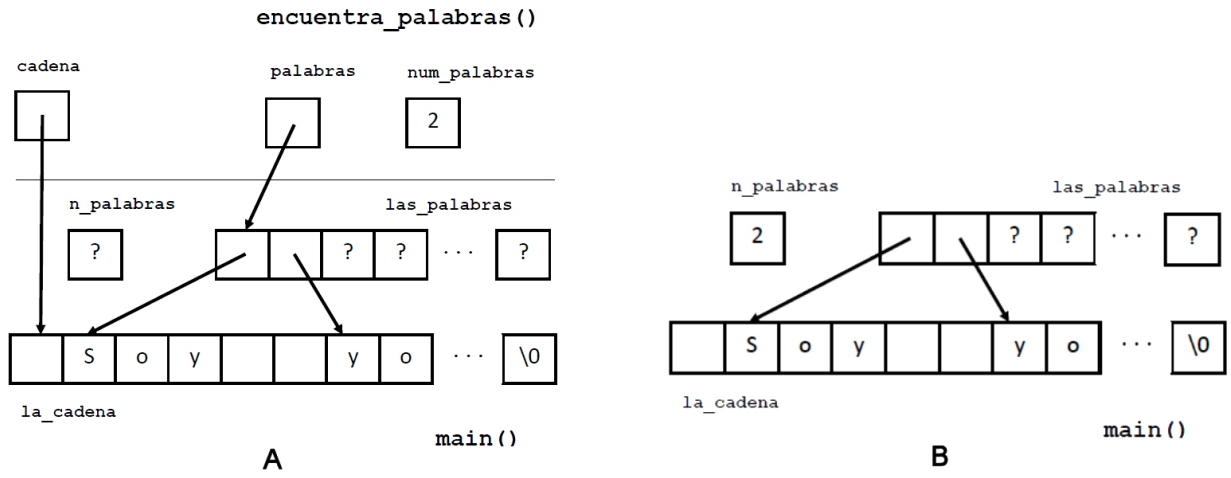



Figura 2: Estado de la pila A) antes de llamar a la función, `encuentra_palabras` y B) inmediatamente después de empezar la función

**Notas:**

- la zona de memoria referenciada por `cadena` **no** se modifica.
- Controle la ocupación del array de punteros `las_palabras`. Si se llenara, muestre un mensaje de aviso, y desde entonces ya no se considerarán más palabras, pero el programa no interrumpe su ejecución.

23. Este ejercicio es una ampliación del ejercicio 22. Ahora estamos interesados en saber no solo dónde empieza cada palabra, sino también dónde acaba.

    Ahora, el vector que usamos para registrar cada palabra es algo más elaborado. Se trata de un array de datos de tipo `info_palabra`, donde `info_palabra` se define:

```
struct info_palabra {
   char * inicio;
   char * fin;
};
```

    donde `inicio` y `fin` son punteros al carácter inicial y final, respectivamente, de cada palabra. Cada pareja de punteros delimita perfectamente una palabra.

    Con este objetivo, el prototipo de la nueva función `encuentra_palabras` será:

```
int encuentra_palabras (info_palabra * palabras, char * cadena);
```

    donde `palabras` es, ahora, un puntero a un *array* de datos `info_palabra` de manera que `palabras[0]` contendrá la dirección de inicio y fin de la primera palabra de `cadena`, `palabras[1]` contendrá la dirección de inicio y fin de la segunda palabra de `cadena`, ...

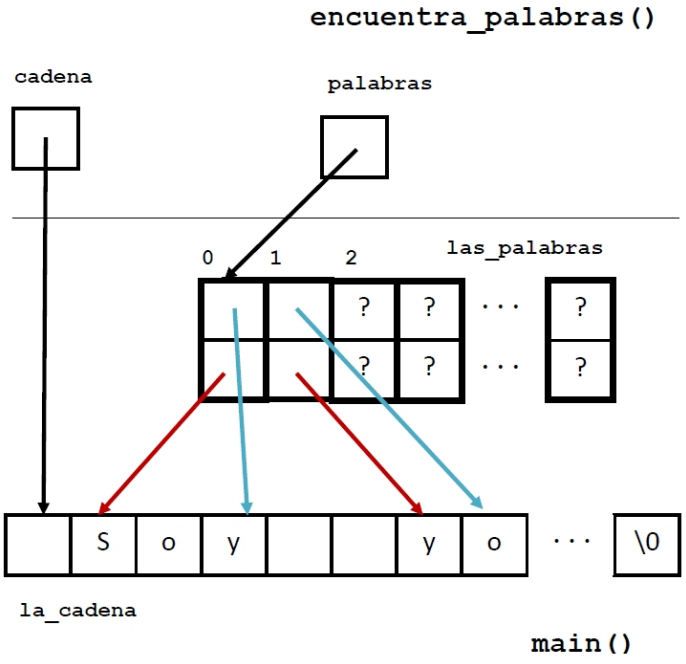



Figura 3: Estado de la pila antes de devolver el control a `main`

En la figura 3 mostramos el estado de la memoria antes de la finalización de la función `encuentra_palabras`. En este ejemplo la función devolverá 2 y se llamó:

```
......
char la_cadena[MAX_CARACTERES];
info_palabra las_palabras[MAX_PALABRAS];
......
int num_palabras = encuentra_palabras(las_palabras, la_cadena);
```

*a*) Escriba la (nueva) función `encuentra_palabras`.

*b*) Escriba la función `muestra_palabras` para mostrar las palabras registradas por la función `encuentra_palabras`. Decida la cabecera de la función.

**Notas:**

- la zona de memoria referenciada por `cadena` **no** se modifica.
- Controle la ocupación del array de punteros `las_palabras`. Si se llenara, muestre un mensaje de aviso, y desde entonces ya no se considerarán más palabras, pero el programa no interrumpe su ejecución.

Escriba un programa que lea una serie indeterminada de *líneas* (finaliza cuando lee una línea que contiene **únicamente** `FIN`). Mostrará cada línea leida, el número de palabras que la componen y las palabras individualmente. Por ejemplo, para la cadena `Escriba un programa` se mostrará:

```
Escriba un programa
   3 palabras --> {Escriba} {un} {programa}
```

24. Una palabra se define como **seudosimétrica** si, cuando se parte por la mitad, da como resultado dos palabras que tienen las mismas letras con las mismas frecuencias. Si la longitud de la palabra es un número impar, la decisión se toma sin considerar la letra central.

    Por ejemplo `gaga` es una palabra seudosimétrica. Las dos mitades `ga` y `ga` tienen los mismos caracteres con la misma frecuencia. Otros ejemplos son `rotor`, `aabccaba`, `xyzxy`. La palabra `abbaab` NO es seudosimétrica.

    Escriba la función

    ```
    bool es_seudosimetrica (const char * cadena);
    ```

    que comprueba si `cadena` es una palabra seudosimétrica. `cadena` debe estar formada por una sóla palabra (en otro caso devuelva `false`) aunque podría tener un número indeterminado de espacios iniciales y finales.

    **Sugerencia:** Utilice las funciones de los ejercicios 14, 19 y 20.

25. Queremos escribir un programa que procese una serie indeterminada de líneas (use un fichero de texto, redirigiendo la entrada estándar) y formatee su contenido generando un nuevo texto *rectangular*. Puede redirigir la salida a otro fichero. Considérelo.

    **Sugerencia:** En la página 15 explicamos cómo procesar (con redirección de entrada) un fichero de texto.

    Por ejemplo, si el fichero contiene:

```
/*********************************************************/
// METODOLOGIA DE LA PROGRAMACION
// GRADO EN INGENIERIA INFORMATICA
//
// (C) FRANCISCO JOSE CORTIJO BON
// DEPARTAMENTO DE CIENCIAS DE LA COMPUTACION E I.A.
//
// RELACION DE PROBLEMAS 1
//
// Fichero: I_DetectaLineasLargas.cpp
//
/*********************************************************/
#include <iostream>
#include <iomanip>
```

    podría generar los siguientes resultados:

```
/*****************************      ********************
// METODOLOGIA DE LA PROGRAMAC      TODOLOGIA DE LA PROG
// GRADO EN INGENIERIA INFORMA      ADO EN INGENIERIA IN
//!!!!!!!!!!!!!!!!!!!!!!!!!!!      !!!!!!!!!!!!!!!!!!!
// (C) FRANCISCO JOSE CORTIJO       ) FRANCISCO JOSE COR
// DEPARTAMENTO DE CIENCIAS DE      PARTAMENTO DE CIENCI
//!!!!!!!!!!!!!!!!!!!!!!!!!!!      !!!!!!!!!!!!!!!!!!!
// RELACION DE PROBLEMAS 1!!!!      LACION DE PROBLEMAS
//!!!!!!!!!!!!!!!!!!!!!!!!!!!      !!!!!!!!!!!!!!!!!!!
// Fichero: I_DetectaLineasLar      chero: I_DetectaLine
//!!!!!!!!!!!!!!!!!!!!!!!!!!!      !!!!!!!!!!!!!!!!!!!
/*****************************      ********************
!!!!!!!!!!!!!!!!!!!!!!!!!!!!      !!!!!!!!!!!!!!!!!!
#include <iostream>!!!!!!!!!!!      ude <iostream>!!!!!!
#include <iomanip>!!!!!!!!!!!!      ude <iomanip>!!!!!!
```

    En el primer caso se genera un texto con treinta caracteres de ancho, y en el segundo con veinte caracteres de ancho. En el primer caso el resultado se calcula a partir del primer carácter de cada línea (no se descarta ninguno) mientras que en el segundo caso se *saltan* los cinco primeros. Observe que los espacios finales se *completan* con el carácter `!`.

    Por hacer el programa más útil para su entrenamiento como programador considere:

    1.- No es preciso dar formato a todas las líneas sino únicamente a las primeras `TOPE_LINEAS_FORMAT` líneas.

2.- Guardar todas las *líneas* procesadas en una matriz de `char`:

```
char resultado[TOPE_LINEAS_FORMAT][TOPE_COLUMNAS_FORMAT+1];
```

Cada línea de la matriz es una cadena clásica de longitud `TOPE_COLUMNAS_FORMAT` (en la última columna se guarda el delimitador `'\0'`, por eso tiene una columna más).

3.- Tenga en cuenta que los tabuladores (`'\t'`) pueden confundirle al presentar el resultado. Podría emplear el carácter `'@'`, por ejemplo, para sustituir los tabuladores.

26. Escriba funciones para leer y devolver un número entero.

*a*) La primera lee un números entero sin restricciones. Su prototipo será:

```
int  LeeEntero (const char * titulo);
```

La lectura está etiquetada por `titulo`. La función:

I) leerá el número usando una cadena clásica,

II) comprobará que los caracteres son correctos. Se permiten:

- los dígitos `'0'`, `'1'`, ..., `'9'` y
- los caracteres + ó - (únicamente en la primera posición)

Evidentemente, deberá descartar los separadores iniciales y/o finales antes de la comprobación.

III) la convertirá a un valor entero.

Si todo es correcto, devolverá el dato `int`; si no lo es, volverá a pedirlo.

*b*) La segunda función (`LeeEnteroEnRango`) lee y devuelve un número entero que pertenece a un intervalo. La lectura estará etiquetada. La función puede recibir:

- dos argumentos (el título y un argumento `int`): admitirá valores entre 0 y el valor indicado.
- tres argumentos (el título y dos argumentos `int`): admitirá valores entre los dos indicados.

*c*) La tercera (`LeeEnteroMayorOIgual`) lee y devuelve un dato que debe ser mayor o igual que el indicado (`menor`). La lectura está etiquetada por `titulo`. Su prototipo es:

```
int LeeEnteroMayorOIgual (const char * titulo, int menor);
```

Probad las funciones escribiendo un programa.

27. Este ejercicio se basa en el ejercicio 6 de esta Relación de Problemas. Reutilizar el código que pueda ser aprovechado, especialmente la función `PosMayor`.

    Escriba un programa que rellene aleatoriamente el vector (completamente o una parte, pero siempres desde la casilla inicial) y que calcule el mayor valor entre dos posiciones:

    *a*) Si el programa se ejecuta sin argumentos, se rellenará completamente (`TOPE` casillas) y se calculará el mayor valor de todo vector (entre las casillas 0 y `TOPE`-1).

    *b*) Si el programa se ejecuta con un argumento (`n`), se rellenarán `n` casillas, y calculará el mayor valor entre ellas (entre las casillas 0 y `n`-1).

    *c*) Si el programa se ejecuta con dos argumentos (`n` y `d`), se rellenarán `n` casillas, y calculará el mayor valor entre las casillas 0 y `d`.

    *d*) Si el programa se ejecuta con tres argumentos (`n`, `i` y `d`), se rellenarán `n` casillas, y calculará el mayor valor entre las casillas `i` y `d`.

    *e*) Si el programa se ejecuta con más de tres argumentos muestra un mensaje de error y no hace nada más.

    **Nota:** Modularice la solución con funciones.

28. Escribir un programa que rellene (parcial o totalmente) dos vectores con números aleatorios `int`, los ordene y los mezcle usando dos versiones del algoritmo de mezcla:

    *a*) manteniendo todos los valores originales en la mezcla, y

    *b*) guardando una única copia de los valores repetidos en los dos vectores.

    Los vectores que se mezclan tienen una capacidad de `TAM`.

    - Si el programa se ejecuta sin argumentos, los dos vectores de entrada se ocuparán **completamente** (`TAM` casillas) y se rellenarán, con valores aleatorios entre 1 y 200 (ambos incluidos).
    - Si el programa se ejecuta con un argumento, los dos vectores de entrada se ocuparán se ocuparán **parcialmente**. El número de casillas ocupadas en ambos vectores será el valor que indiquemos con el único parámetro. Se generarán valores aleatorios entre 1 y 200 (ambos incluidos).
    - Si se ejecuta con dos argumentos, los dos vectores de entrada se ocuparán **parcialmente**. El número de casillas ocupadas en ambos vectores será el valor que indiquemos en el primer argumento. El segundo argumento indica el mayor valor aleatorio permitido, por lo que se generarán valores aleatorios entre 1 y ese valor.
    - Si se ejecuta con tres argumentos, los dos vectores de entrada se ocuparán **parcialmente**. El número de casillas ocupadas en ambos vectores será el valor que indiquemos en el primer argumento. Los otros dos argumentos indican el menor y mayor valor aleatorio permitido (no necesariamente en este orden).

En la función `main` declarar `v1` y `v2`, dos vectores de datos `int` con una capacidad de `TAM`.

Usar la función;

```
void RellenaVector (int v[], int util, int min, int max);
```

para rellenar `util` casillas del vector `v` con datos generados aleatoriamente. Los datos aleatorios están comprendidos entre `min` y `max` (ambos incluidos).

Usar una función como:

```
void OrdenaVector (int v[], int util);
```

para ordenar las `util` casillas del vector `v`. Usar el método que desee: burbuja, inserción o selección.

El contenido de un vector se mostrará usando la función:

```
void MuestraVector (char *mensaje, int v[], int util, int n);
```

Antes de presentar los datos del array `v` (que tiene `util` datos) se muestra `mensaje`. Los datos se muestran por líneas, separados por espacios, y en cada línea hay `n` datos (posiblemente la última línea tenga menos).

La mezcla ordenada de los dos vectores se hará con las funciones:

```
int MezclaVectores (int mezcla[], int v1[], int util_v1,
                    int v2[], int util_v2);
int MezclaVectoresSelectiva (int mezcla[], int v1[],
                    int util_v1, int v2[], int util_v2);
```

que mezclan ordenadamente los datos de los vectores `v1` y `v2`, que tienen `util_v1` y `util_v2` datos respectivamente.

- Se supone que `mezcla` tiene suficiente capacidad para albergar la mezcla (al menos `util_v1` + `util_v2` casillas).
- El valor devuelto es el número de casillas usadas en `mezcla`.
- `MezclaVectoresSelectiva` guarda una única copia de cada valor, descartando los valores repetidos.

**Las funciones no pueden usar el operador `[]` : Debe usar aritmética de punteros**.

29. Modificar el proyecto realizado en el ejercicio 28 de manera que ahora las funciones de mezcla quedan *resumidas* en una única función:

```
void MezclaVectores (int mezcla[], int &util_mezcla,
       int v1[], int util_v1, int v2[], int util_v2,
       const char * selectiva = "no");
```

    de manera que:

    - La referencia `util_mezcla` proporciona el número de casillas útiles de `mezcla`.
    - El parámetro (opcional, con valor por defecto) `selectiva` indica a la función si debe realizar una mezcla selectiva o no. El valor por defecto es `no`.
    - La función realizará una ordenación selectiva si se llama con los valores `si`, `SI`, `Si` ó `sI`. La mezcla será completa si se llama con cualquier otro valor (o ninguno).

    Reutilice el código escrito. Analice si es mejor reescribirlo para esta nueva versión.

30. En este ejercicio retomamos la clase `SecuenciaEnteros` con la que trabajamos en la asignatura **Fundamentos de Programación**. Considere la implementación (`SecuenciaEnteros_basico.cpp`) que proporcionamos en PRADO como la implementación básica para este ejercicio.

    1.- Modularice el código dado, proporcionando la biblioteca `libSecuenciaEnteros.a` y el fichero de cabecera asociado `SecuenciaEnteros.h`

    2.- Añada un constructor que reciba la dirección de memoria de un `int` (debería ser la dirección de memoria de una casilla de un array) y un valor `int` (debería ser un número de casillas de ese mismo array).

```
/*************************************************/
// Construye una secuencia con "n_datos" valores
// PRE: 0 <= n_datos <= TAMANIO
// PRE: A partir de "p" hay "n_datos" valores

SecuenciaEnteros (int * p, int n_datos);
```

    Creará una secuencia de `n_datos` valores. El primero será el que esta en la dirección indicada en `p`.

    3.- Añada un constructor que cree una secuencia *plana*.

```
/*************************************************/
// Construye una secuencia con "n_datos" valores iguales
// PRE: 0 <= n_datos <= TAMANIO

SecuenciaEnteros (int n_datos, int valor=0);
```

    Creará una secuencia de `n_datos` valores iguales. Si no se indica el segundo argumento, todos serán 0.

4.- Añada un constructor que cree una secuencia con números aleatorios.

```
/*****************************************************/
// Construye una secuencia con "n_datos" valores aleatorios
// PRE: 0 <= n_datos <= TAMANIO

SecuenciaEnteros (int n_datos, int min_aleat, int max_aleat);
```

El constructor creará una secuencia de `n_datos` valores aleatorios comprendidos entre `min_aleat` y `max_aleat`.

5.- Reescriba el método `Elimina` evitando el ciclo `for` que "desplaza" los valores hacia posiciones más bajas (use la función `memmove`).

6.- Reescriba el método `Inserta` evitando el ciclo `for` que "desplaza" los valores hacia posiciones más altas (use la función `memmove`).

7.- Unifique los métodos `Elemento` y `Modifica` en el método `Valor`:

```
/*****************************************************/
// Devuelve una ref. al elemento de la casilla "indice"
// PRE: 0 <= indice < total_utilizados

int & Valor (int indice);
```

8.- Añada el método `EsIgualA`:

```
/*****************************************************/
// Devuelve true si la secuencia implícita es igual a "otra"

bool EsIgualA (const SecuenciaEnteros & otra);
```

Deberá probar todos los métodos implementados para asegurarse de que son correctos (podría reutilizar algunos de los ejercicios de **Fundamentos de Programación**).

En cualquier caso, encontrará en PRADO una función `main` (en `I_DemoSecuenciaEnteros.cpp`) que trabaja sobre la clase `SecuenciaEnteros` y que le servirá para validar la implementación realizada.

31. Escriba la función

```
void Ordena (int *vec, int **ptr, int izda, int dcha);
```

que reorganiza los punteros de `ptr` de manera que recorriendo los elementos referenciados por esos punteros encontraríamos que están ordenados de manera creciente.

- Los elementos referenciados por los punteros son los elementos del vector `vec` que se encuentran entre las casillas `izda` y `dcha`.
- Los elementos de `vec` no se modifican.

Observe que el vector de punteros debe ser un parámetro de la función, y además debe estar reservado previamente con un tamaño, al menos, igual al del vector.

```
const int TOPE = 50; // Capacidad
int  vec [TOPE]; // Array de datos
int *ptr [TOPE]; // Indice al array "vector"
```

En la figura 4 mostramos el resultado de la función cuando se "ordena" el vector `vec` entre las casillas 0 y 5.

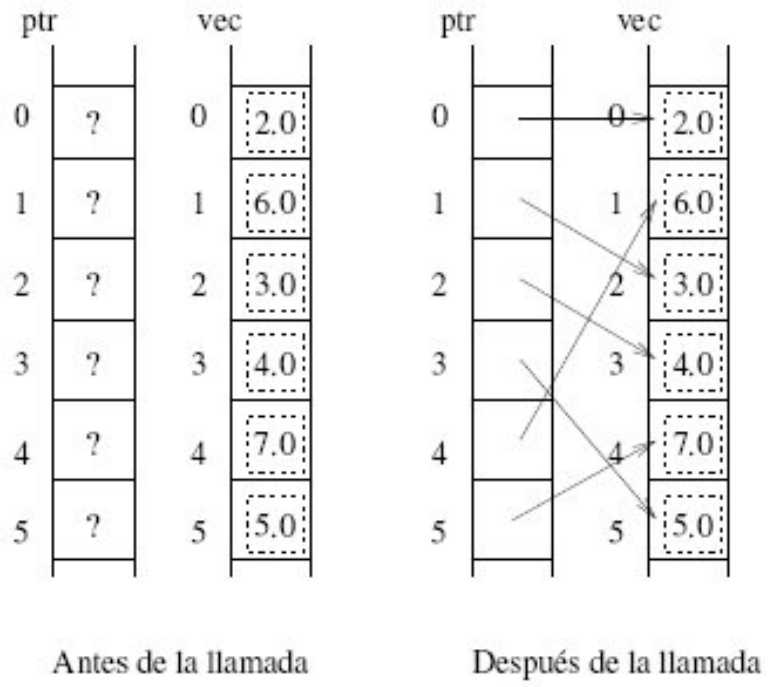



Figura 4: Resultado de ordenar el vector de punteros

Escriba un programa que haciendo uso de la función permita "ordenar" el vector entre dos posiciones:

- Si el programa se ejecuta sin argumentos "ordenará" todo vector.
- Si el programa se ejecuta con dos argumentos, "ordenará" el vector entre las dos posiciones dadas.
- En otro caso, muestra un mensaje de error y no hace nada más.

**Nota:** Iniciar aleatoriamente el vector `vec`.

**Nota:** Modularice la solución con funciones.

32. Escriba un programa que rellene completamente un vector con números aleatorios y que calcule el menor y el mayor valor entre dos posiciones dadas.

    Modularizar la solución usando, al menos, funciones para:

    *a*) Rellenar el vector con valores aleatorios entre dos valores comprendidos entre 1 y 500.

    *b*) Calcular el mínimo y el máximo valor del vector (o una parte de él) *con una función*. La función recibirá la dirección de inicio del vector y las posiciones inicial y final entre las que realizar la búsqueda. Devolverá (mediante **referencias**) los valores calculados.

    Intente generalizar la solución recibiendo los valores críticos (número de casillas, posiciones izquierda y derecha, etc.) en la línea de órdenes

33. Se quiere monitorizar los datos de ventas semanales de una empresa que cuenta con varias sucursales. La empresa tiene 100 sucursales y dispone de un catálogo de 10 productos. Algunas sucursales no han vendido ningún producto, y algún producto no se ha vendido en ninguna sucursal.

    Cada operación de venta se registra con tres valores: el identificador de la sucursal (número entero, con valores desde 1 a 100), el código del producto (un carácter, con valores desde `a` hasta `j`) y el número de unidades vendidas (un entero).

    El programa debe leer un número *indeterminado* de datos de ventas: la lectura de datos finaliza cuando se encuentra el valor `-1` como código de sucursal. El programa sólo procesa datos de venta de las sucursales que hayan realizado operaciones de ventas, por lo que no todas las sucursales ni productos aparecerán.

    **Recomendación:** Leer los datos utilizando la redirección de entrada. Usad para ello un fichero de texto como los disponibles en la página de la asignatura.

    Después de leer los datos de ventas el programa mostrará:

    *a*) El número total de operaciones de venta.

    *b*) La sucursal que más unidades ha vendido y cuántas unidades.

    *c*) Listado en el que aparecerán: código de sucursal y número total de productos vendidos en la sucursal. Aparecerán únicamente las sucursales que hayan vendido algún producto.

    *d*) El número de sucursales que hayan vendido algún producto.

    *e*) Número total de unidades vendidas (calculado como suma de las ventas por sucursales).

    *f*) Producto más vendido y cuántas unidades.

    *g*) Listado en el que aparecerán: código de producto y número total de unidades vendidas. Aparecerán únicamente los productos que hayan tenido alguna venta.

*h*) Cuántos tipos de producto han sido vendidos.

*i*) El número total de unidades vendidas (calculado como suma de las ventas por producto).

*j*) Tabla-resumen con toda la información. Por ejemplo:

```
    |    a    b    c    d    e    f    h
-------------------------------------------
  1 |    2   14    2    0   20    0    0 |   38
  2 |   20   21    0    0    0    0    0 |   41
  3 |    0   11   49    5    0    0    0 |   65
  4 |    0    0    0   42   10    0    0 |   52
  5 |    3    0   10    0   20   23    4 |   60
  7 |    0   15    0   12    0    8    0 |   35
  9 |   10   44    0    0    0    0    0 |   54
 10 |    0    7   30    0    0    0    0 |   37
-------------------------------------------
    |   35  112   91   59   50   31    4 |  382
```

**Nota:** Modularizar con funciones la solución desarrollada, de manera que cada una de las tareas a realizar las lleve a cabo una función.

Para poder guardar en memoria la información requerida para los cálculos proponemos una matriz bidimensional `ventas` con tantas filas como número (máximo) de sucursales y columnas como número (máximo) de productos. La casilla (*s*, *p*) de esta matriz guardará en número total de unidades vendidas por la sucursal *s* del producto *p* (o el número de de unidades vendidas del producto *p* en la sucursal *s*).

No se conoce a priori el número de operaciones de venta. Tampoco se conoce el número de sucursales que se van a procesar, ni el código de éstas (algunas sucursales puede que no hayan realizado ventas). Se sabe que los códigos de sucursales son números entre 1 y 100.

Tampoco se conoce a priori los productos que se han vendido, ni el código de éstos (algunos productos puede que no hayan vendido). Se sabe que los códigos de producto son caracteres entre `'a'` y `'j'`.

**Nota:** Emplead datos `int` para los índices de las filas y `char` para las columnas, de manera que se accede a las ventas del producto `'b'` por la sucursal 3, por ejemplo, con la construcción: `ventas[3]['b']`.

**Nota:** Aconsejamos que una vez leídos los datos, y actualizada la matriz `ventas`:

*a*) Usad un vector, `ventas_sucursal`, con tantas casillas como filas tenga la matriz `ventas`. Guardará el número total de unidades vendidas por cada sucursal.

*b*) Usad un vector, `ventas_producto`, con tantas casillas como columnas tenga `ventas`. Guardará el número total de unidades vendidas de cada producto.

34. *(Adaptación del examen de la convocatoria extraordinaria de FP - Febrero 2019)* Una **secuencia de ADN** es una secuencia de nucleótidos que pueden tomar uno entre cuatro valores: **A** (Adenina), **C** (Citosina), **G** (Guanina) y **T** (Timina). Para gestionar esta información, una **secuencia de ADN** individual se codificará en una cadena de caracteres clásica de una capacidad máxima (`MAXLONGITUD`) de 100 caracteres. Un ejemplo de secuencia de ADN sería el siguiente:

    ```
    T C G G G G A T T T C C
    ```

    Nuestro problema a resolver trata sobre un laboratorio que realiza lecturas de varias secuencias de ADN para diferentes células que comparten información genética. Se utiliza una matriz para almacenar un máximo de (`CAPACIDAD`) 2000 secuencias.

    Aunque las células comparten información genética, las secuencias correspondientes sufren mutaciones, de forma que, por ejemplo una **A** podría cambiar a una **T**. En la siguiente tabla se muestra un ejemplo con un conjunto de secuencias. En mayúsculas aparecen destacados los nucleótidos originales y en minúsculas, las posibles mutaciones (observe que realmente los nucleótidos de las secuencias siempre se representan con una mayúscula. Se han usado minúsculas en la tabla del ejemplo para resaltar la mutación)

    | | | | | | | | | | | | |
    |---|---|---|---|---|---|---|---|---|---|---|---|
    | **T** | **C** | **G** | **G** | **G** | **G** | g | **T** | **T** | **T** | t | t |
    | c | **C** | **G** | **G** | t | **G** | **A** | c | **T** | **T** | a | **C** |
    | a | **C** | **G** | **G** | **G** | **G** | **A** | **T** | **T** | **T** | t | **C** |
    | **T** | t | **G** | **G** | **G** | **G** | **A** | c | **T** | **T** | t | t |
    | a | a | **G** | **G** | **G** | **G** | **A** | c | **T** | **T** | **C** | **C** |
    | **T** | t | **G** | **G** | **G** | **G** | **A** | c | **T** | **T** | **C** | **C** |
    | **T** | **C** | **G** | **G** | **G** | **G** | **A** | **T** | **T** | c | a | t |
    | **T** | **C** | **G** | **G** | **G** | **G** | **A** | **T** | **T** | c | **C** | t |
    | **T** | a | **G** | **G** | **G** | **G** | **A** | a | c | **T** | a | **C** |
    | **T** | **C** | **G** | **G** | **G** | t | **A** | **T** | a | a | **C** | **C** |

    De acuerdo a la descripción anterior, escriba un programa que:

    *a*) Rellene dos tablas como las descritas anteriormente, referidas a dos series de experimentos sobre el mismo problema (las secuencias tendrán la misma longitud, aunque el número de secuencias en cada tabla puede ser distinto).

    *b*) Calcule la *secuencia de consenso* de una tabla.

    Implemente la función `SecuenciaConsenso` que calculará y devolverá la secuencia de ADN más probable a partir de la información almacenada en una tabla.

    Dicha secuencia es una secuencia de ADN (una cadena clásica) que se forma asignando a su $i$-ésima posición el nucleótido (A, C, G, T) que más veces se repite en la posición $i$ de todas las secuencias (es decir, en la columna $i$ de la tabla)

En caso de empate, se elige cualquiera de los repetidos. En el ejemplo anterior, la secuencia de consenso generada sería: `T C G G G G A T T T C C`

*c*) Calcule si dos tablas tienen la misma secuencia de consenso.
Implemente para ello la función `MismoConsenso`

*d*) Calcule la *secuencia inconexa* de una tabla.
Implemente la función `SecuenciaInconexa` que calculará y devolverá la secuencia que más se "aleja" de la secuencia de consenso. Para ello utilizaremos la *distancia de Hamming* que suma el valor 1 si los valores en la misma posición de dos secuencias son distintos, y 0 si son iguales. En caso de igual distancia, queda a elección del programador la secuencia de ADN devuelta. Por ejemplo, la distancia de Hamming entre `T T G C A` y `T T T A A` sería 2.
Implemente el cálculo de la distancia de Hamming con una función.

35. *(Adaptación del examen de la convocatoria ordinaria de FP - Enero 2019)*

Se pretende establecer un nuevo orden para comparar valores enteros positivos. Un valor entero $a$ será mayor que otro $b$ si el número de dígitos nueve es mayor en $a$ que en $b$. Si el número de nueves es igual en los dos, el mayor será el que tiene más ochos. Si hay empate también con este dígito, se considera el siete. Así hasta llegar al cero, si fuese necesario. Si la frecuencia de todos los dígitos es igual en ambos valores, se les considera iguales, bajo este orden.

Deberá escribir una función que reciba dos enteros positivos, $a$ y $b$, y devuelva `true` si $a$ es menor estricto que $b$, atendiendo al orden especificado y `false` en caso contrario:

```
bool MenorNuevoOrdenEnteros (int a, int b);
```

Escribir un programa que inicialice aleatoriamente un array de datos `int` (capacidad = 100 casillas) y lo ordene con los métodos de *selección*, *inserción* e *intercambio* (empleando funciones) de acuerdo al criterio de orden enunciado antes.

Por ejemplo, si el array contuviera 5 valores aleatorios entre 0 y 999 (500, 244, 900, 99, 550) tras la ordenación de acuerdo a este criterio quedaría así: (244, 500, 550, 900, 99).

La función que realiza la ordenación por selección, p.e., podría tener el siguiente prototipo:

```
void OrdenaSeleccionNuevoOrdenEnteros (int *v, int tamanio);
```

Ordena por selección un vector de datos `int` (la parte comprendida desde el elemento cuya dirección es `v` que tiene `tamanio` elementos).

El programa recibe valores de la línea de órdenes:

- Si el programa se ejecuta sin argumentos se rellenarán todas las casillas disponibles con números aleatorios entre 0 y 999 (ambos incluidos).
- Si se ejecuta con un argumento se rellenarán tantas casillas como se indique en el argumento dado, con números aleatorios entre 0 y 999 (ambos incluidos).
- Si se ejecuta con dos argumentos se rellenarán tantas casillas como se indique en el primer argumento con números aleatorios entre 0 y el valor dado en el segundo argumento (ambos incluidos).
- Finalmente, si se ejecuta con tres argumentos, se rellenarán tantas casillas como se indique en el primer argumento con números aleatorios entre los valores dados por el segundo y tercer argumento (ambos incluidos).

36. Este ejercicio amplía la funcionalidad del ejercicio 35.

    Podrían reescribirse las funciones que realizan las tareas de ordenación para que recibieran un argumento adicional: la función que se empleará para comparar las parejas de elementos del vector. En realidad lo que se recibirá es la *dirección de memoria* de dicha función, por lo que el argumento formal será un *puntero a funciones*.

    De esta manera, puede considerar las siguientes funciones para la comparación además de la ya mencionada `MenorNuevoOrdenEnteros`:

    ```
    bool MenorClasico (int a, int b)
    ```

    Devuelve `true` si $a < b$ en el sentido matemático *clásico*.

    ```
    bool MenorPrimeraCifra (int a, int b)
    ```

    Devuelve `true` si el primer dígito de `a` es menor que el primero de `b`.

    Cualquiera de las tres funciones podría usarse como parámetro real en la llamada a las funciones que realizan la ordenación el array.

    Reutilice el programa anterior y ordene el array usando los tres métodos de ordenación con los tres criterios de ordenación (en total: 9 ordenaciones).


UNIVERSIDAD
DE GRANADA

# METODOLOGÍA DE LA PROGRAMACIÓN

## GUIONES DETALLADOS


**Grado en Ingeniería Informática**
**GRUPO B**

**Francisco José Cortijo Bon**
cb@decsai.ugr.es

**2021/2022**

Departamento de
Ciencias de la Computación
e Inteligencia Artificial

# Guión de la práctica 1

# El modelo de compilación en C++. El programa g++

## Introducción

En la figura 5 mostramos el esquema básico del proceso de compilación de programas y creación de bibliotecas en C++. En este gráfico, indicamos mediante un *rectángulo con esquinas redondeadas* los diferentes programas involucrados en estas tareas, mientras que los *cilindros* indican los tipos de ficheros (con su extensión habitual) que intervienen.

Este esquema puede servirnos para enumerar las tareas de programación habituales. La tarea más común es la generación de un programa ejecutable. Como su nombre indica, es un fichero que contiene código directamente ejecutable. Éste puede construirse de diversas formas:

1. A partir de un fichero con código fuente.
2. Enlazando ficheros con código objeto.
3. Enlazando el fichero con código objeto con una(s) biblioteca(s).

Las dos últimas requieren que previamente se hayan construido los ficheros objeto (opción 2) y los ficheros de biblioteca (opción 3). Como se puede comprobar en el esquema anterior, la creación de éstos tambien está contemplada en el esquema.

Así, es posible generar únicamente ficheros objeto para:

1. Enlazarlos con otros y generar un ejecutable.

   Exige que uno de los módulos objeto que se van a enlazar contenga la función `main()`.

   Esta forma de construir ejecutables es muy común y usualmente los módulos objeto se borran una vez se han usado para construir el ejecutable, ya que no tiene interés su permanencia.

2. Incorporarlos a una biblioteca.

   Una biblioteca, en la terminología de C++, será es una *colección de módulos objeto*. Entre ellos existirá alguna *relación*, que debe entenderse en un sentido amplio: si dos

módulos objeto están en la misma biblioteca, contendrán funciones que trabajen sobre un mismo *tema* (por ejemplo, funciones de procesamiento de cadenas de caracteres).

Si el objetivo final es la creación de un ejecutable, en última instancia uno o varios módulos objeto de una biblioteca se enlazarán con un módulo objeto que contenga la función `main()`.

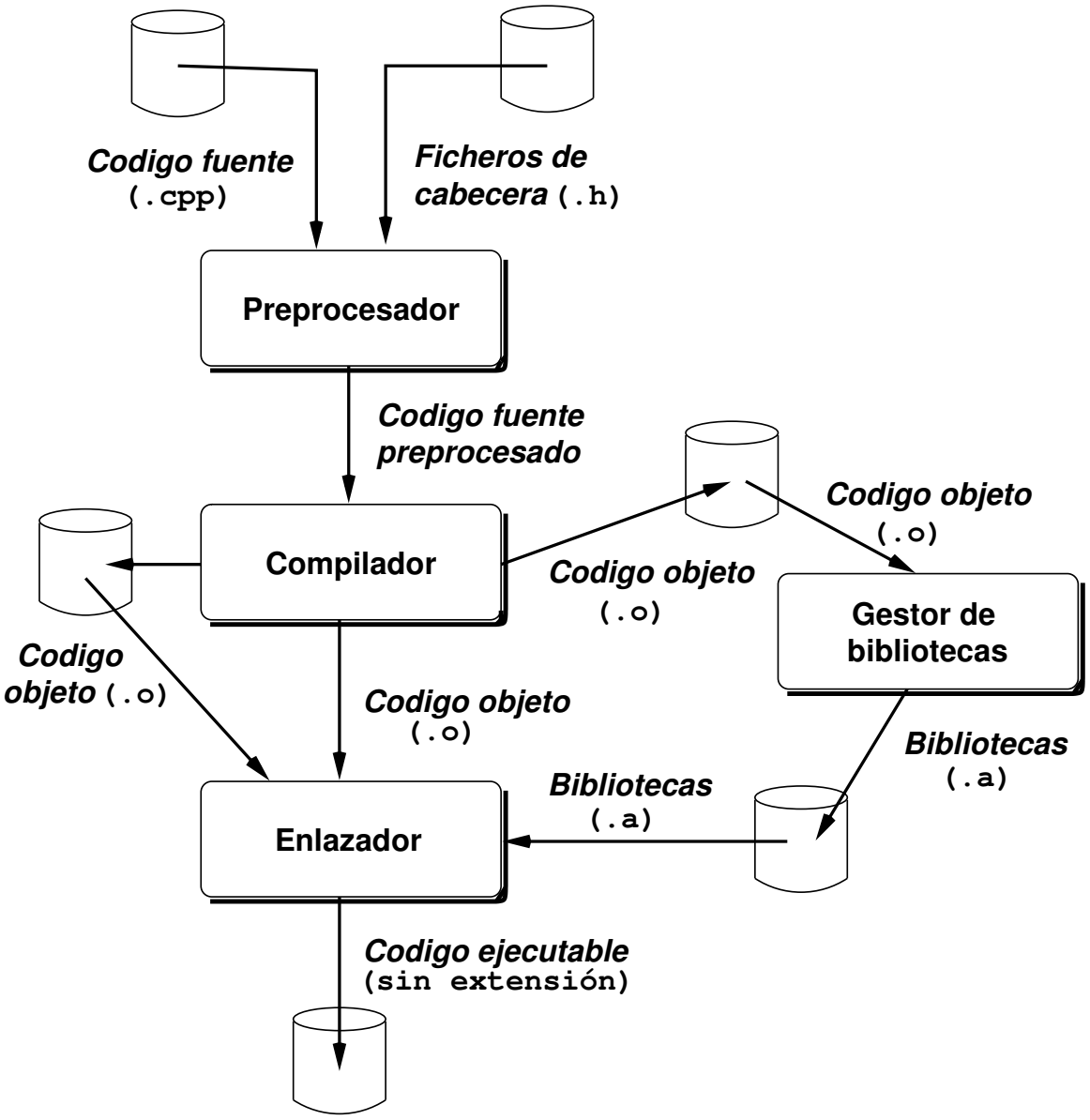



Figura 5: El proceso de compilación (generación de programas ejecutables) en C++

Todos estos puntos se discutirán con mucho más detalle posteriormente. Ahora, introducimos de forma muy general los conceptos y técnicas más importantes del proceso de compilación en C++.

**Ejercicio**

El orden es fundamental para el desarrollo y mantenimiento de programas. Y la premisa más elemental del orden es "un sitio para cada cosa y cada cosa en su sitio".
Hemos visto que en el proceso de desarrollo de software intervienen distintos tipos de ficheros. Cada tipo se guardará en un directorio específico:

- `src`: contendrá los ficheros fuente de C++ (`.cpp`)
- `include`: contendrá los ficheros de cabecera (`.h`)
- `obj`: contendrá los ficheros objeto (`.o`)
- `lib`: contendrá los ficheros de biblioteca (`.a`)
- `bin`: contendrá los ficheros ejecutables. Éstos no tienen asociada ninguna *extensión* predeterminada, sino que la capacidad de ejecución es una propiedad del fichero.

En este ejercicio se trata de **crear una estructura de directorios** de manera que:

1. todos los directorios enumerados anteriormente sean *hermanos*
2. "cuelguen" de un directorio llamado `MP`, y
3. el directorio `MP` cuelgue de vuestro directorio personal (`~`)

## El preprocesador

El preprocesador (del inglés, *preprocessor*) es una herramienta que *filtra* el código fuente antes de ser compilado. El preprocesador acepta como entrada **código fuente** (.cpp) y se encarga de:

1. Eliminar los comentarios.
2. Interpretar y procesar las directivas de preprocesamiento. El preprocesador proporciona un conjunto de directivas que resultan una herramienta sumamente útil al programador. Todas las directivas comienzan *siempre* por el símbolo #.

Dos de las directivas más comúnmente empleadas en C++ son `#include` y `#define`. En la sección  tratamos con más profundidad estas directivas y algunas otras más complejas. En cualquier caso, retomando el esquema mostrado en la figura 5 destacaremos que el preprocesador **no** genera un fichero de salida (en el sentido de que no se guarda el código fuente preprocesado). El código resultante se pasa directamente al compilador. Así, aunque formalmente puden distinguirse las fases de preprocesado y compilación, en la práctica el preprocesado se considera como la primera fase de la compilación. Gráficamente, en la figura 6 mostramos el esquema detallado de lo que se conoce comúnmente por compilación.

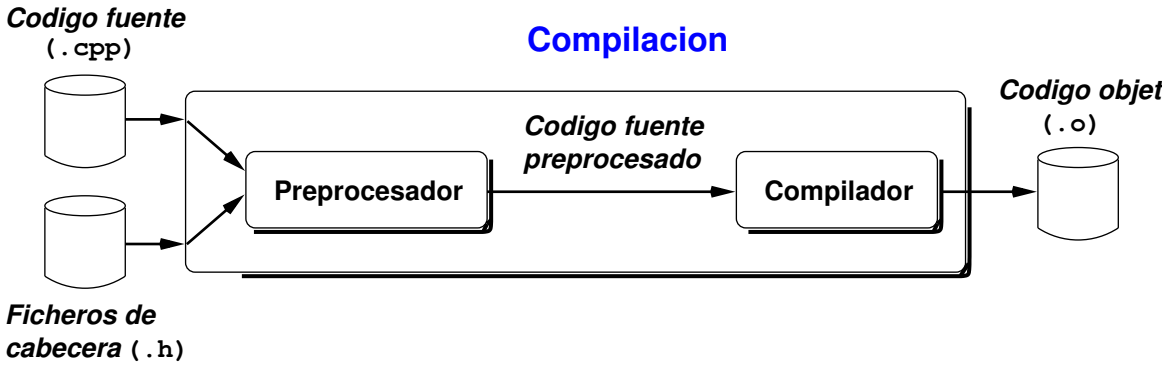



Figura 6: Fase de compilación

## El compilador

El compilador (del inglés, *compiler*) analiza la sintaxis y la semántica del código fuente preprocesado y lo traduce a un **código objeto** que se almacena en un archivo o módulo objeto (.o).

En el proceso de compilación se realiza la traducción del código fuente a código objeto pero no se resuelven las posibles referencias a elementos externos al archivo. Las referencias externas se refieren a variables y principalmente a *funciones* que, aunque se utilizan en el archivo -y por tanto deben estar declaradas en él- no se encuentran definidas en éste, sino en otro archivo distinto. La declaración servirá al compilador para comprobar que las referencias externas son sintácticamente correctas.

## El enlazador

El enlazador (del inglés, *linker*) resuelve las referencias a elementos externos y genera un fichero ejecutable.

Completa los "huecos´´ de un fichero objeto (las referencias pendientes) cuyo código *compilado* se encuentra en otros ficheros con código objeto, que pueden ser otros ficheros objeto independientes o formar parte de alguna **biblioteca** (del inglés, *library*).

## Bibliotecas. El gestor de bibliotecas

C++ es un lenguaje muy reducido. Muchas de las posibilidades incorporadas en forma de funciones en otros lenguajes, no se incluyen en el repertorio de instrucciones de C++. Por ejemplo, el lenguaje no incluye ninguna facilidad de entrada/salida, manipulación de cadenas de caracteres, funciones matemáticas, etc. Esto no significa que C++ sea un lenguaje pobre. Todas estas funciones se incorporan a través de un amplio conjunto de bibliotecas que *no* forman parte, hablando propiamente, del lenguaje de programación.

No obstante, y afortunadamente, algunas bibliotecas *se enlazan automáticamente* al generar un programa ejecutable, lo que induce al error de pensar que las funciones presentes en esas bibliotecas son propias del lenguaje C++. Otra cuestión es que se ha definido y estandarizado la llamada **biblioteca estándar de C++** (en realidad, bibliotecas) de forma que cualquier compilador que quiera tener el "marchamo" de *compatible con el estándard C++* debe asegurar que las funciones proporcionadas en esas bibliotecas se comportan de forma similar a como especifica el comité ISO/IEC para la estandarización de C++ (http://www.open-std.org/jtc1/sc22/wg21/). A efectos prácticos, las funciones de la biblioteca estándar pueden considerarse parte del lenguaje C++.

En cualquier caso el conjunto de bibliotecas disponible y las funciones incluidas en ellas pueden variar de un compilador a otro y el programador responsable deberá asegurarse que cuando usa una función, ésta forma parte de la biblioteca estándar: *este es el procedimiento más seguro para construir programas transportables entre diferentes plataformas y compiladores*.

Cualquier programador puede desarrollar sus propias bibliotecas de funciones y enriquecer de esta manera el lenguaje. En la figura 5 se muestra el proceso de creación y uso de bibliotecas propias. En esta figura se ilustra que una biblioteca es, en realidad, una "objetoteca", si se nos permite el término. De esta forma nos referimos a una biblioteca como a una *colección de módulos objeto*. Estos módulos objeto contendrán el código objeto correspondiente a variables, constantes y funciones que pueden usarse por otros módulos si se enlazan de forma adecuada.

# g++ : el compilador de GNU para C++

## Un poco de historia

Fuente: wikipedia (http://es.wikipedia.org/wiki/GNU)

El **proyecto GNU** fue iniciado por Richard Stallman con el objetivo de crear un sistema operativo completamente libre: el sistema GNU.

El 27 de septiembre de 1983 se anunció públicamente el proyecto por primera vez en el grupo de noticias net.unix-wizards. Al anuncio original, siguieron otros ensayos escritos por Richard Stallman como el "Manifiesto GNU", que establecieron sus motivaciones para realizar el proyecto GNU, entre las que destaca "volver al espíritu de cooperación que prevaleció en los tiempos iniciales de la comunidad de usuarios de computadoras".

GNU es un acrónimo recursivo que significa ***GNU No es Unix*** (**G**NU is **N**ot **U**nix). Puesto que en inglés "gnu" (en español "ñu") se pronuncia igual que "new", Richard Stallman recomienda pronunciarlo "guh-noo". En español, se recomienda pronunciarlo ñu como el antílope africano o fonéticamente; por ello, el término mayoritariamente se deletrea (G-N-U) para su mejor comprensión. En sus charlas Richard Stallman finalmente dice siempre «Se puede pronunciar de cualquier forma, la única pronunciación errónea es decirle 'linux'».

UNIX es un Sistema Operativo *no libre* muy popular, porque está basado en una arquitectura que ha demostrado ser técnicamente estable. El sistema GNU fue diseñado para ser totalmente compatible con UNIX. El hecho de ser compatible con la arquitectura de UNIX implica que GNU esté compuesto de pequeñas piezas individuales de software, muchas de las cuales ya estaban disponibles, como el sistema de edición de textos TeX y el sistema gráfico X Window, que pudieron ser adaptados y reutilizados; otros en cambio tuvieron que ser reescritos.

Para asegurar que el software GNU permaneciera libre para que todos los usuarios pudieran "ejecutarlo, copiarlo, modificarlo y distribuirlo", el proyecto debía ser liberado bajo una licencia diseñada para garantizar esos derechos al tiempo que evitase restricciones posteriores de los mismos. La idea se conoce en Inglés como copyleft -'copia permitida'- (en clara oposición a copyright -'derecho de copia'-), y está contenida en la *Licencia General Pública de GNU* (**GPL**).

En 1985, Stallman creó la *Free Software Foundation* (**FSF** o Fundación para el Software Libre) para proveer soportes logísticos, legales y financieros al proyecto GNU. La FSF también contrató programadores para contribuir a GNU, aunque una porción sustancial del desarrollo fue (y continúa siendo) producida por voluntarios. A medida que GNU ganaba renombre, negocios interesados comenzaron a contribuir al desarrollo o comercialización de productos GNU y el correspondiente soporte técnico. En 1990, el sistema GNU ya tenía un editor de texto llamado Emacs, un exitoso compilador (**GCC**), y la mayor parte de las bibliotecas y utilidades que componen un sistema operativo UNIX típico. Pero faltaba un

componente clave llamado núcleo (*kernel* en inglés).

En el manifiesto GNU, Stallman mencionó que “un núcleo inicial existe, pero se necesitan muchos otros programas para emular Unix”. Él se refería a TRIX, que es un núcleo de llamadas remotas a procedimientos, desarrollado por el MIT y cuyos autores decidieron que fuera libremente distribuido; TRIX era totalmente compatible con UNIX versión 7. En diciembre de 1986 ya se había trabajado para modificar este núcleo. Sin embargo, los programadores decidieron que no era inicialmente utilizable, debido a que solamente funcionaba en “algunos equipos sumamente complicados y caros” razón por la cual debería ser portado a otras arquitecturas antes de que se pudiera utilizar. Finalmente, en 1988, se decidió utilizar como base el núcleo Mach desarrollado en la CMU. Inicialmente, el núcleo recibió el nombre de Alix (así se llamaba una novia de Stallman), pero por decisión del programador Michael Bushnell fue renombrado a Hurd. Desafortunadamente, debido a razones técnicas y conflictos personales entre los programadores originales, el desarrollo de Hurd acabó estancándose.

En 1991, **Linus Torvalds** empezó a escribir el núcleo Linux y decidió distribuirlo bajo la licencia GPL. Rápidamente, múltiples programadores se unieron a Linus en el desarrollo, colaborando a través de Internet y consiguiendo paulatinamente que Linux llegase a ser un núcleo compatible con UNIX. En 1992, el núcleo Linux fue combinado con el sistema GNU, resultando en un sistema operativo libre y completamente funcional. El Sistema Operativo formado por esta combinación es usualmente conocido como “**GNU**/**Linux**” o como una “distribución Linux” y existen diversas variantes.

También es frecuente hallar componentes de GNU instalados en un sistema UNIX no libre, en lugar de los programas originales para UNIX. Esto se debe a que muchos de los programas escritos por el proyecto GNU han demostrado ser de mayor calidad que sus versiones equivalentes de UNIX. A menudo, estos componentes se conocen colectivamente como “herramientas GNU”. Muchos de los programas GNU han sido también transportados a otros sistemas operativos como Microsoft Windows y Mac OS X.

**GNU Compiler Collection** (colección de compiladores GNU) es un conjunto de compiladores creados por el proyecto GNU. GCC es software libre y lo distribuye la FSF bajo la licencia GPL.

Estos compiladores se consideran estándar para los sistemas operativos derivados de UNIX, de código abierto o también de propietarios, como Mac OS X. GCC requiere el conjunto de aplicaciones conocido como binutils para realizar tareas como identificar archivos objeto u obtener su tamaño para copiarlos, traducirlos o crear listas, enlazarlos, o quitarles símbolos innecesarios.

Originalmente GCC significaba *GNU C Compiler* (compilador GNU para C), porque sólo compilaba el lenguaje C. Posteriormente se extendió para compilar C++, Fortran, Ada y otros.

**g++** es el *alias* tradicional de GNU C++, un conjunto gratuito de compiladores de C++. Forma parte del GCC. En sistemas operativos GNU, gcc es el comando usado para ejecutar el compilador de C, mientras que g++ ejecuta el compilador de C++.

Otros programas del Proyecto GNU relacionados con nuestra materia son:

- **GNU ld**: la implementación de GNU del enlazador de Unix `ld`. Su nombre se forma a partir de la palabra ***l**oa**d**er*

  Un enlazador es un programa que toma los ficheros de código objeto generado en los primeros pasos del proceso de compilación, la información de todos los recursos necesarios (biblioteca), quita aquellos recursos que no necesita, y enlaza el código objeto con su(s) biblioteca(s) y produce un fichero ejecutable. En el caso de los programas enlazados dinámicamente, el enlace entre el programa ejecutable y las bibliotecas se realiza en tiempo de carga o ejecución del programa.

- **GNU ar**: la implementación de GNU del archivador de Unix `ar`. Su nombre proviene de la palabra ***ar**chiver*

  Es una utilidad que mantiene grupos de ficheros como un único fichero (básicamente, un empaquetador-desempaquetador). Generalmente, se usa `ar` para crear y actualizar ficheros de *biblioteca* que utiliza el enlazador; sin embargo, se puede usar para crear archivos con cualquier otro propósito.

## Sintaxis

La ejecución de `g++` sigue el siguiente patrón sintáctico:

`g++` [ `-`*opción* [*argumento(s)_opción*]] *nombre_fichero*

donde:

- Cada **opción** va precedida por el signo `-` Algunas opciones **no** están acompañadas de argumentos (por ejemplo, `-c` ó `-g`) de ahí que *argumento(s)_opción* sea opcional.

  En el caso de ir acompañadas de algún argumento, se especifican a continuación de la opción. Por ejemplo, la opción `-o saludo.o` indica que el nombre del fichero resultado es `saludo.o`, la opción `-I /usr/include` indica que se busquen los ficheros de cabecera en el directorio `/usr/include`, etc. Las opciones mas importantes se describen con detalle en la sección .

- *nombre_fichero* indica el fichero a procesar. Siempre debe especificarse.

  El compilador interpreta por defecto que un fichero contiene código en un determinado formato (C, C++, fichero de cabecera, etc.) dependiendo de la extensión del fichero. Las extensiones más importantes que interpreta el compilador son las siguientes: `.c` (código fuente C), `.h` (fichero de cabecera: este tipo de ficheros no se compilan ni se enlazan directamente, sino a través de su inclusión en otros ficheros fuente), `.cpp` (código fuente C++).

  Por defecto, el compilador realizará distintas tareas dependiendo del tipo de fichero que se le especifique. Como es natural, existen opciones que especifican al compilador que realice sólo aquellas etapas del proceso de compilación que deseemos.

## Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

`-ansi` considera únicamente código fuente escrito en C/C++ estándar y rechaza cualquier extensión que pudiese tener conflictos con ese estándar.

`-c` realizar solamente el preprocesamiento y la compilación de los ficheros fuentes. No se lleva a cabo la etapa de enlazado.

**Observe que estas acciones son las que corresponden a lo que se ha definido como compilación.** El hecho de tener que modificar el comportamiento de `g++` con esta opción para que solo compile es indicativo de que el comportamiento por defecto de `g++` no es -solo- compilar sino realizar el trabajo completo: ***compilar y enlazar*** para crear un ejecutable.

El programa enlazador proporcionado por GNU es `ld`. Sin embargo, no es usual llamar a este programa explícitamente sino que éste es invocado convenientemente por `g++`. Así, vemos que `g++` es más que un compilador (formalmente hablando) ya que al llamar a `g++` se proprocesa el código fuente, se compila, e incluso se enlaza.

**Ejercicio**

1. Crear el fichero `saludo.cpp` que imprima en la pantalla un mensaje de bienvenida (el famoso `¡¡hola, mundo!!`) y guardarlo en el directorio `src`.
2. Ejecutar la siguiente orden y observar e interpretar el resultado

   ```
   g++ src/saludo.cpp
   ```
3. Ejecutar la siguiente orden y observar e interpretar el resultado

   ```
   g++ -c src/saludo.cpp
   ```

`-o` *fichero_salida* especifica el nombre del fichero de salida, resultado de la tarea solicitada al compilador.

Si no se especifica la opción `-o`, el compilador generará un fichero y le asignará un nombre por defecto (dependiendo del tipo de fichero que genere). Lo normal es que queramos asignarle un nombre determinado por nosotros, por lo que esta opción siempre se empleará.

**Ejercicio**

Ejecutar la siguiente orden y observar e interpretar el resultado. El diagrama de dependencias se muestra en la figura 7.

```
g++ -o bin/saludo src/saludo.cpp
```


Figura 7: Diagrama de dependencias para `saludo`

**Ejercicio**

Ejecutar las siguientes órdenes y observar e interpretar el resultado. El diagrama de dependencias se muestra en la figura 8.

1. `g++ -c -o obj/saludo.o src/saludo.cpp`
2. `g++ -o bin/saludo_en_dos_pasos obj/saludo.o`

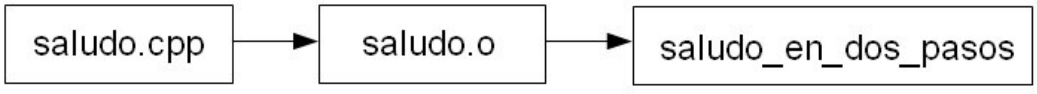



Figura 8: Diagrama de dependencias para `saludo_en_dos_pasos`

`-W` *all* Muestra todos los mensajes de advertencia del compilador.

`-g` Incluye en el ejecutable la información necesaria para poder trazarlo empleando un depurador.

`-v` Muestra con detalle en las órdenes ejecutadas por `g++`.

**Ejercicio**

1. Ejecutar la siguiente orden y observar e interpretar el resultado

   ```
   g++ -v -o bin/saludo src/saludo.cpp
   ```

2. Ejecutar la siguiente orden y observar e interpretar el resultado

   ```
   g++ -Wall -v -o bin/saludo src/saludo.cpp
   ```

3. Ejecutar la siguiente orden y observar e interpretar el resultado, comparando el tamaño del fichero obtenido con el de `saludo`

   ```
   g++ -g -o bin/saludo_con_g src/saludo.cpp
   ```

`-I` *path* especifica el directorio donde se encuentran los ficheros a incluir por la directiva `#include`. Se puede utilizar esta opción varias veces para especificar distintos directorios.

**Ejercicio**

Ejecutar la siguiente orden:

```
g++ -v -c -I/usr/local/include -o obj/saludo.o
src/saludo.cpp
```

Observad cómo añade el directorio `/usr/local/include` a la lista de directorios en los que buscar los ficheros de cabecera. Si el fichero `saludo.cpp` incluyera un fichero de cabecera que se encuentra en el directorio `/usr/local/include`, la orden anterior hace que `g++` pueda encontrarlo para el preprocesador. Pueden incluirse cuantos directorios se deseen, por ejemplo:

```
g++ -v -c -I/usr/local/include -I./include -o obj/saludo.o
     src/saludo.cpp
```

`-L` *path* indica al enlazador el directorio donde se encuentran los ficheros de biblioteca. Como ocurre con la opción `-I`, se puede utilizar la opción `-L` varias veces para especificar distintos directorios de biblioteca.

1. Los ficheros de biblioteca que deben usarse se proporcionan a `g++` con la opción `-l`*fichero*.
2. Esta opción hace que el enlazador busque en los directorios de bibliotecas (entre los que están los especificados con `-L`) un fichero de biblioteca llamado `lib`*fichero*`.a` y lo usa para enlazarlo.

**Ejercicio**

Ejecutar la siguiente orden:

```
g++ -v -o bin/saludo -L/usr/local/lib obj/saludo.o -lutils
```

Observe que se llama al enlazador para que enlace el objeto `saludo.o` con la biblioteca `libutils.a` y obtenga el ejecutable `saludo`. Concretamente se busca el fichero de biblioteca `libutils.a` en el directorio `/usr/local/lib`.

`-D` *nombre*[=*cadena*] define una constante simbólica llamada *nombre* con el valor *cadena*. Si no se especifica el valor, *nombre* simplemente queda definida. *cadena* no puede contener blancos ni tabuladores. Equivale a una linea `#define` al principio del fichero fuente, salvo que si se usa `-D`, el ámbito de la macrodefinición incluye todos los ficheros especificados en la llamada al compilador.

`-O` Optimiza el tamaño y la velocidad del código compilado. Existen varios niveles de optimización cada uno de los cuales proporciona un código menor y más rápido a costa de emplear más tiempo de compilación y memoria principal. Ordenadas de menor a mayor intensidad son: `-O`, `-O1`, `-O2` y `-O3`. Existe una opción adicional `-Os` orientada a optimizar exclusivamente el tamaño del código compilado (esta opción no lleva a cabo ninguna optimización de velocidad que implique un aumento de código).

# Resumen: un ejemplo completo

Los ficheros necesarios para realizar este ejercicio puede encontralos en `PRADO`. El fichero `demo.cpp` se copiará en la carpeta `src`, el fichero `utils.h` en la carpeta `include` y el fichero `libutils.a` en la carpeta `lib`.

La biblioteca *utils* (fichero de biblioteca `libutils.a`) tiene asociada el fichero de cabecera `utils.h`. Enlazando convenientemente esa biblioteca pueden emplearse las funciones cuyas cabeceras (*prototipos*) encontrará en `utils.h`.

En `demo.cpp` puede ver que éste contiene únicamente la función `main()`, donde se llama a las funciones `DivisionEntera()` y `RestoDivision()`. En `demo.cpp` se incluye el fichero de cabecera `utils.h` con la línea:

```
#include "utils.h"
```

La inclusión hace que el compilador pueda conocer la cabecera de las funciones mencionadas anteriormente (en realidad sólo está interesado en conocer que se trata de funciones que reciben dos `int` y devuelven un valor `int`). Así, puede dar por válida las llamadas aunque no es capaz de generar código ejecutable ya que desconoce el código de esas funciones. Genera, por tanto, código objeto susceptible se ser ejecutable (al contener la función `main()`).

No obstante, para poder generar el fichero objeto asociado a `demo.cpp`, al incluir éste un fichero de cabecera particular, deberemos indicar al compilador la carpeta en la que debe buscarlo usando la opción `-I`.

---

Para generar el objeto `demo.o` (en la carpeta `./obj`) a partir del fuente `demo.cpp` (en la carpeta `./src`) indicando que el fichero de cabecera está en la carpeta `./include` escribiremos:

```
g++ -c -o ./obj/demo.o ./src/demo.cpp -I./include
```

---

Para generar el fichero ejecutable es preciso “completar” o “rellenar” los huecos que tiene `demo.o` con el código de las funciones, que se encuentra en la biblioteca *utils*: esta es la tarea del enlazador.

Como la biblioteca que se usa es una biblioteca particular, deberemos indicar al enlazador la carpeta en la que debe buscarla usando la opción `-L`. Además, cuando se usa una biblioteca no se escribe el nombre del fichero de biblioteca (`libutils.a` en este caso) sino que se emplea la opción `-l` y se emplea el nombre corto (no se escribe ni `lib` ni `.a`).

Para generar el ejecutable `demo` (en la carpeta `./bin`) a partir del objeto `demo.o` (en la carpeta `./obj`) y la biblioteca *utils* (fichero `libutils.a`) indicando que la biblioteca está en la carpeta `./lib` escribiremos:

```
g++ -o ./bin/demo ./obj/demo.o -lutils -L./lib
```

**Nota de compatibilidad:** Si obtuviera algún tipo de error durante el enlace debido a un problema de compatibilidad de la biblioteca deberá generar la biblioteca a partir del fichero fuente que contiene el código de las funciones y del fichero de cabecera (se estudiará con detalle en la Práctica 3). Primero generará el fichero objeto y después creará la biblioteca a partir de ese fichero objeto.

```
g++ -c -o ./obj/utils.o ./src/utils.cpp -I./include
ar -rvs ./lib/libutils.a ./obj/utils.o
```

# El preprocesador de C++. Directivas de preprocesamiento

Recordemos que el preprocesamiento es la primera etapa del proceso de compilación de programas C++. El preprocesador es una herramienta que *filtra* el código fuente antes de ser compilado. Acepta como entrada código fuente y se encarga de:

1. Eliminar los comentarios.
2. Interpretar y procesar las **directivas de preprocesamiento**. El preprocesador proporciona un conjunto de directivas que resultan una herramienta sumamente útil al programador. Todas las directivas comienzan *siempre* por el símbolo `#`.

   - `#include`: Sustituye la línea por el contenido del fichero especificado.

     Por ejemplo, `#include <iostream>` incluye el fichero `iostream.h`, que contiene declaraciones de tipos y funciones de entrada/salida de la biblioteca estándar de C++. La inclusión implica que *todo* el contenido del fichero incluído sustituye a la línea `#include`.

     Los nombres de los ficheros de cabecera heredados de C comienzan por la letra c, y se incluyen usando la misma sintaxis. Por ejemplo: `#include <cstring>`,`#include<cstdlib>`,...
   - `#define`: Define una constante (identificador) simbólico.

     Sustituye las apariciones del identificador por el valor especificado, salvo si el identificador se encuentra dentro de una constante de cadena de caracteres (entre comillas).

     Por ejemplo, `#define MAX_SIZE 100` establece el valor de la constante simbólica `MAX_SIZE` a 100. En el programa se utilizará la constante simbólica y el preprocesador sustituye cada aparación de `MAX_SIZE` por el literal 100 (de tipo `int`).

El uso de las directivas de preprocesamiento proporciona varias ventajas:

1. Facilita el desarrollo del software.
2. Facilita la lectura de los programas.
3. Facilita la modificación del software.
4. Ayuda a hacer el código C++ portable a diferentes arquitecturas.
5. Facilita el ocultamiento de información.

En este apéndice veremos algunas de las directivas más importantes del preprocesador de C++:

`#define`: Creación de constantes simbólicas y macros funcionales.

`#undef`: Eliminación de constantes simbólicas.

`#include`: Inclusión de ficheros.

`#if` (`#else`, `#endif`): Inclusión condicional de código.

## Constantes simbólicas y macros funcionales

### Constantes simbólicas

La directiva `#define` se puede emplear para definir constantes simbólicas de la siguiente forma:

`#define` *identificador* *texto de sustitución*

El preprocesador sustituye todas las apariciones de *identificador* por el *texto de sustitución*. Funciona de la misma manera que la utilidad "Busca y Sustituye" que tienen casi todos los editores de texto. La única excepción son las apariciones dentro de constantes de cadena de caracteres (delimitadas entre comillas), que no resultan afectadas por la directiva `#define`. El ámbito de la definición de una constante simbólica se establece desde el punto en el que se define hasta el final del fichero fuente en el que se encuentra.

Veamos algunos ejemplos sencillos.

```
#define TAMMAX 256
```

hace que sustituya todas las apariciones del identificador `TAMMAX` por la constante numérica (entera) 256.

```
#define UTIL_VEC
```

simplemente define la constante simbólica `UTIL_VEC`, aunque sin asignarle ningún valor de sustitución: se puede interpretar como una "bandera" (existe/no existe). Se suele emplear para la inclusión condicional de código (ver página 18 de este apéndice).

```
#define begin {
#define end }
```

para aquellos que odian poner llaves en el comienzo y final de un bloque y prefieren escribir `begin`..`end`.

### Macros funcionales (con argumentos)

Podemos también definir macros funcionales (con argumentos). Son como "pequeñas funciones" pero con algunas diferencias:

1. Puesto que su implementación se lleva a cabo a través de una sustitución de texto, su efecto en el programa no es el de las funciones tradicionales.
2. En general, las macros recursivas no funcionan.
3. En las macros funcionales el tipo de los argumentos es indiferente. Suponen una gran ventaja cuando queremos hacer el mismo tratamiento a diferentes tipos de datos.

Las macros funcionales pueden ser problemáticas para los programadores descuidados. Hemos de recordar que lo único que hacen es realizar una sustitución de texto. Por ejemplo, si definimos la siguiente macro funcional:

```
#define DOBLE(x) x+x
```

y tenemos la sentencia

```
a = DOBLE(b) * c;
```

su expansión será la siguiente: `a = b + b * c;` Ahora bien, puesto que el operador `*` tiene mayor precedencia que +, tenemos que la anterior expansión se interpreta, realmente, como `a = b + (b * c);` lo que probablemente no coincide con nuestras intenciones iniciales. La forma de "reforzar" la definición de `DOBLE()` es la siguiente

```
#define DOBLE(x) ((x)+(x))
```

con lo que garantizamos la evaluación de los operandos antes de aplicarle la operación de suma. En este caso, la sentencia anterior (`a = DOBLE(b) * c`) se expande a

```
a = ((b) + (b)) * c;
```

con lo que se avalúa la suma antes del producto.
Veamos ahora algunos ejemplos adicionales.

```
#define MAX(A,B) ((A)>(B)?(A):(B))
```

La ventaja de esta definición de la "función" máximo es que podemos emplearla para cualquier tipo para el que esté definido un orden (si está definido el operador >)

```
#define DIFABS(A,B) ((A)>(B)?((A)-(B)):((B)-(A)))
```

Calcula la diferencia absoluta entre dos operandos.

### Eliminación de constantes simbólicas

La directiva: `#undef` *identificador* anula una definición previa del *identificador* especificado. Es preciso anular la definición de un identificador para asignarle un nuevo valor con un nuevo `#define`.

## Inclusión de ficheros

La directiva `#include` hace que se incluya el contenido del fichero especificado en la posición en la que se encuentra la directiva. Se emplean casi siempre para realizar la inclusión de ficheros de cabecera de otros módulos y/o bibliotecas. El nombre del fichero puede especificarse de dos formas:

- `#include <`*fichero*`>`
- `#include "`*fichero*`"`

La única diferencia es que <*fichero*> indica que el fichero se encuentra en alguno de los directorios de ficheros de cabecera del sistema o entre los especificados como directorios de ficheros de cabecera (opción `-I` del compilador: ver página 9), mientras que "*fichero*" indica que se encuentra en el directorio donde se esta realizando la compilación. Así,

```
#include <iostream>
```

incluye el contenido del fichero de cabecera que contiene los prototipos de las funciones de entrada/salida de la biblioteca estándar de C++. Busca el fichero entre los directorios de ficheros de cabecera del sistema.

### Inclusión condicional de código

La directiva `#if` evalúa una expresión constante entera. Se emplea para incluir código de forma selectiva, dependiendo del valor de condiciones evaluadas en tiempo de compilación (en concreto, durante el preprocesamiento). Veamos algunos ejemplos.

```
#if ENTERO == LARGO
   typedef long mitipo;
#else
   typedef int mitipo;
#endif
```

Si la constante simbólica `ENTERO` tiene el valor `LARGO` se crea un alias para el tipo `long` llamado `mitipo`. En otro caso, `mitipo` es un alias para el tipo `int`.

La cláusula `#else` es opcional, aunque siempre hay que terminar con `#endif`. Podemos encadenar una serie de `#if` - `#else` - `#if` empleando la directiva `#elif` (resumen de la secuencia `#else` - `#if`):

```
#if SISTEMA == SYSV
   #define CABECERA "sysv.h"
#elif SISTEMA == LINUX
   #define CABECERA "linux.h"
#elif SISTEMA == MSDOS
   #define CABECERA "dos.h"
#else
   #define CABECERA "generico.h"
#endif

#include CABECERA
```

De esta forma, estamos seguros de que incluiremos el fichero de cabecera apropiado al sistema en el que estemos compilando. Por supuesto, debemos especificar de alguna forma el valor de la constante `SISTEMA` (por ejemplo, usando macros en la llamada al compilador, como indicamos en la página 9).

Podemos emplear el predicado: `defined(`*identificador*`)` para comprobar la existencia del *identificador* especificado. Éste existirá si previamente se ha utilizado en una macrodefinición (siguiendo a la claúsula `#define`). Este predicado se suele usar en su forma resumida (columna derecha):

| | |
|---|---|
| `#if defined(`*identificador*`)` | `#ifdef(`*identificador*`)` |
| `#if !defined(`*identificador*`)` | `#ifndef(`*identificador*`)` |

Su utilización está aconsejada para prevenir la inclusión repetida de un fichero de cabecera en un mismo fichero fuente. Aunque pueda parecer que ésto no ocurre a menudo ya que a nadie se le ocurre escribir, por ejemplo, en el mismo fichero:

```
#include "cabecera1.h"
#include "cabecera1.h"
```

sí puede ocurrir lo siguiente:

```
#include "cabecera1.h"
#include "cabecera2.h"
```

y que `cabecera2.h` incluya, a su vez, a `cabecera1.h` (una inclusión “transitiva”). El resultado es que el contenido de `cabecera1.h` se copia dos veces, dando lugar a errores por definición múltiple. Esto se puede evitar *protegiendo* el contenido del fichero de cabecera de la siguiente forma:

```
#ifndef (HDR)
#define HDR

    Resto del contenido del fichero de cabecera

#endif
```

En este ejemplo, la constante simbólica `HDR` se emplea como *testigo*, para evitar que nuestro fichero de cabecera se incluya varias veces en el programa que lo usa. De esta forma, cuando se incluye la primera vez, la constante `HDR` no está definida, por lo que la evaluación de `#ifndef(HDR)` es cierta, se define y se procesa (incluye) el resto del fichero de cabecera. Cuando se intenta incluir de nuevo, como `HDR` ya está definida la evaluación de `#ifndef (HDR)` es falsa y el preprocesador salta a la línea siguiente al predicado `#endif`. Si no hay nada tras este predicado el resultado es que no incluye nada.

Todos los ficheros de cabecera que acompañan a los ficheros de la biblioteca estándar tienen un prólogo de este estilo.

# Anexo: Introducción al depurador DDD

## Conceptos básicos

El programa `ddd` es, básicamente, una interfaz (*front-end*) separada que se puede utilizar con un depurador en línea de órdenes. En el caso que concierne a este documento, `ddd` será la interfaz de alto nivel del depurador `gdb`.

Para poder utilizar el depurador es necesario compilar los ficheros fuente con la opción `-g`. En otro caso mostrará un mensaje de error. En cualquier caso, el depurador se invoca con la orden

```
ddd fichero-binario
```

## Pantalla principal

La pantalla principal del depurador se muestra en la figura 9.a).

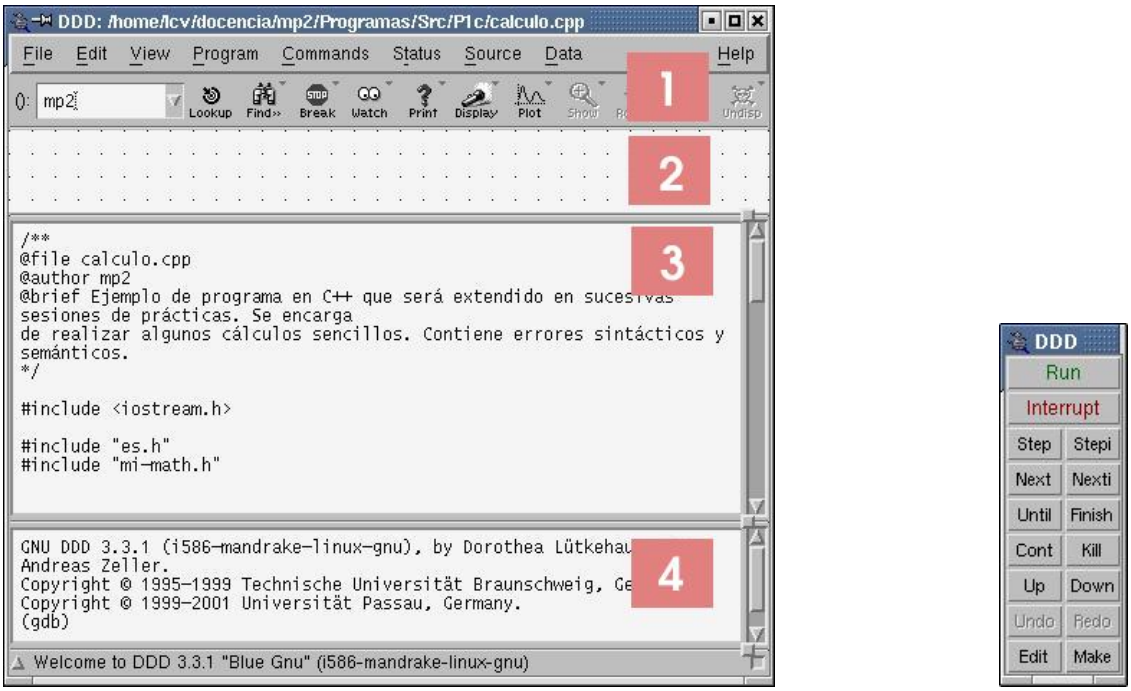
En ella se pueden apreciar las siguientes partes.

1. Zona de menú y barra de herramientas. Con los componentes típicos de cualquier programa.
2. Zona de visualización de datos. En esta parte de la ventana se mostrarán las variables que se hayan elegido y sus valores asociados. Si esta zona no estuviese visible, menú View - Data Window.
3. Zona de visualización de código fuente. Se muestra el código fuente que se está depurando y la línea por la que se está ejecutando el programa. Si esta zona no estuviese visible, menú View - Source Window.
4. Zona de visualización de mensajes de `gdb`. Muestra los mensajes del verdadero depurador, en este caso, `gdb`· Si esta zona no estuviese visible, menú View - Gdb Console.

Sobre la ventala principal aparece una ventana flotante de herramientas que se muestra en la figura 9.b) desde la que se pueden hacer, de forma simplificada, las mayoría de las operaciones de depuración.

## Ejecución de un programa paso a paso

Una vez cargado un programa binario, se puede comenzar la ejecución siguiendo cualquiera de los métodos mostrados en el cuadro 1. Hay que tener en cuenta que esta orden inicia




(a) (b)

Figura 9: Pantalla principal de `ddd`

la ejecución del programa de la misma forma que si se hubiese llamado desde la línea de argumentos, de forma que, de no haber operaciones de entrada/salida desde el teclado, el programa comenzará a ejecutarse sin control directo desde el depurador hasta que termine, momento en el que devuelve el control al depurador mostrando el siguiente mensaje

```
(gdb) Program exited normally
```

En cualquier momento se puede terminar la ejecución de un programa mediante distintas formas, la más rápida es mediante la orden `kill` (ver cuadro 1). También se pueden pasar argumentos a la función `main` desde la ventana que aparece en la figura 10.

Para comenzar a ejecutar un programa bajo control del depurador es conveniente colocar un punto de ruptura[1] en la primera línea ejecutable del código. Una vez colocado este punto de ruptura se puede comenzar la ejecución del programa paso a paso según lo mostrado en el cuadro 1 y teniendo en cuenta que `ddd` señala la línea de código activa con una pequeña flecha verde a la izquierda de la línea  . `ddd` también muestra la salida de la ejecución del

[1] Un punto de ruptura (abreviadamente PR) es una marca en una línea de código ejecutable de forma que su ejecución siempre se interrumpe antes de ejecutar esta línea, pasando el control al depurador. `ddd` visualiza esta marca como una pequeña señal de STOP .

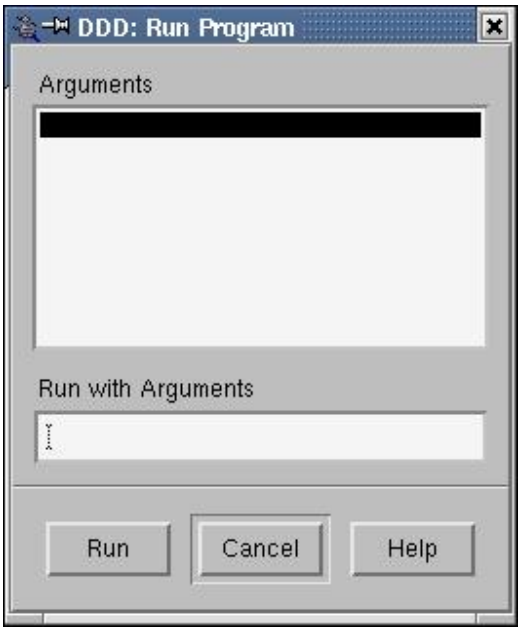



Figura 10: Ventana para pasar argumentos a `main`

programa en una ventana independiente (`DDD: Execution window`). Si esta ventana no estuviese visible, entonces puede mostrarse pulsando en menú Program - Run in execution window.

## Inspección y modificación de datos

`ddd`, como cualquier depurador, permite inspeccionar los valores asociados a cualquier variable y modificar sus valores. Se puede visualizar datos temporalmente, de forma que sólo se visualizan sus valores durante un tiempo limitado, o permanentemente en la ventana de datos (*watch*, de forma que sus valores se visualicen durante toda la ejecución (ver cuadro 1). Es necesario aclarar que sólo se puede visualizar el valor de una variable cuando la línea de ejecución activa se encuentre en un ámbito en el que sea visible esta variable. Asímismo, `ddd` permite modificar, en tiempo de ejecución, los valores asociados a cualquier variable de un programa, bien desde la ventana del código, bien desde la ventana de visualización de datos.

## Inspección de la pila

Durante el proceso de ejecución de un programa se suceden llamadas a módulos que se van almacenando en la pila. `ddd` ofrece la posibilidad de inspeccionar el estado de esta pila

| **Acción** | **Menu** | **Teclas** | **Barra herramientas** | **Otro** |
|---|---|---|---|---|
| Comenzar la ejecución | Program - Run | F2 | Run | |
| Matar el programa | Program - Kill | F4 | Kill | |
| Poner un PR | - | - | Break | Pinchar derecho - Set breakpoint |
| Quitar un PR | - | - | - | Pinchar derecho sobre STOP - Disable Breakpoint |
| Paso a Paso (sí llamadas) | Program - Step | F5 | Step | |
| Paso a Paso (no llamadas) | Program - Next | F6 | Next | |
| Continuar indefinidamente | Program - Continue | F9 | Cont | |
| Continuar hasta el cursor | Program - Until | F7 | Until | Pinchar derecho - Continue Until Here |
| Continuar hasta el final de la función actual | Program - Finish | F8 | Finish | |
| Mostrar temporalmente el valor de una variable | Escribir su nombre en **():** - Botón Print | - | - | Situar ratón sobre cualquier ocurrencia |
| Mostrar permanentemente el valor de una variable (ventana de datos) | Escribir su nombre en **():** - Botón Display | - | - | Pinchar derecho sobre cualquier ocurrencia - Display |
| Borrar una variable de la ventana de datos | - | - | - | Pinchar derecho sobre visualización - Undisplay |
| Cambiar el valor de una variable | Pinchar sobre variable (en ventana de datos o código) - Botón Set | - | - | Pinchar derecho sobre visualización - Set value |

Cuadro 1: Principales acciones del programa `ddd` y las formas más comunes de invocarlas

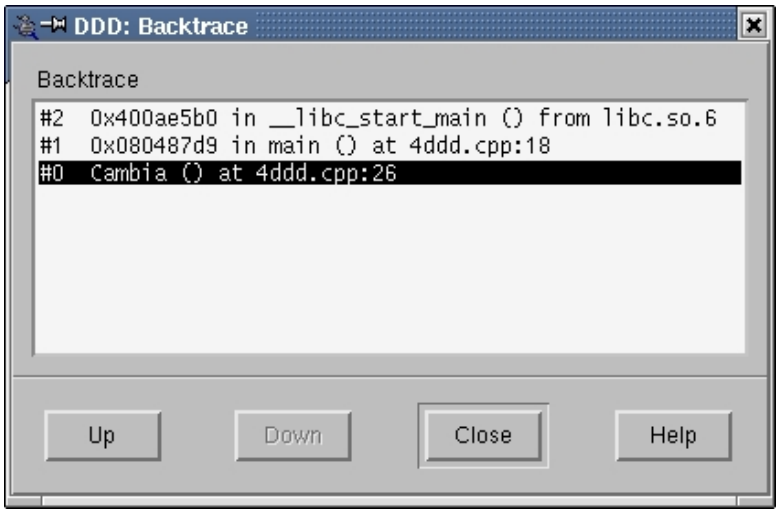



Figura 11: Ventana que muestra el estado de la pilla de llamadas a módulos

y analizar qué llamadas se están resolviendo en un momendo dado de la ejecución de un programa (ver cuadro 1).

## Mantenimiento de sesiones de depuración

Una vez que se cierra el programa `ddd` se pierde toda la información sobre PR, visualización permanente de datos, etc, que se hubiese configurado a lo largo de una sesión de depuración. Para evitar volver a introducir toda esta información, `ddd` permite grabar sesiones de depuración a través del menú principal (opciones de sesiones). Cuando se graba una sesión de depuración se graba exclusivamente la configuración de depuración, en ningún caso se puede volver a restaurar la ejecución de un programa antiguo con sus valores de memoria, etc.

## Reparación del código

Durante una sesión con `ddd` es normal que sea necesario modificar el código para reparar algún error detectado. En este caso es necesario mantener bien actualizada la versión del programa que se encuentra cargada. Para ello lo mejor es interrumpir la ejecución del programa, recompilar los módulos que fuese necesario y recargarlo para continuar la depuración.

# Guión de la práctica 2

# Gestión de un proyecto software. El programa `make`

## Introducción

La gestión y mantenimiento del software durante el proceso de desarrollo puede ser una tarea ardua si éste se estructura en diferentes ficheros fuente y se utilizan, además, funciones ya incorporadas en ficheros de biblioteca. Durante el proceso de desarrollo se modifica frecuentemente el software y las modificaciones incorporadas pueden afectar a otros módulos: la modificación de una función en un módulo afecta *necesariamente* a los módulos que usan dicha función, que deben actualizarse. Estas modificaciones deben *propagarse* a los módulos que dependen de aquellos que han sido modificados, de forma que el programa ejecutable final refleje las modificaciones introducidas.

Esta cascada de modificaciones afectará forzosamente al programa ejecutable final. Si esta secuencia de modificaciones no se realiza de forma ordenada y metódica podemos encontrarnos con un programa ejecutable que no considera las modificaciones introducidas. Este problema es tanto más acusado cuanto mayor sea la complejidad del proyecto software, lo que implica unos complejos diagramas de dependencias entre los módulos implicados, haciendo tedioso y propenso a errores el proceso de propagación hacia el programa ejecutable de las actualizaciones introducidas.

La utilidad `make` proporciona los mecanismos adecuados para la gestión de proyectos software. Esta utilidad mecaniza muchas de las etapas de desarrollo y mantenimiento de un programa, proporcionando mecanismos simples para obtener versiones actualizadas de los programas, por complicado que sea el diagrama de dependencias entre módulos asociado al proyecto. Esto se logra proporcionando a la utilidad `make` la secuencia de mandatos que crean ciertos archivos, y la lista de archivos que necesitan otros archivos (*lista de dependencias*) para ser actualizados antes de que se hagan dichas operaciones. Una vez especificadas las dependencias entre los distintos módulos del proyecto, cualquier cambio en uno de ellos provocará la creación de una nueva versión de los módulos dependientes de aquel que se modifica, reduciendo al mínimo necesario e imprescindible el número de modulos a recompilar para crear un nuevo fichero ejecutable.

La utilización de la orden `make` exige la creación previa de un fichero de descripción llamado genéricamente `makefile`, que contiene las órdenes que debe ejecutar `make`, así como las dependencias entre los distintos módulos del proyecto. Este archivo de descripción es un fichero de texto.

La sintaxis del fichero `makefile` varía ligeramente de un sistema a otro, al igual que la sintaxis de `make`, si bien las líneas básicas son similares y la comprensión y dominio de ambos en un sistema hace que el aprendizaje para otro sistema sea una tarea trivial. Esta visión de generalidad es la que nos impulsa a estudiar esta utilidad y descartemos el uso de gestores de proyectos como los que proporcionan los entornos de programación integrados. En esta sección nos centraremos en la descripción de la utilidad `make` y en la sintaxis de los ficheros `makefile` de GNU.

Resumiendo, el uso de la utilidad `make` conjuntamente con los ficheros makefile proporcionan el mecanismo para realizar una gestión inteligente, sencilla y precisa de un proyecto software, ya que permite:

1. Una forma sencilla de especificar la dependencia entre los módulos de un proyecto software,
2. La recompilación únicamente de los módulos que han de actualizarse,
3. Obtener siempre la versión última que refleja las modificaciones realizadas, y
4. Un mecanismo *casi estándar* de gestión de proyectos software, independiente de la plataforma en la que se desarrolla.

# El programa *make*

La utilidad `make` utiliza las reglas descritas en el fichero `makefile` para determinar qué ficheros ha de construir y cómo construirlos.

Examinando las listas de dependencia determina qué ficheros ha de reconstruir. El criterio es muy simple, se comparan fechas y horas: si el fichero fuente es más reciente que el fichero destino, reconstruye el destino. Este sencillo mecanismo hace posible mantener siempre actualizada la última versión.

## Sintaxis

La sintaxis de la llamada al programa `make` es la siguiente:

`make` [*opciones*] [*destino(s)*]

donde:

- cada ***opción*** va precedida por un signo `-` o una barra inclinada `/`.
- ***destino*** indica el destino que debe crear. Generalmente se trata del fichero que debe crear o actualizar, estando especificado en el fichero makefile el procedimiento de creación/actualización del mismo (página 35). Una explicación detallada de los destinos puede encontrarse en las páginas 32 y 36.

Obsérvese que tanto las opciones como los destinos son *opcionales*.

## Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

`-h` ó `--help` Proporciona ayuda acerca de `make`.

`-f` *fichero*. Utilizaremos esta opción si se proporciona a `make` un nombre de fichero distinto del de `makefile` o `Makefile`. Se toma el fichero llamado `fichero` como el fichero *makefile*.

`-n` , `--just-print`, `--dry-run` ó `--recon`: Muestra las instrucciones que *ejecutaría* la utilidad `make`, pero **no** los ejecuta. Sirve para verificar la corrección de un fichero makefile.

`-p` , `--print-data-base`: Muestra las reglas y macros asociadas al fichero makefile, incluidas las *predefinidas*.

## Funcionamiento de *make*

El funcionamiento de la utilidad `make` es el siguiente:

1. En primer lugar, busca el fichero makefile que debe interpretar. Si se ha especificado la opción `-f` *fichero*, busca ese fichero. Si no, busca en el directorio actual un fichero llamado `makefile` ó `Makefile`. En cualquier caso, si lo encuentra, lo interpreta; si no, da un mensaje de error y termina.

2. Intenta construir el(los) destino(s) especificado(s). Si no se proporciona ningún destino, intenta construir *solamente* el primer destino que aparece en el fichero makefile. Para construir un destino es posible que deba construir antes otros destinos si el destino especificado depende de otros que no están construidos. Para saber qué destinos debe construir comprueba las listas de dependencias. Esta reacción en cadena se llama **dependencia encadenada**.

3. Si en cualquier paso falla al construir algún destino, se detiene la ejecución, muestra un mensaje de error y borra el destino que estaba construyendo.

# Ficheros *makefile*

Un fichero makefile contiene las órdenes que debe ejecutar la utilidad `make`, así como las dependencias entre los distintos módulos del proyecto. Este archivo de descripción es un fichero de texto.

Los elementos que pueden incluirse en un fichero makefile son los siguientes:

1. Comentarios.
2. Reglas explícitas.
3. Órdenes.
4. Destinos simbólicos.

## Comentarios

Los comentarios tienen como objeto clarificar el contenido del fichero makefile. Una línea del comentario tiene en su primera columna el símbolo #. Los comentarios tienen el ámbito de una línea.

**Ejercicio**

Crear un fichero llamado `makefile` con el siguiente contenido:

```
# Fichero: makefile
# Construye el ejecutable saludo a partir de saludo.cpp
bin/saludo : src/saludo.cpp
    g++ src/saludo.cpp -o bin/saludo
```

Se incluyen dos líneas de comentario al principio del fichero `makefile` que indican las tareas que realizará la utilidad `make`.
Si el fichero `makefile` se encuentra en el directorio actual, para realizar las acciones en él indicadas tan solo habrá que ejecutar la orden:
`% make`
ya que el fichero makefile se llama `makefile`. El mismo efecto hubiéramos obtenido ejecutando la orden:
`% make -f makefile`

## Reglas. Reglas explícitas

Las reglas constituyen el mecanismo por el que se indica a la utilidad `make` los destinos (*objetivos*), las listas de dependencias y cómo construir los destinos. Como puede deducirse, son la parte fundamental de un fichero makefile. Las reglas que instruyen a `make` son de dos tipos: explícitas e implícitas y se definen de la siguiente forma:

- Las **reglas explícitas** dan instrucciones a `make` para que construya los ficheros especificados.
- Las **reglas implícitas** dan instrucciones generales que `make` sigue cuando no puede encontrar una regla explícita.

El formato habitual de una regla explícita es el siguiente:

**objetivo:** ***lista de dependencia***
*orden(es)*

donde:

- El **objetivo** identifica la regla e indica el fichero a crear.
- La ***lista de dependencia*** especifica los ficheros de los que depende **objetivo**. Esta lista contiene los nombres de los ficheros separados por espacios en blanco.

  Si alguno de los ficheros especificados en esta lista se ha modificado, se busca una regla que contenga a ese fichero como destino y se construye. Una vez se han construido las últimas versiones de los ficheros especificados en ***lista de dependencia*** se construye **objetivo**.
- Las ***órden(es)*** son órdenes válidas para el sistema operativo en el que se ejecute la utilidad `make`. Pueden incluirse varias instrucciones en una regla, cada uno en una línea distinta. Usualmente estas instrucciones sirven para construir el **objetivo** (en esta asignatura, habitualmente son llamadas al compilador `g++`), aunque no tiene porque ser así.

  **MUY IMPORTANTE:** Cada línea de órdenes empezará con un TABULADOR. Si no es así, `make` mostrará un error y no continuará procesando el fichero makefile.

**Ejercicio**

En el ejemplo anterior (fichero `makefile`) encontramos una única regla:

```
bin/saludo : src/saludo.cpp
    g++ src/saludo.cpp -o bin/saludo
```

que indica que para construir el objetivo `saludo` se requiere la existencia de `saludo.cpp` (`saludo` *depende de* `saludo.cpp`). Esta dependencia se esquematiza en el diagrama de dependencias mostrado en la figura 12. Finalmente, el destino se construye ejecutando la orden:

```
    g++ src/saludo.cpp -o bin/saludo
```

que compila el fichero `saludo.cpp` generando un fichero objeto temporal y lo enlaza con las bibliotecas adecuadas para generar finalmente `saludo`.

1. Ejecutar las siguientes órdenes e interpretar el resultado:

```
% make
% make -f makefile
% make bin/saludo
% make -f makefile bin/saludo
```

2. Antes de volver a ejecutar `make` con las cuatro variantes enumeradas anteriormente, modificar el fichero `saludo.cpp`. Interpretar el resultado.

3. Probar la orden `touch` sobre `saludo.cpp` y volver a ejecutar `make`. Interpretar el resultado.


Figura 12: Diagrama de dependencias para `saludo`

**Ejercicio**

A partir del diagrama de dependencias mostrado en la figura 13 construir el fichero makefile llamado `makefile2.mak`.
Ejecutar las siguientes órdenes e interpretar el resultado:

```
% make -f makefile2.mak bin/saludo

% make -f makefile2.mak
```

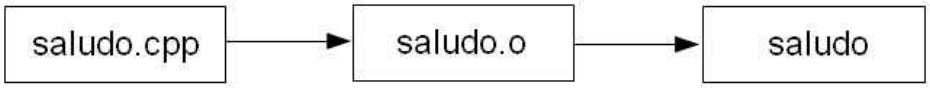



Figura 13: Diagrama de dependencias para `makefile2.mak`

## Órdenes. Prefijos de órdenes

Como se indicó anteriormente, se puede incluir cualquier orden válida del sistema operativo en el que se ejecute la utilidad `make`. Pueden incluirse cuantas órdenes se requieran como parte de una regla, cada una en una línea distinta, y como nota importante, recordamos que es *imprescindible* que cada línea empiece con un tabulador para que `make` interprete correctamente el fichero `makefile`.

Las órdenes pueden ir precedidas por **prefijos**. Los más importantes son:

- `@` Desactivar el *eco* durante la ejecución de esa orden.
- `-` Ignorar los errores que puede producir la orden a la que precede.

**Ejercicio**

Copiar el fichero `makefile3.mak` en vuestro directorio de trabajo.
En este fichero makefile se especifican dos órdenes en cada una de las reglas, entre ellas una para mostrar un mensaje en pantalla (`echo`), indicando qué acción se desencadena en cada caso. Las órdenes `echo` van precedidos por el prefijo `@` en el fichero makefile para indicar que debe desactivarse el eco durante la ejecución de esa instrucción.

1. Ejecutar `make` sobre este fichero makefile e interpretar el resultado.
2. Poner el prefijo `@` en la llamada a `g++` y volver a ejecutar `make`.
3. Eliminar los prefijos y volver a ejecutar `make`.

**Ejercicio**

Usando como base `makefile3.mak` añadir (al final) la siguiente regla, y guardar el nuevo contenido en el fichero `makefile4.mak`:

```
# Esta regla especifica un destino sin lista de dependencia
clean :
    @echo Borrando ficheros objeto
    rm obj/*.o
```

Esta nueva regla, cuyo destino es `clean` no tiene asociada una lista de dependencia. La construcción del destino `clean` no requiere la construcción de otro destino previo ya que la construcción de ese destino no depende de nada. Para ello bastará ejecutar:

```
% make -f makefile4.mak clean
```

## Destinos Simbólicos

Un destino simbólico se especifica en un fichero makefile en la primera línea operativa del mismo. En su sintaxis se asemeja a la especificación de una regla, con la diferencia que no tiene asociada ninguna orden. El formato es el siguiente:

**destino simbólico:** ***lista de destinos***

donde:

- **destino simbólico** es el nombre del destino simbólico. El nombre particular no tiene ninguna importancia, como se deducirá de nuestra explicación.
- ***lista de destinos*** especifica los destinos que se construirán cuando se invoque a `make`.

La finalidad de incluir un destino simbólico en un fichero makefile es la de que se construyan varios destinos sin necesidad de invocar a `make` tantas veces como destinos se desee construir.

Al estar en la primera línea operativa del fichero makefile, la utilidad `make` intentará construir el **destino simbólico**. Para ello, examinará la lista de dependencia (llamada ahora *lista de destinos*) y construirá cada uno de los destinos de esta lista: **debe existir una regla para cada uno de los destinos**. Finalmente, intentará construir el destino simbólico y como no habrá ninguna instrucción que le indique a `make` cómo ha de construirlo no hará nada más. Pero el objetivo está cumplido: se han construido varios destinos con una sóla ejecución de `make`. Obsérvese cómo el nombre dado al destino simbólico no tiene importancia.

**Ejercicio**

Copiar el fichero `unico.cpp` en vuestro directorio de trabajo. Construir el fichero `makefile5.mak` a partir de `makefile4.mak` con un destino simbólico llamado `todo` que cree los ejecutables `saludo` y `unico`. Ejecutar:

1. `% make -f makefile5.mak todo`
2. `% make -f makefile5.mak`

**Ejercicio**

Añadir el destino `clean` a la lista de dependencia del destino simbólico `todo`, así como la regla asociada.
Añadir un destino llamado `salva` a la lista de dependencia del destino simbólico `todo`, así como la regla asociada:

```
salva : saludo unico
    echo Creando directorio resultado
    mkdir resultado
    echo Moviendo los ejecutables al directorio resultado
    move $^ resultado
```

En la última orden asociada a la última regla se hace uso de la macro `$^` que se sustituye por todos los nombres de los ficheros de la lista de dependencias. O sea, `make` interpreta la orden anterior como:

```
    move saludo unico resultado
```

## Destinos .PHONY

Si en un fichero makefile apareciera una regla:

```
clean :
  @echo Borrando ficheros objeto
  rm obj/*.o
```

y hubiera un fichero llamado `clean`, la ejecución de la orden:

```
make clean
```

no funcionará como está previsto (no borrará los ficheros) puesto que el destino `clean` no tiene ninguna dependencia y existe el fichero `clean`. Así, `make` supone que no tiene que volver a generar el fichero `clean` con la orden asociada a esta regla, pues el fichero `clean` está actualizado, y no ejecutaría la orden que borra los ficheros de extensión `.o`. Una forma de solucionarlo es declarar este tipo de destinos como *falsos (phony)* usando `.PHONY` de la siguiente forma:

```
.PHONY : clean
```

Esta regla la podemos poner en cualquier parte del fichero makefile, aunque normalmente se coloca antes de la regla `clean`. Haciendo esto, al ejecutar la orden

```
make clean
```

todo funcionará bien, aunque exista un fichero llamado `clean`. Observar que también sería conveniente hacer lo mismo para el caso del destino simbólico `saludos` en los ejemplos anteriores.

## Macros en ficheros *makefile*

Una **macro o variable MAKE** es una *cadena* que se expande cuando se usa en un fichero makefile.

Las macros permiten crear ficheros makefile genéricos o *plantilla* que se adaptan a diferentes proyectos software. Una macro puede representar listas de nombres de ficheros, opciones del compilador, programas a ejecutar, directorios donde buscar los ficheros fuente, directorios donde escribir la salida, etc. Puede verse como una versión más potente que la directiva `#define` de C++, pero aplicada a ficheros makefile.

La sintaxis de definición de macros en un fichero makefile es la siguiente:

**NombreMacro = texto a expandir**

donde:

- **NombreMacro** es el nombre de la macro. Es sensible a las mayúsculas y no puede contener espacios en blanco. La costumbre es utilizar nombres en mayúscula.
- **texto a expandir** es una cadena que puede contener cualquier carácter alfanumérico, de puntuación o espacios en blanco

Para definir una macro llamada, por ejemplo, `OBJ` que representa a la cadena `~/MP/obj` se especificará de la siguiente manera:

```
OBJ = ~/MP/obj
```

Si esta línea se incluye en el fichero makefile, cuando `make` encuentra la construcción `$(OBJ)` en él, sustituye dicha construcción por `~/MP/obj`. Cada macro debe estar en una línea separada en un fichero makefile y se situan, normalmente, al principio de éste. Si `make` encuentra más de una definición para el mismo nombre (no es habitual), la nueva definición reemplaza a la antigua.

La expansión de la macro se hace *recursivamente*. O sea, que si la macro contiene referencias a otras macros, estas referencias serán expandidas también. Veamos un ejemplo.

Podemos definir una macro para cada uno de los directorios de trabajo:

```
SRC = src
BIN = bin
OBJ = obj
INCLUDE = include
LIB = lib
```

Si el fichero makefile está situado en la misma carpeta que los directorios, y en el fichero aparece:

```
$(BIN)/saludo: $(SRC)/saludo.cpp
     g++ -o $(BIN)/saludo $(SRC)/saludo.cpp
```

se sustituye por:

```
bin/saludo: src/saludo.cpp
     g++ -o bin/saludo src/saludo.cpp
```

¿y si el fichero makefile estuviera en una carpeta distinta? Podría añadirse una macro (la primera):

```
HOMEDIR = /home/users/app/new/MP
```

y se modifican las anteriores por:

```
SRC = $(HOMEDIR)/src
BIN = $(HOMEDIR)/bin
OBJ = $(HOMEDIR)/obj
INCLUDE = $(HOMEDIR)/include
LIB = $(HOMEDIR)/lib
```

Ahora, la regla anterior se sustituye por:

```
/home/users/app/new/MP/bin/saludo: /home/users/app/new/MP/saludo.cpp
     g++ -o /home/users/app/new/MP/saludo
            /home/users/app/new/MP/saludo.cpp
```

Si los directorios se situaran en otra carpeta bastará con cambiar la macro `HOMEDIR`. En nuestro caso podríamos escribir:

```
HOMEDIR = ~/MP
```

**Ejercicio**

Modificar el fichero `makefile5.mak` para que incluya las macros referentes a los directorios que hemos enumerado anteriormente.

## Macros predefinidas

Las macros predefinidas utilizadas habitualmente en ficheros makefile son las que enumeramos a continuación:

`$@` Nombre del fichero *destino* de la regla.

`$<` Nombre de la *primera dependencia* de la regla.

`$^` Equivale a *todas* las dependencias de la regla, con un espacio entre ellas.

`$?` Equivale a *las dependencias de la regla más nuevas que el destino*, con un espacio entre ellas.

**Ejercicio**

Modificar el fichero `makefile5.mak` de manera que se muestren los valores de las macros predefinidas `$@` , `$<`, `$^`, y `$?` en las reglas que generan algún fichero como resultado. Usad la orden `@echo`

## Sustituciones en macros

La utilidad `make` permite sustituir caracteres temporalmente en una macro previamente definida. La sintaxis de la sustitución en macros es la siguiente:

**$(NombreMacro:TextoOriginal = TextoNuevo)**

que se interpreta como: sustituir en la cadena asociada a **NombreMacro** todas las apariciones de **TextoOriginal** por **TextoNuevo**. Es importante resaltar que:

1. **No** se permiten espacios en blanco antes o despues de los dos puntos.
2. **No** se redefine la macro **NombreMacro**, se trata de una sustitución *temporal*, por lo que **NombreMacro** mantiene el valor dado en su definición.

Por ejemplo, dada una macro llamada `FUENTE` definida como:

```
FUENTES = f1.cpp f2.cpp f3.cpp
```

se pueden sustituir *temporalmente* los caracteres `.cpp` por `.o` escribiendo `$(FUENTES:.cpp=.o)` que da como resultado `f1.o f2.o f3.o`. El valor de la macro `FUENTES` **no** se modifica, ya que la sustitución es temporal.

## Macros como parámetros en la llamada a `make`

Además de las opciones básicas indicadas en la página 29, hay una opción que permite especificar el valor de una constante simbólica que se emplea en un fichero makefile en la llamada a `make` en lugar de especificar su valor en el fichero makefile.

El mecanismo de sustitución es similar al expuesto anteriormente, salvo que ahora `make` no busca el valor de la macro en el fichero makefile. La sintaxis de la llamada a `make` con macros es la siguiente:

`make` ***NombreMacro***[=***cadena***] [*opciones...*] [*destino(s)*]]

> ***NombreMacro***[=***cadena***] define la constante simbólica ***NombreMacro*** con el valor especificado (si lo hubiera) después del signo =. Si ***cadena*** contiene espacios, será necesario encerrar ***cadena*** entre comillas.
>
> Si ***NombreMacro*** también está definida dentro del fichero makefile, se ignorará la definición del fichero.

El uso de macros en llamadas a `make` permite la construcción de ficheros makefile genéricos, ya que el mismo fichero puede utilizarse para diferentes tareas que se deciden en el momento de invocar a `make` con el valor apropiado de la macro.

**Ejercicio**

Modificar el fichero `makefile_ppal_2` para que el resultado (el ejecutable `ppal_2`) lo guarde en un directorio cuyo nombre **completo** (camino absoluto, desde la raiz `/`) se indica como parámetro al fichero makefile con una macro llamada `DESTDIR`.
**Importante**: Como el directorio puede no existir, crearlo en el propio fichero makefile.

1. Ejecutar `make` sobre este fichero especificando el directorio destino apropiadamente.
2. ¿Qué ocurre si se vuelve a ejecutar la orden anterior?
3. Modificar apropiadamente el fichero `makefile_ppal_2` para evitar el error.

**Ejercicio**

Extender el makefile anterior con una opción para incluir información de depuración o no.

# Anexo I: Reglas implícitas

En los ficheros makefile aparecen, en la mayoría de los casos, reglas que se parecen mucho. En los ejercicios anteriores se puede ver, por ejemplo, que las reglas que generan los ficheros objeto son idénticas, sólo se diferencian en los nombres de los ficheros que manipulan.

Las reglas implícitas son reglas que `make` interpreta para actualizar destinos sin tener que escribirlas dentro del fichero makefile. De otra forma: *si no se especifica una regla explícita para construir un destino, se utilizará una regla implícita (que no hay que escribir)*.

Existe un catálogo de reglas implícitas predefinidas que pueden usarse para distintos lenguajes de programación (ver `http://www.gnu.org/software/make/manual/make.html#Implicit-Rules`) .

El que `make` elija una u otra dependerá del nombre y extensión de los ficheros.

Por ejemplo, para el caso que nos interesa, compilación de programas en C++, existe una regla implícita que dice cómo obtener el fichero objeto (`.o`) a partir del fichero fuente (`.cpp`). Esa regla se usa cuando no existe una regla explícita que diga como construir ese módulo objeto. En tal caso se ejecutará la siguiente orden para construir el módulo objeto cuando el fichero fuente sea modificado:

```
$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
```

Las reglas implícitas que usa `make` utilizan una serie de macros predefinidas, tales como las del caso anterior (`CXX`, `CPPFLAGS` y `CXXFLAGS`). Para más información ver

`http://www.gnu.org/software/make/manual/make.html#Implicit-Variables`

El valor de estas macros pueden ser definidas:

1. Dentro del fichero makefile,
2. A través de argumentos pasados a `make`, o
3. Con valores predefinidos. Por ejemplo,
   - `CXX`: Programa para compilar programas en C++; Por defecto: `g++`.
   - `CPPFLAGS`: Modificadores extra para el preprocesador de C. El valor por defecto es la cadena vacía.
   - `CXXFLAGS`: Modificadores extra para el compilador de C++. El valor por defecto es la cadena vacía.

Si deseamos que `make` use reglas implícitas, en el fichero makefile escribiremos la regla *sin ninguna orden*. Es posible añadir nuevas dependencias a la regla.

**Ejercicio**

Usar como base `makefile_ppal_2` y copiarlo en `makefile_ppal_3`, modificando la regla que crea el ejecutable para que éste se llame `ppal_3`.

1. Eliminar las reglas que crean los ficheros objeto y ejecutar `make`.
2. Forzar la dependencia de los módulos objeto respecto a los ficheros de cabecera (`.h`) adecuados para que cuando se modifique algún fichero de cabecera se ejecute la regla implícita que actualiza el o los ficheros objeto necesarios, y finalmente el ejecutable.

   Nota: Los ficheros de cabecera se encuentran en un subdirectorio del directorio `MP` llamado `include`. Modificar la variable `CXXFLAGS` con el valor `-I$(INCLUDE)` (se expanderá a `-I./include` para cada ejecución de la regla implícita).

   *a*) Modificar (`touch`) `adicion.cpp` y ejecutar `make`. Observad qué instrucciones se ejecutan y en qué orden.

   *b*) Modificar (`touch`) `adicion.h` y ejecutar `make`. Observad qué instrucciones se ejecutan y en qué orden.

Otra regla que puede usarse es la regla implícita que permite enlazar el programa. La siguiente regla:

```
$(CC) $(LDFLAGS) n.o $(LOADLIBES) $(LDLIBS)
```

se interpreta como sigue: `n` se construirá a partir de `n.o` usando el enlazador (`ld`). Esta regla funciona correctamente para programas *con un solo fichero fuente* aunque también funcionarácorrectamente en programas con múltiples ficheros objeto si uno de los cuales tiene el mismo nombre que el ejecutable.

**Ejercicio**

Copiar `ppal_2.cpp` en `ppal_4.cpp`
Usar como base `makefile_ppal_3` y copiarlo en `makefile_ppal_4`,
Eliminar la orden en la regla que crea el ejecutable manteniendo la lista de dependencia y ejecutar `make`.

## Reglas implícitas patrón

Las reglas implícitas patrón pueden ser utilizadas por el usuario para definir nuevas reglas implícitas en un fichero makefile. También pueden ser utilizadas para redefinir las reglas implícitas que proporciona `make` para adaptarlas a nuestras necesidades. Una *regla patrón* es parecida a una regla normal, pero el destino de la regla contiene el caracter `%` en alguna parte (sólo una vez). Este destino constituye entonces un patrón para emparejar nombres de ficheros.

Por ejemplo el destino `%.o` empareja a todos los ficheros con extensión `.o`. Una regla patrón `%.o:%.cpp` dice cómo construir cualquier fichero `.o` a partir del fichero `.cpp` correspondiente. En una regla patrón podemos tener varias dependencias que también pueden contener el caracter `%`. Por ejemplo:

```
%.o : %.cpp %.h comun.h
    g++ -c $< -o $@
```

significa que cada fichero `.o` debe volver a construirse cuando se modifique el `.cpp` o el `.h` correspondiente, o bien `comun.h`. Una reglas patrón del tipo `%.o:%.cpp` puede simplificarse escribiendo `.cpp.o:`

La *regla implícita patrón predefinida* para compilar ficheros `.cpp` y obtener ficheros `.o` es la siguiente:

```
%.o : %.cpp
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@
```

Esta regla podría ser redefinida en nuestro fichero makefile escribiendo esta regla con el mismo destino y dependencias, pero modificando las órdenes a nuestra conveniencia. Si queremos que `make` ignore una regla implícita podemos escribir una regla patrón con el mismo destino y dependencias que la regla implícita predefinida, y sin ninguna orden asociada.

**IMPORTANTE:** Una regla regla implícita patrón puede aplicarse a cualquier destino que se empareja con su patrón, pero sólo se aplicará cuando el destino no tiene órdenes que lo construya mediante otra regla distinta, y sólo cuando puededan encontrase las dependecias. Cuando se puede aplicar más de una regla implícita, sólo se aplicará una de ellas: la elección depende del orden de las reglas.

## Reglas patrón estáticas

Las reglas patrón estáticas son otro tipo de reglas que se pueden utilizar en ficheros makefile y que son muy parecidas en su funcionamiento a las reglas implícitas patrón. Estas reglas no se consideran implícitas, pero al ser muy parecidas en funcionamiento a las reglas patrón implícitas, las exponemos en esta sección. El formato de estas reglas es el siguiente:

**destino(s)**: *patrón de destino* : *patrones de dependencia*
*orden(es)*

donde:

- La lista *destinos* especifica a qué destinos se aplicará la regla. Esta es la principal diferencia con las reglas patrón implícitas. Ahora la regla se aplica únicamente a la lista de destinos, mientras que en las implícitas se intenta aplicar a todos los que se emparejan con el patrón destino. Los destinos pueden contener caracteres comodín como `*` y `?`
- El *patrón de destino* y los *patrones de dependencia* dicen cómo calcular las dependencias para cada destino.

Veamos un pequeño ejemplo que muestra como obtener los ficheros `f1.o` y `f2.o`.

```
OBJETOS = f1.o f2.o

$(OBJETOS): %.o: %.cpp
        g++ -c $(CFLAGS) $< -o $@
```

En este ejemplo, la regla sólo se aplica a los ficheros `f1.cpp` y `f2.cpp`. Si existen otros ficheros con extensión `.cpp`, esta regla no se aplicará ya que no se han incluido en la lista *destinos*.

# Anexo II: Directivas condicionales en ficheros *makefile*

Las directivas condicionales se parecen a las directivas condicionales del preprocesador de C++. Permiten a `make` dirigir el flujo de procesamiento en un fichero makefile a un bloque u otro dependiendo del resultado de la evaluación de una condición, evaluada con una directiva condicional.

Para más información ver
`http://www.gnu.org/software/make/manual/make.html#Conditionals`

La sintaxis de un condicional simple sin else sería la siguiente:

*directiva condicional*
    *texto (si el resultado es **verdad**)*
`endif`

La sintaxis para un condicional con parte `else` sería:

*directiva condicional*
    *texto (si el resultado es **verdad**)*
`else`
    *texto (si el resultado es **falso**)*
`endif`

Las directivas condicionales para ficheros makefile son las siguientes:

`ifdef` *macro*: Actúa como la directiva `#ifdef` de C++ pero con macros en lugar de directivas `#define`.

`ifndef` *macro*: Actúa como la directiva `#ifndef` de C++ pero con macros, en lugar de directivas `#define`.

`ifeq (`*arg1*`,`*arg2*`)` ó `ifeq '`*arg1*`' '`*arg2*`'` ó `ifeq "`*arg1*`" "`*arg2*`"` ó `ifeq "`*arg1*`" '`*arg2*`'` ó `ifeq '`*arg1*`' "`*arg2*`"`

Devuelve verdad si los dos argumentos expandidos son iguales.

`ifneq (`*arg1*`,`*arg2*`)` ó `ifneq '`*arg1*`' '`*arg2*`'` ó `ifneq "`*arg1*`" "`*arg2*`"` ó `ifneq "`*arg1*`" '`*arg2*`'` ó `ifneq '`*arg1*`' "`*arg2*`"`

Devuelve verdad si los dos argumentos expandidos son distintos

`else`

Actúa como un `else` de C++.

`endif`

Termina una declaración `ifdef`, `ifndef` `ifeq` ó `ifneq`.

# Guión de la práctica 3

# Modularización de software. Bibliotecas. El programa `ar`

## Introducción

Cuando se escriben programas medianos o grandes resulta sumamente recomendable (por no decir *obligado*) dividirlos en diferentes módulos fuente. Este enfoque proporciona muchas e importantes ventajas, aunque complica en cierta medida la tarea de compilación.

Básicamente, lo que se hará será dividir nuestro programa fuente en varios ficheros. Estos ficheros se compilarán por separado, obteniendo diferentes ficheros objeto. Una vez obtenidos, los módulos objeto se pueden, si se desea, reunir para formar bibliotecas. Para obtener el programa ejecutable, se enlazará el módulo objeto que contiene la función `main()` con varios módulos objeto y/o bibliotecas.

## Ventajas de la modularización del software

1. Los módulos contendrán de forma natural conjuntos de funciones relacionadas desde un punto de vista lógico.
2. Resulta fácil aplicar un enfoque orientado a objetos. Cada objeto (tipo de dato abstracto) se agrupa en un módulo junto con las operaciones del tipo definido.
3. El programa puede ser desarrollado por un equipo de programadores de forma cómoda. Cada programador puede trabajar en distintos aspectos del programa, localizados en diferentes módulos. pueden reusarse en otros programas, reduciendo el tiempo y coste del desarrollo del software.
4. La compilación de los módulos se puede realizar por separado. Cuando un módulo está validado y compilado no será preciso recompilarlo. Además, cuando haya que modificar el programa, sólo tendremos que recompilar los ficheros fuente alterados por la modificación. La utilidad `make` será muy útil para esta tarea.
5. El ocultamiento de información puede conseguirse con la modularización. El usuario de un módulo objeto o de una biblioteca de módulos desconoce los detalles de implementación de las funciones y objetos definidos en éstos. Mediante el uso de ficheros de cabecera proporcionaremos la interface necesaria para poder usar estas funciones y objetos.

## Cómo dividir un programa en varios ficheros

Cuando se divide un programa en varios ficheros, cada uno de ellos contendrá una o más funciones. Sólo uno de estos ficheros contendrá la función `main()`. Los programadores normalmente comienzan a diseñar un programa dividiendo el problema en subtareas que resulten más manejables. Cada una de estas tareas se implementará como una o más funciones. Normalmente, todas las funciones de una subtarea residen en un fichero fuente.

Por ejemplo, cuando se realice la implementación de tipos de datos abstractos definidos por el programador, lo normal será incluir todas las funciones que acceden al tipo definido en el mismo fichero. Esto supone varias ventajas importantes:

1. La estructura de datos se puede utilizar de forma sencilla en otros programas.
2. Las funciones relacionadas se almacenan juntas.
3. Cualquier cambio posterior en la estructura de datos, requiere que se modifique y recompile el mínimo número de ficheros y sólo los imprescindibles.

Cuando las funciones de un módulo invocan objetos o funciones que se encuentran definidos en otros módulos, necesitan alguna información acerca de cómo realizar estas llamadas. El compilador requiere que se proporcionen declaraciones de funciones y/o objetos definidos en otros módulos. La mejor forma de hacerlo (y, probablemente, la única razonable) es mediante la creación de ficheros de cabecera (`.h`), que contienen las declaraciones de las funciones definidas en el fichero `.cpp` correspondiente. De esta forma, cuando un módulo requiera invocar funciones definidas en otros módulos, bastará con que se inserte una línea `#include` del fichero de cabecera apropiado.

## Organización de los ficheros fuente

Los ficheros fuente que componen nuestros programas deben estar organizados en un cierto orden. Normalmente será el siguiente:

1. Una primera parte formada por una serie de constantes simbólicas en líneas `#define`, una serie de líneas `#include` para incluir ficheros de cabecera y redefiniciones (`typedef`) de los tipos de datos que se van a tratar.
2. La declaración de variables externas y su inicialización, si es el caso. Se recuerda que, en general, no es recomendable su uso.
3. Una serie de funciones.

El orden de los elementos es importante, ya que en el lenguaje C++, cualquier objeto debe estar declarado antes de que sea usado, y en particular, las funciones deben declararse antes de que se realice cualquier llamada a ellas. Se nos presentan dos posibilidades:

1. **Que la definición de la función se encuentre en el mismo fichero en el que se realiza la llamada**. En este caso,

   *a*) se sitúa la definición de la función antes de la llamada (ésta es una solución raramente empleada por los programadores),

   *b*) se puede incluir una linea de declaración (prototipo) al principio del fichero: la definición se puede situar en cualquier punto del fichero, incluso después de las llamadas a ésta.

2. **Que la definición de la función se encuentre en otro fichero diferente al que contiene la llamada**. En este caso es preciso incluir al principio del mismo una línea de declaración (prototipo) de la función en el fichero que contiene las llamadas a ésta. Normalmente se realiza incluyendo (mediante la directiva de preprocesamiento `#include`) el fichero de cabecera asociado al módulo que contiene la definición de la función.

   Cuando modularizamos nuestros programas, hemos de hacer un uso adecuado de los ficheros de cabecera asociados a los módulos que estamos desarrollando. Además, los ficheros de cabecera también son útiles para contener las declaraciones de tipos, funciones y otros objetos que necesitemos compartir entre varios de nuestros módulos.

   Así pues, a la hora de crear el fichero de cabecera asociado a un módulo debemos tener en cuenta qué objetos del módulo van a ser compartidos o invocados desde otros módulos (**públicos**) y cuáles serán estrictamente **privados** al módulo: en el fichero de cabecera pondremos, únicamente, los públicos.

   **Recordar**: *En C++, todos los objetos deben estar declarados y definidos antes de ser usados*.

**Ejercicio**

Copiar el fichero `unico.cpp` en vuestro directorio de trabajo. Generar el ejecutable `unico`.

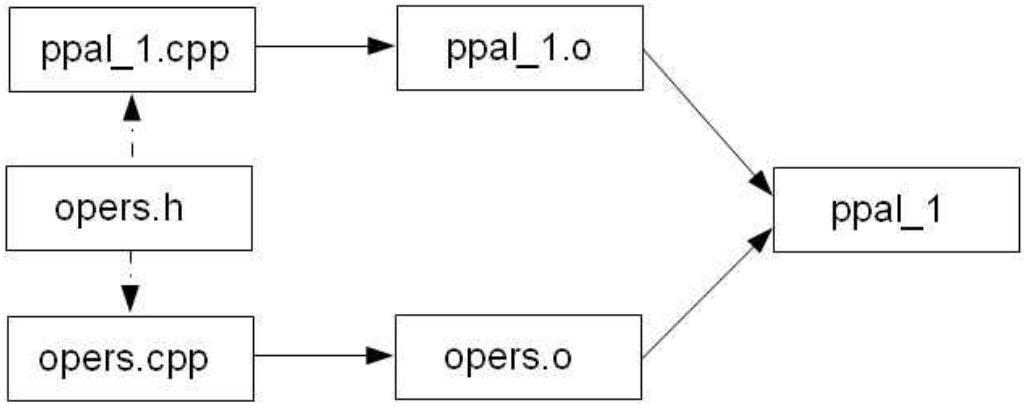



Figura 14: Diagrama de dependencias para construir `ppal_1`

**Ejercicio**

Organizar el código fuente en distintos ficheros de manera que separaremos la declaración y la definición de las funciones. Guardar cada fichero en la carpeta adecuada.

Los ficheros involucrados y sus dependencias se muestran en la figura 14. Con detalle:

1. El fichero (`ppal_1.cpp`) contendrá la función `main()`
2. El código asociado a las funciones se organiza en dos ficheros: uno contiene las declaraciones (`opers.h`) y otro las definiciones (`opers.cpp`)

Realizar las siguientes tareas:

1. Generar el objeto `ppal_1.o` a partir de `ppal_1.cpp`
2. Generar el objeto `opers.o` a partir de `opers.o`
3. Generar el ejecutable `ppal_1` a partir de los dos módulos objeto generados.

**Ejercicio**

Escribir un fichero llamado `makefile_ppal_1` para generar el ejecutable `ppal_1`, siguiendo el diagrama de dependencia mostrado en la figura 14.

Usad macros para especificar los directorios de trabajo.

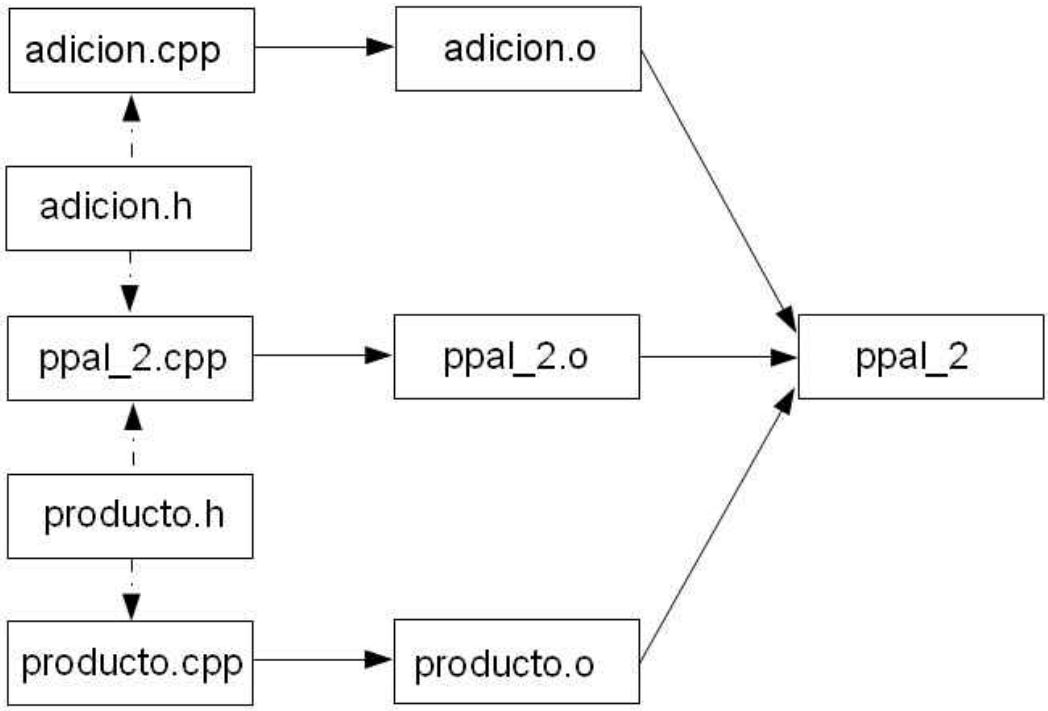



Figura 15: Diagrama de dependencias para construir `ppal_2`

**Ejercicio**

Continuaremos distribuyendo el código fuente en distintos ficheros, siguiendo el esquema indicado en la figura 15.

1. El fichero (`ppal_2.cpp`) contendrá la función `main()`
2. El código asociado a las funciones se organiza en cuatro ficheros organizados *en pares declaración-definición*:
   *a*) El primer par (`adicion.h` y `adicion.cpp`) contiene las funciones `suma()` y `resta()`
   *b*) El segundo (`producto.h` y `producto.cpp`) contiene las funciones `multiplica()` y `divide()`

Realizar las siguientes tareas:

1. Generar el objeto `ppal_2.o` a partir de `ppal_2.cpp`
2. Generar los objetos `adicion.o` y `producto.o`
3. Generar el ejecutable `ppal_2` a partir de los tres módulos objeto generados.

**Ejercicio**

Modificar la función `divide()` de `producto.cpp` de manera que procese adecuadamente la excepción que se genera cuando hay una división por cero.

1. Analizar qué módulos deben recompilarse para generar un nuevo ejecutable, actualizado con la modificación propuesta.
2. Volver a generar el ejecutable `ppal_2`

**Ejercicio**

Escribir un fichero llamado `makefile_ppal_2` para generar el ejecutable `ppal_2`, siguiendo el diagrama de dependencia mostrado en la figura 15.

Usad macros para especificar los directorios de trabajo.

# Bibliotecas

En la práctica de la programación se crean conjuntos de funciones útiles para muy diferentes programas: funciones de depuración de entradas, funciones de presentación de datos, funciones de cálculo, etc. Si cualquiera de esas funciones quisiera usarse en un nuevo programa, la práctica de "Copiar y Pegar" el trozo de código correspondiente a la función deseada no resulta, a la larga, una buena solución, ya que,

1. El tamaño total del código fuente de los programas se incrementa innecesariamente y es redundante.
2. Si se hace necesaria una actualización del código de una función es preciso modificar **TODOS** los programas que usan esta función, lo que llevará a utilizar diferentes versiones de la misma función si el control no es muy estricto o a un esfuerzo considerable para mantener la coherencia del software.

En cualquier caso, esta práctica llevará irremediablemente en un medio/largo plazo a una situación insostenible. La solución obvia consiste en agrupar estas funciones usadas frecuentemente en módulos de biblioteca, llamados comúnmente **bibliotecas**.

*Una **biblioteca** contiene código objeto que puede ser enlazado con el código objeto de un módulo que usa una función de esa biblioteca*.

De esta forma tan solo existe una versión de cada función por lo que la actualización de una función implicará únicamente recompilar el módulo donde está esa función y los módulos que usan esa función, sin necesidad de modificar nada más (siempre que no se modifique la cabecera de la función, y como consecuencia, la llamada a ésta, claro está). Si además nuestros proyectos se mantienen mediante ficheros makefile el esfuerzo de mantenimiento y recompilación se reduce drásticamente.

Esta **modularidad** redunda en un mantenimiento más eficiente del software y en una disminución del tamaño de los ficheros fuente, ya que tan sólo incluirán la llamada a las funciones de biblioteca, y no su definición.

Vulgarmente se emplea el término *librería* para referirse a una biblioteca, por la similitud con el original inglés *library*. En términos formales, la acepción correcta es **biblioteca**, porque es la traducción correcta de **library**, mientras que el término inglés para librería es *bookstore* o *book shop*. También es habitual referirse a ella con el término de origen anglosajón *toolkit* (conjunto, equipo, maletín, caja, estuche, juego (kit) de herramientas).

# Tipos de bibliotecas

### Bibliotecas estáticas

La dirección real, las referencias para saltos y otras llamadas a las funciones de las bibliotecas se almacenan en una *dirección relativa* o *simbólica*, que no puede resolverse hasta que todo el código es asignado a direcciones estáticas finales.

El enlazador resuelve todas las direcciones no resueltas convirtiéndolas en direcciones fijas, o relocalizables desde una base común. El enlace estático da como resultado un archivo ejecutable con todos los símbolos y módulos respectivos incluidos en dicho archivo. Este proceso se realiza antes de la ejecución del programa y debe repetirse cada vez que alguno de los módulos es recompilado.

La ventaja de este tipo de enlace es que hace que un programa no dependa de ninguna biblioteca (puesto que las enlazó al compilar), haciendo más fácil su distribución.

### Bibliotecas dinámicas

Enlace dinámico significa que los módulos de una biblioteca son cargadas en un programa en tiempo de ejecución, en lugar de ser enlazadas en tiempo de compilación, y se mantienen como archivos independientes separados del fichero ejecutable del programa principal.

El enlazador realiza una mínima cantidad de trabajo en tiempo de compilación, registra qué modulos de la biblioteca necesita el programa y el índice de nombres de los módulos en la biblioteca.

Algunos sistemas operativos sólo pueden enlazar una biblioteca en tiempo de carga, antes de que el proceso comience su ejecución, otros son capaces de esperar hasta después de que el proceso haya empezado a ejecutarse y enlazar la biblioteca sólo cuando efectivamente se hace referencia a ella (es decir, en tiempo de ejecución). Esto último se denomina retraso de carga". En cualquier caso, esa biblioteca es una biblioteca enlazada dinámicamente.

## Estructura de una biblioteca

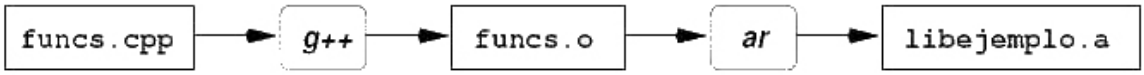
Una biblioteca se estructura internamente como un **conjunto de módulos objeto**. Cada uno de estos módulos será el resultado de la compilación de un fichero de código fuente que puede contener una o varias funciones.

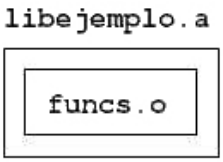
La extensión por defecto de los ficheros de biblioteca es `.a` y se acostumbra a que su nombre empiece por el prefijo `lib`. Así, por ejemplo, si hablamos de la biblioteca `ejemplo` el fichero asociado se llamará `libejemplo.a`

Veamos, con un ejemplo, cómo se estructura una biblioteca. La biblioteca `libejemplo.a` contiene 10 funciones. Los casos extremos en la construcción de esta biblioteca serían:

1. Está formada por *un único fichero objeto* (por ejemplo, `funcs.o`) que es el resultado de la compilación de un fichero fuente (`funcs.cpp`). Este caso se ilustra en la figura 16.

Figura 16: Construcción de una biblioteca a partir de un único módulo objeto (caso 1)




```
// funcs.cpp
// Contiene la definicion de 10 funciones
//

int funcion_1 (int a, int b)
{
   ...........
}

char *funcion_2 (char *s, char *t)
{
   ...........
}

...........

int funcion_10 (char *s, int x)
{
   ...........
}
```

2. Está formada por 10 ficheros objeto (por ejemplo, `fun01.o`, ..., `fun10.o`) resultado de la compilación de 10 ficheros fuente (por ejemplo, `fun01.cpp`, ..., `fun10.cpp`) que contienen, cada uno, la definición de una única función. Este caso se ilustra en las figuras 17 y 18.

fun01.cpp

```
// fun01.cpp
// Contiene la definicion de "funcion_1()"
//

int funcion_1 (int a, int b)
{
   ...........
}
```

fun10.cpp

```
// fun10.cpp
// Contiene la definicion de "funcion_10()"
//

int funcion_10 (char *s, int x)
{
   ...........
}
```

Figura 17: Varios módulos fuente (caso 2)

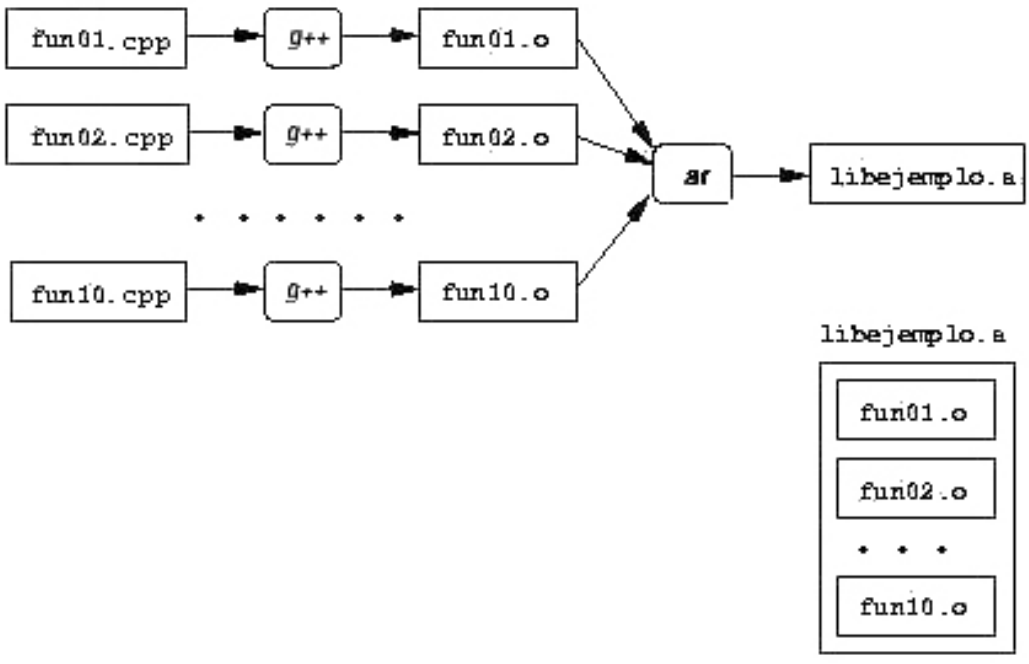



Figura 18: Construcción de una biblioteca a partir de varios módulos objeto (caso 2)

Para generar un fichero ejecutable que utiliza una función de una biblioteca, el enlazador **enlaza el módulo objeto que contiene la función `main()` con el módulo objeto (completo) de la biblioteca donde se encuentra la función utilizada**. De esta forma, *en el ejecutable sólo se incluirán los módulos objeto que contienen alguna función llamada por el programa*.

Por ejemplo, supongamos que `ppal.cpp` contiene la función `main()`. Esta función usa la función `funcion_2()` de la biblioteca `libejemplo.a`. Independientemente de la estructura de la biblioteca, `ppal.cpp` se escribirá de la siguiente forma:

```
#include "ejemplo.h"
// Contiene: prototipos de las funciones de "libejemplo.a"

int main (void)
{
   ......
   cad1 = funcion_2 (cad2, cad3);
   ......
}
```

Como regla general **cada biblioteca llevará asociado un fichero de cabecera** que contendrá los **prototipos** de las funciones que se ofrecen en la biblioteca (**funciones públicas**). Este fichero de cabecera actúa de *interface* entre las funciones de la biblioteca y los programas que la usan.

En nuestro ejemplo, independientemente de la estructura interna de la biblioteca, ésta ofrece 10 funciones cuyos prototipo se encuentran declarados en `ejemplo.h`.

Para la elección de la estructura óptima de la biblioteca hay que recordar que la parte de la biblioteca que se enlaza al código objeto de la función `main()` es el módulo objeto **completo** en el que se encuentra la función de biblioteca usada. En la figura 19 mostramos cómo se construye un ejecutable a partir de un módulo objeto (`ppal.o`) que contiene la función `main()` y una biblioteca (`libejemplo.a`).

Sobre esta figura, distinguimos dos situaciones diferentes dependiendo de cómo se construye la biblioteca. En ambos casos el fichero objeto que se enlazará con la biblioteca (con más precisión, con el módulo objeto adecuado de la biblioteca) se construye de la misma forma: el programa que usa una función de biblioteca no conoce (ni le importa) cómo se ha construído la biblioteca.

- **Caso 1**: El módulo objeto `ppal.o` se enlaza con el módulo objeto `funcs.o` (extraido de la biblioteca `libejemplo.a`) para generar el ejecutable `prog_1`. El módulo `funcs.o` contiene el código objeto de **todas** las funciones, por lo que se enlaza mucho código que no se usa.
- **Caso 2**: El módulo objeto `ppal.o` se enlaza con el módulo objeto `fun02.o` (extraido de la biblioteca `libejemplo.a`) para generar el ejecutable `prog_2`. El módulo `fun02.o` contiene **únicamente** el código objeto de la función que se usa, por lo que se enlaza el código estrictamente necesario.

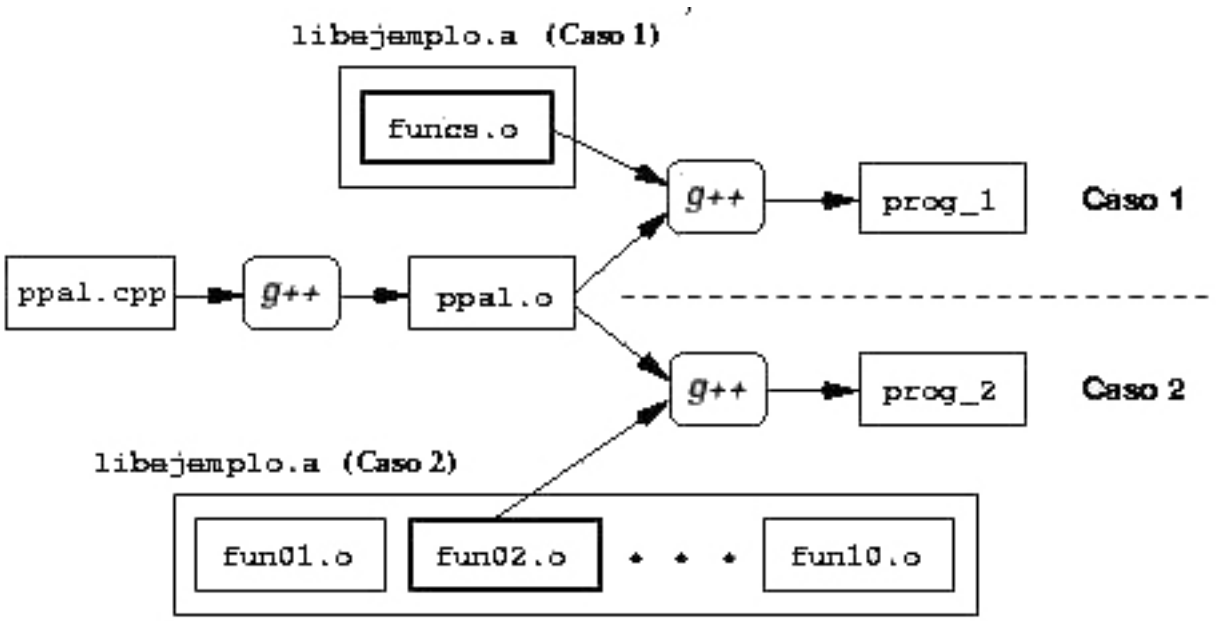



Figura 19: Construcción de un ejecutable que usa una función de biblioteca. Caso 1: biblioteca formada por un módulo objeto. Caso 2: biblioteca formada por 10 módulos objeto

En cualquier caso, como **usuario** de la biblioteca se actúa de la misma manera en ambas situaciones: enlaza el módulo objeto que conntiene la función `main()` con la biblioteca.

Como **diseñador** de la biblioteca sí debemos tener en cuenta la estructura que vamos a adoptar para ésta. La idea básica es conocida: si las bibliotecas están formadas por módulos objeto con muchas funciones cada una, los tamaños de los ejecutables serán muy grandes. En cambio, si las bibliotecas están formadas por módulos objeto con pocas funciones cada una, los tamaños de los ejecutables serán más pequeños[2].

[2]Estas afirmaciones suponen que las funciones son equiparables en tamaño, obviamente

# El programa `ar`

El programa gestor de bibliotecas de GNU es `ar`. Con este programa es posible crear y modificar bibliotecas existentes: añadir nuevos módulos objeto, eliminar módulos objeto o reemplazarlos por otros más recientes. La sintaxis de la llamada a `ar` es:

`ar` [`-`]*operación* [*modificadores* ] *biblioteca* [*módulos objeto*]

donde:

- *operación* indica la tarea que se desea realizar sobre la biblioteca. Éstas pueden ser:

  `r` **Adición o reemplazo**. Reemplaza el módulo objeto de la biblioteca por la nueva versión. Si el módulo no se encuentra en la biblioteca, se añade a la misma. Si se emplea, además, el modificador `v`, `ar` imprimirá una línea por cada módulo añadido o reemplazado, especificando el nombre del módulo y las letras `a` o `r`, respectivamente.

  `d` **Borrado**. Elimina un módulo de la biblioteca.

  `x` **Extracción**. Crea el fichero objeto cuyo nombre se especifica y copia su contenido de la biblioteca. La biblioteca queda inalterada.

  Si no se especifica ningún nombre, se extraen todos los ficheros de la biblioteca.

  `t` **Listado**. Proporciona una lista especificando los módulos que componen la biblioteca.

- *modificadores*: Se pueden añadir a las operaciones, modificando de alguna manera el comportamiento por defecto de la operación. Los más importantes (y casi siempre se utilizan) son:

  `s` **Indexación**. Actualiza o crea (si no existía previamente) el índice de los módulos que componen la biblioteca. *Es necesario que la biblioteca tenga un índice para que el enlazador sepa cómo enlazarla*. Este modificador puede emplearse acompañando a una operación o por si solo.

  `v` **Verbose**. Muestra información sobre la operación realizada.

- *biblioteca* es el nombre de la biblioteca a crear o modificar.

- *módulos objeto* es la lista de ficheros objeto que se van a añadir, eliminar, actualizar, etc. en la biblioteca.

Los siguientes ejercicios ayudarán a entender el funcionamiento de las distintas opciones de `ar`.

**Ejercicio**

Repetir los siguientes ejemplos:

1. Crear la biblioteca `libprueba.a` a partir del módulo objeto `opers.o`.
   ```
   ar -rvs lib/libprueba.a obj/opers.o
   ```
2. Mostrar los módulos que la componen.
   ```
   ar -tv lib/libprueba.a
   ```

**Ejercicio**

1. Añadir los módulos `adicion.o` y `producto.o` a la biblioteca `libprueba.a`.
   ```
   ar -rvs lib/libprueba.a obj/adicion.o obj/producto.o
   ```
2. Mostrar los módulos que la componen.
   ```
   ar -tv lib/libprueba.a
   ```

**Ejercicio**

1. Extraer el módulo `adicion.o` de la biblioteca `libprueba.a`.
   ```
   ar -xvs lib/libprueba.a adicion.o
   ```
2. Mostrar los módulos que la componen.
   ```
   ar -tv lib/libprueba.a
   ```
3. Mostrar el directorio actual.

**Ejercicio**

1. Borrar el módulo `producto.o` de la biblioteca `libprueba.a`.
   ```
   ar -dvs lib/libprueba.a producto.o
   ```
2. Mostrar los módulos que la componen.
   ```
   ar -tv lib/libprueba.a
   ```
3. Mostrar el directorio actual.

# g++, make y ar trabajando conjuntamente

Como hemos señalado, ar es un programa general que empaqueta/desempaqueta ficheros en/desde archivos, de manera similar a como hacen otros programas como *zip*, *rar*, *tar*, etc. (una lista amplia puede encontrarse en http://es.wikipedia.org/wiki/Anexo:Archivadores_de_ficheros).

Nuestro interés es aprender cómo puede emplearse una biblioteca para la generación de un ejecutable, y cómo escribir las dependencias en ficheros makefile para poder automatizar todo el proceso de creación/actualización con la orden make.

## Creación de la biblioteca

Emplearemos una regla para la construcción de la biblioteca.

1. El *objetivo* será la biblioteca a construir.
2. La *lista de dependencias* estará formada por los módulos objeto que constituirán la biblioteca.

Para los dos casos estudiados en la página 55, escribiríamos:

1. Caso 1.

```
lib/libejemplo.a : obj/funcs.o
    ar -rvs /libejemplo.a obj/funcs.o
```

2. Caso 2.

```
lib/libejemplo.a : obj/fun01.o obj/fun02.o obj/fun03.o obj/fun04.o\
                   obj/fun05.o obj/fun06.o obj/fun07.o obj/fun08.o\
                   obj/fun09.o obj/fun10.o
    ar -rvs lib/libejemplo.a obj/fun01.o obj/fun02.o obj/fun03.o\
                   obj/fun04.o obj/fun05.o obj/fun06.o obj/fun07.o\
                   obj/fun08.o obj/fun09.o obj/fun10.o
```

(La barra simple invertida (\) es muy útil para dividir en varias líneas órdenes muy largas ya que permite continuar escribiendo en una nueva línea la misma orden)

Evidentemente, resulta aconsejable escribir de manera más compacta y generalizable esta regla:

```
MODS_EJEMPLO = obj/fun01.o obj/fun02.o obj/fun03.o obj/fun04.o\
               obj/fun05.o obj/fun06.o obj/fun07.o obj/fun08.o\
               obj/fun09.o obj/fun10.o
......
lib/libejemplo.a : $(MODS_EJEMPLO)
    ar -rvs lib/libejemplo.a $(MODS_EJEMPLO)
```

## Enlazar con una biblioteca

El enlace lo realiza g++, llamando al enlazador ld. Extenderemos la regla que genera el ejecutable:

1. El *objetivo* es el mismo: generar el ejecutable.
2. La *lista de dependencias* incluirá ahora a la biblioteca que se va a enlazar.
3. La *orden* debe especificar:
   *a*) El directorio dónde buscar la biblioteca (opción -L)
   Recordemos que la opción -L*path* indica al enlazador el directorio donde se encuentran los ficheros de biblioteca. Se puede utilizar la opción -L varias veces para especificar distintos directorios de biblioteca.
   *b*) El nombre (resumido) de la biblioteca (opción -l).
   Los ficheros de biblioteca se proporcionan a g++ de manera resumida, escribiendo -l*nombre* para referirnos al fichero de biblioteca lib*nombre*.a El enlazador busca en los directorios de bibliotecas (entre los que están los especificados con -L) un fichero de biblioteca llamado lib*nombre*.a y lo usa para enlazarlo.

Para los dos casos estudiados en la página 55, escribiríamos:

1. Caso 1.

```
bin/prog_1 : obj/ppal.o lib/libejemplo.a
    g++ -o bin/prog_1 obj/ppal.o -L./lib -lejemplo
```

2. Caso 2.

```
bin/prog_2 : obj/ppal.o lib/libejemplo.a
    g++ -o bin/prog_2 obj/ppal.o -L./lib -lejemplo
```

# Anexo: Ejercicios

Para poder realizar los ejercicios propuestos es necesario disponer de los ficheros:

- `ppal_1.cpp`, `ppal_2.cpp`, `opers.cpp`, `adicion.cpp` y `producto.cpp`
  Usaremos un nuevo fichero fuente, `ppal.cpp`, que será una copia de `ppal_1.cpp` Este fichero se usará en todos los ejercicios, y únicamente cambiará(n) la(s) línea(s) `#include`
- `opers.h`, `adicion.h` y `producto.h`
- `makefile_ppal_1` y `makefile_ppal_2`

En todos los ejercicios debe poder diferenciar claramente los dos actores que pueden intervenir:

1. El creador de la biblioteca
2. El usuario de la bibilioteca

aunque sea la misma persona (en este caso, usted) quien realice las dos tareas. Esta distinción es fundamental cuando se maneje los ficheros de cabecera: puede llegar a manejar dos ficheros exactamente iguales aunque con nombre diferentes. Uno de ellos será empleado por el creador de la biblioteca y una vez construida la biblioteca, proporcionará otro fichero de cabecera que sirva de interface de la biblioteca a los usuarios de la misma.

**Ejercicio**

Utilizar como base `makefile_ppal_1` y construir el fichero makefile llamado `makefile_1lib_1mod` que implementa el diagrama de la figura 20

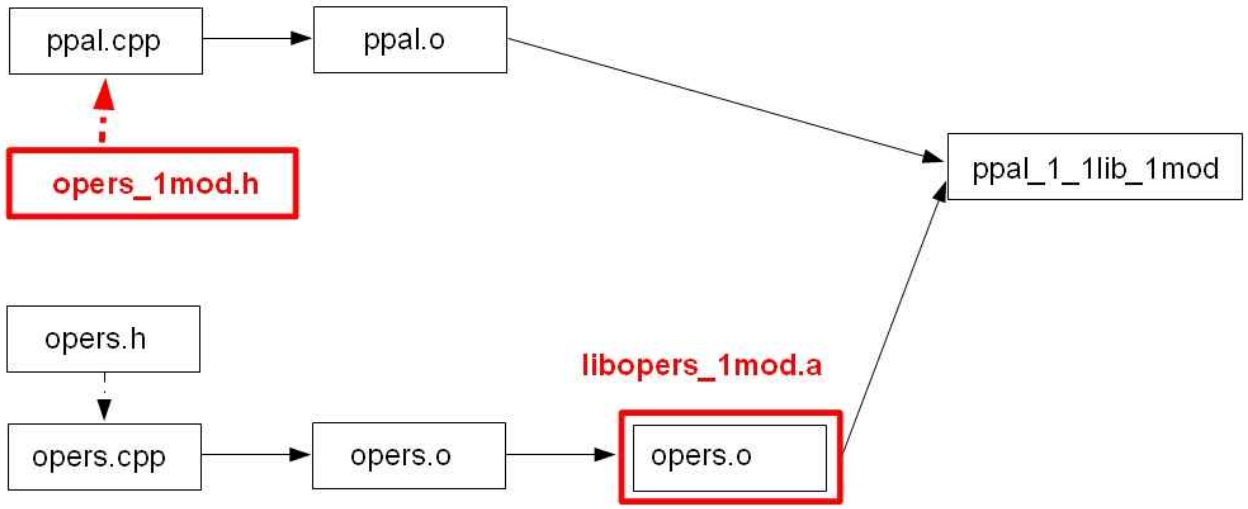



Figura 20: Construcción de un ejecutable que usa funciones de biblioteca (1). Una biblioteca formada por un módulo objeto que implementa cuatro funciones.

En este caso tenemos una biblioteca formada por un único módulo objeto. Todas las funciones están definidas en ese módulo objeto. Observe que el ejecutable será exactamente igual que el que llamamos anteriormente `ppal_1` (ver figura 3 de la práctica 2). Explique por qué.

El fichero de cabecera `opers_1mod.h` asociado a la biblioteca `libopers_1mod.a` podría tener el mismo contenido que el que se emplea para construir la biblioteca.

**Ejercicio**

Utilizar como base `makefile_ppal_2` y construir el fichero makefile llamado `makefile_1lib_2mod` que implementa el diagrama de la figura 21

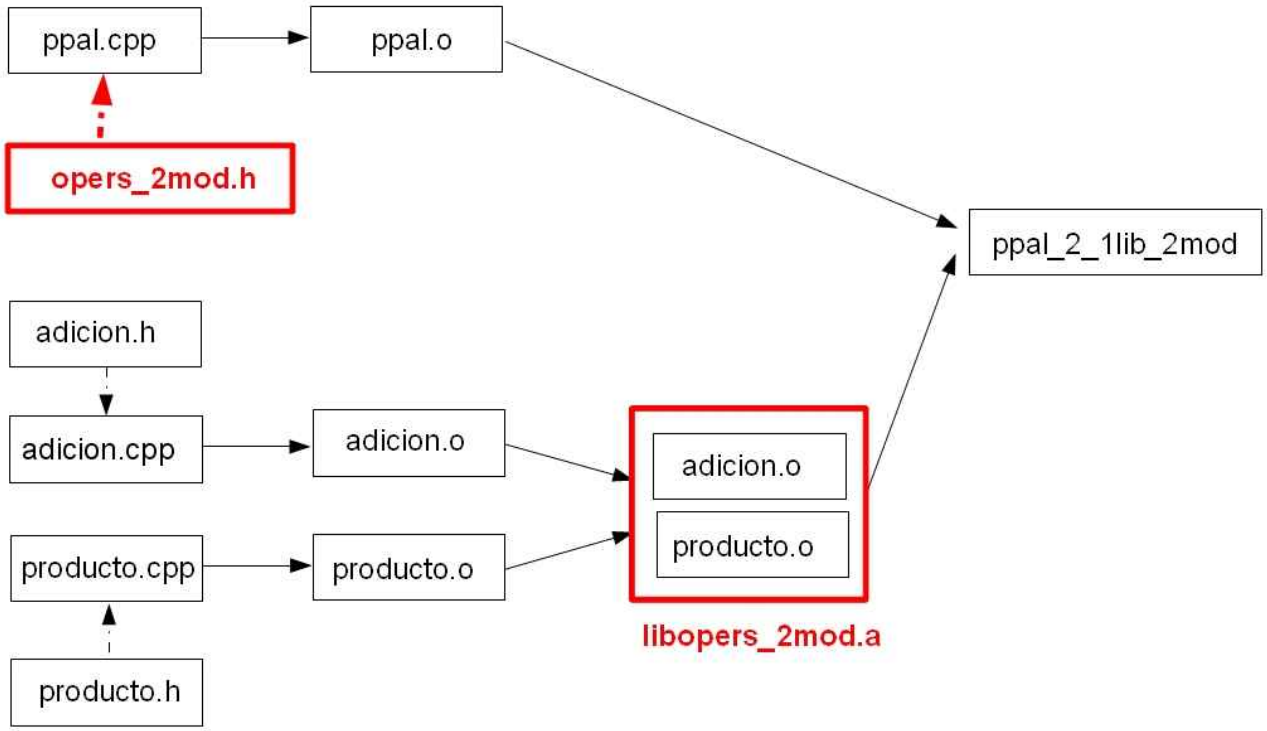



Figura 21: Construcción de un ejecutable que usa funciones de biblioteca (2). Una biblioteca formada por dos módulos objeto que implementan dos funciones cada uno de ellos.

En este caso tenemos una biblioteca formada por dos módulos objeto. Cada uno define dos funciones. Observe que el ejecutable será exactamente igual que el que llamamos anteriormente `ppal_2` (ver figura 4 de la práctica 2). Explique por qué.

El fichero de cabecera `opers_2mod.h` asociado a la biblioteca `libopers_2mod.a` podría tener el mismo contenido que `opers_1mod.h` (ejercicio anterior) si se pretende ofrecer la misma funcionalidad.

**Ejercicio**

Construir el fichero makefile llamado `makefile_1lib_4mod`. El fichero makefile implementa el diagrama de dependencias mostrado en la figura 22 En este caso tenemos una biblioteca formada por cuatro módulos objeto, y cada uno define una única función.

Como trabajo previo deberá crear los ficheros fuente `suma.cpp`, `resta.cpp`, `multiplica.cpp` y `divide.cpp`. Observe que no se han considerado ficheros de cabecera para cada uno de éstos ¿por qué?

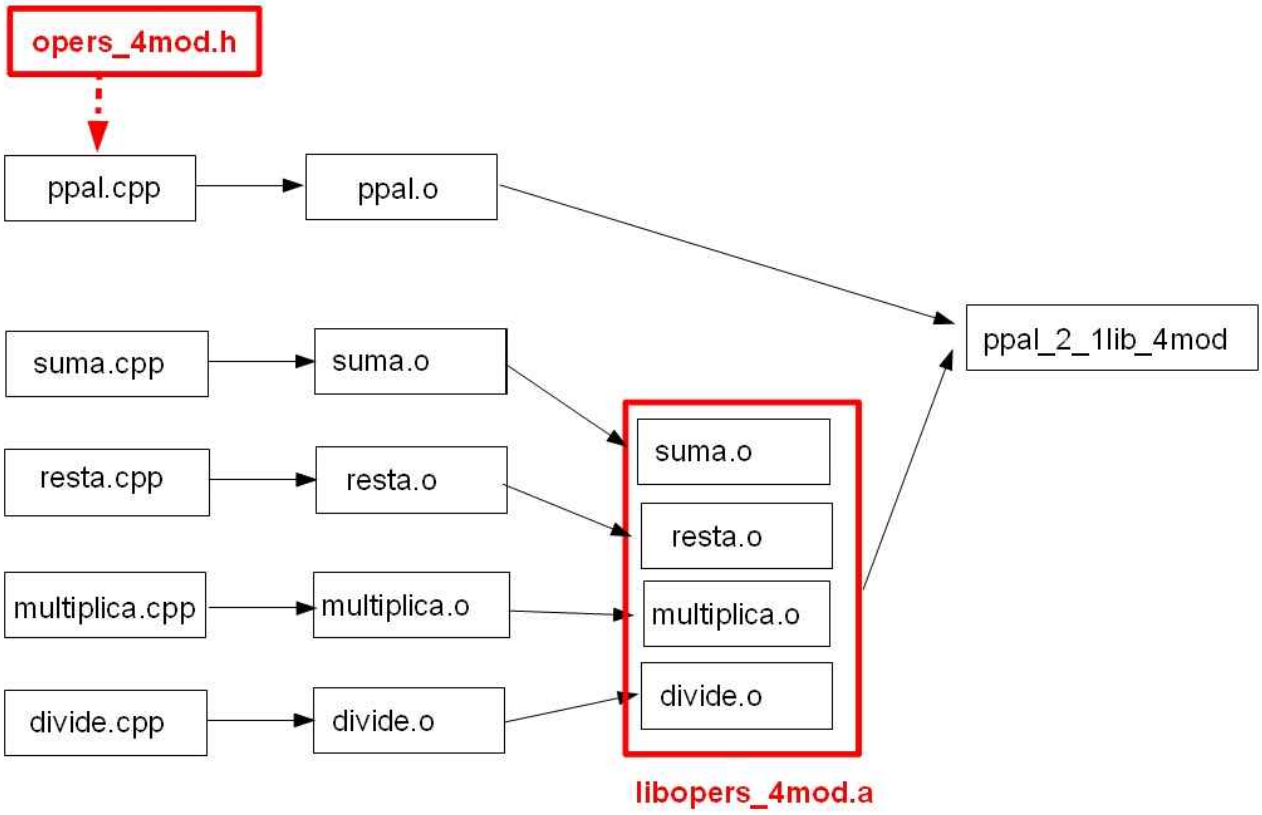



Figura 22: Construcción de un ejecutable que usa funciones de biblioteca (3). Una biblioteca formada por cuatro módulos objeto, y cada uno define una única función.

El fichero de cabecera `opers_4mod.h` asociado a la biblioteca `libopers_4mod.a` podría tener el mismo contenido que `opers_2mod.h` y `opers_1mod.h` (ejercicios anteriores) si se pretende ofrecer la misma funcionalidad.

**Ejercicio**

Construir el fichero makefile llamado `makefile_2lib_2mod`. El fichero makefile implementa el diagrama de dependencias mostrado en la figura 23

Observad que el ejecutable resultante debe ser exactamente igual que el anterior ¿por qué?

En este caso tenemos dos bibliotecas (`libadic_2mod.a` y `libproducto_2mod.a`) con sus ficheros de cabecera asociados (`adic_2mod.h` y `producto_2mod.h`). Cada una de las bibliotecas está compuesta de dos módulos objeto, y cada uno define dos funciones.

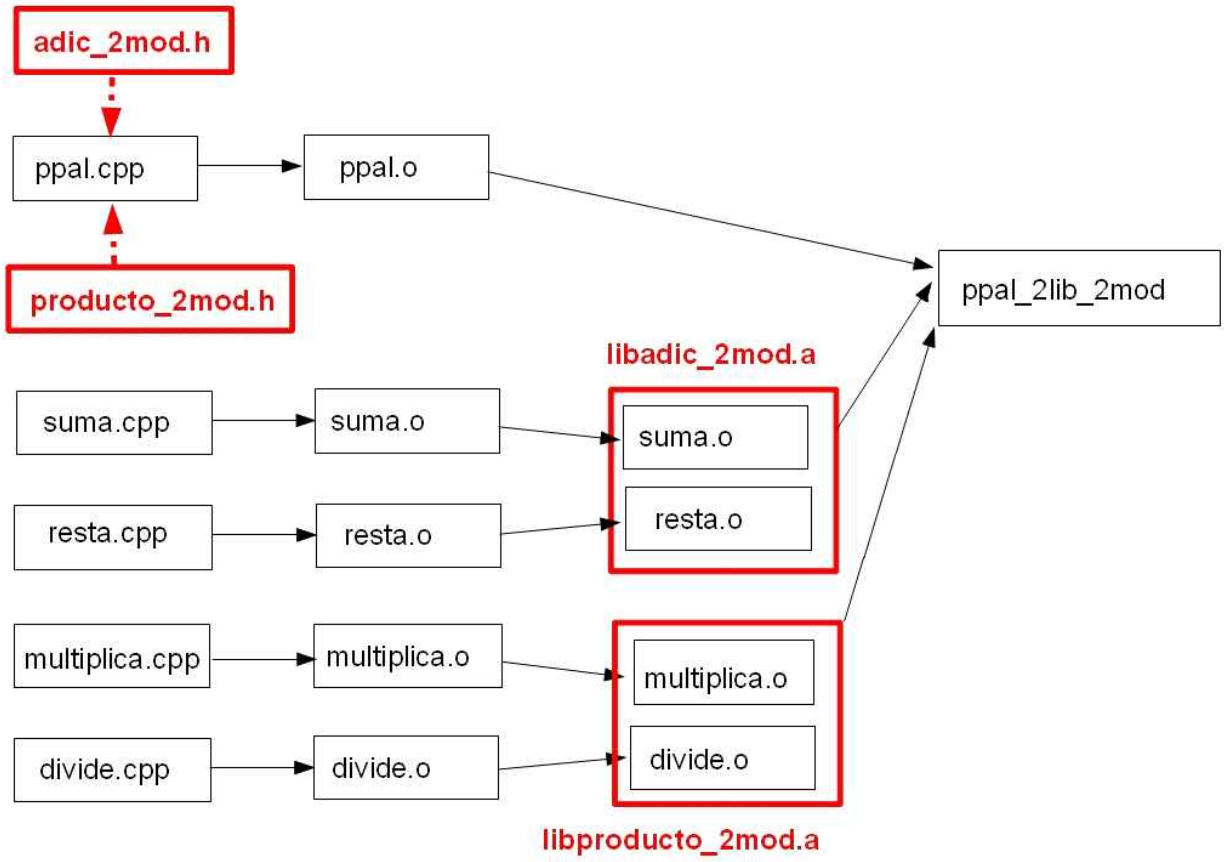



Figura 23: Construcción de un ejecutable que usa funciones de biblioteca (4). Dos bibliotecas formadas cada una por dos módulos objeto, y cada uno define dos funciones.

**Ejercicio**

1. Tomar nota de los tamaños de los ejecutables `ppal_1lib_1mod`, `ppal_1lib_2mod`, `ppal_1lib_4mod` y `ppal_2lib_2mod`.
2. Editar el fichero `ppal.cpp` y comentar la línea que llama a la función `suma()`.
3. Volver a construir los ejecutables `ppal_1lib_1mod`, `ppal_1lib_2mod`, `ppal_1lib_4mod` y `ppal_2lib_2mod`.
4. Anotar los tamaños de los ejecutables, compararlos con los anteriores y discutir.