Introducción a la Compilación de Programas en C++

Metodología de la Programación II

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Objetivo de este documento

El objetivo de este documento no es el de ser exhaustivos en la materia, sino introducir los conceptos esenciales que serán necesarios para el desarrollo de la asignatura. En cualquier caso, para obtener información más detallada, se recomienda consultar la lista de libros seleccionados de la biblioteca de la Universidad de Granada que se incluye en la Sección 9.

Índice

1. El modelo de compilación en C++ **3**
1.1. Introducción . . . 3
1.2. El Preprocesador . . . 4
1.3. El compilador . . . 4
1.4. El enlazador . . . 5
1.5. Uso de bibliotecas . . . 6

2. La modularización del software en C++ **7**
2.1. Introducción . . . 7
2.2. Ventajas de la modularización del software . . . 7
2.3. Cómo dividir un programa en varios ficheros . . . 7
2.4. Organización de los ficheros fuente . . . 8
2.5. Ocultamiento de información . . . 8
2.5.1. Un fichero de cabecera para nuestras bibliotecas . . . 9
2.6. Compilación de programas compuestos por varios ficheros fuente. Compilación separada 9

3. El preprocesador de C++ **12**
3.1. Introducción . . . 12
3.2. Creación de constantes simbólicas y macros funcionales . . . 12
3.2.1. Constantes simbólicas . . . 12
3.2.2. Macros funcionales (con argumentos) . . . 14
3.3. Eliminación de constantes simbólicas . . . 15
3.4. Inclusión de ficheros . . . 15
3.5. Inclusión condicional de código . . . 17

4. El compilador *g++* **18**
4.1. Introducción . . . 18
4.2. Sintaxis . . . 19
4.3. Opciones más importantes . . . 19
4.4. Ejemplos . . . 20

5. Introducción al depurador DDD **23**
5.1. Conceptos básicos . . . 23
5.2. Pantalla principal . . . 24
5.3. Opciones fundamentales del menú del depurador . . . 24
5.4. Ejecución de un programa paso a paso . . . 28
5.5. Inspección y modificación de datos . . . 28
5.6. Inspección de la pila . . . 29
5.7. Mantenimiento de sesiones de depuración . . . 29
5.8. Reparación del código . . . 29

6. El programa *make* y la construcción de ficheros *makefile* **30**
6.1. Introducción . . . 30
6.2. El programa *make* . . . 31
6.2.1. Sintaxis . . . 31
6.2.2. Opciones más importantes . . . 31
6.2.3. Funcionamiento de *make* . . . 31
6.3. Ficheros *makefile* . . . 32
6.3.1. Comentarios . . . 32
6.3.2. Reglas. Reglas explícitas . . . 33
Reglas explícitas . . . 33
6.3.3. Órdenes . . . 34
Prefijos de órdenes . . . 34

6.3.4. Destinos Simbólicos 36
Macros predefinidas 38
Destinos .PHONY 38
6.4. Ejemplo: Gestión de un proyecto software 38
6.4.1. Versión 1 39
6.4.2. Versión 2 39
6.4.3. Versión 3 40
6.5. Macros, reglas implícitas y directivas condicionales 41
6.5.1. Macros en ficheros *makefile* 41
Sustituciones de cadenas en macros 43
Macros en llamadas a *make* 44
6.5.2. Reglas implícitas 46
Reglas implícitas patrón 49
Reglas patrón estáticas 51
6.5.3. Directivas condicionales en ficheros *makefile* 52

7. Uso y construcción de bibliotecas con el programa *ar* **56**
7.1. Introducción 56
7.2. Estructura de una biblioteca 57
7.3. Gestión de bibliotecas con el programa *ar* 60
7.4. Ejemplos 60
7.4.1. Ejemplos básicos 61
7.4.2. Ejemplos avanzados 69
7.4.3. Ejemplos para expertos 74

8. Código **78**
8.1. Código de ejemplo 8.1 78
8.1.1. Un programa para depurar muy simple 78
8.2. Código de ejemplo 8.2 79
8.2.1. Un makefile muy simple 79
8.3. Código de ejemplo 8.3 80
8.3.1. Fichero calculos.cpp 80
8.4. Código de ejemplo 8.4 82
8.4.1. Fichero ordena1.cpp 82
8.4.2. makefile 83
8.5. Código de ejemplo 8.5 83
8.5.1. Fichero ppal.cpp 83
8.5.2. Fichero funcsvec.cpp 84
8.5.3. Fichero funcsvec.h 85
8.5.4. makefile 86
8.6. Código de ejemplo 8.6 86
8.6.1. Fichero ppal.cpp 86
8.6.2. Fichero vec_ES.cpp 87
8.6.3. Fichero vec_ES.h 87
8.6.4. Fichero ordena.cpp 88
8.6.5. Fichero ordena.h 88
8.6.6. makefile 88

9. Más información **89**
9.1. Documentación dentro del SO 89
9.2. Documentación en Internet 89

1. El modelo de compilación en C++

1.1. Introducción

En la figura 1 mostramos el esquema básico del proceso de compilación de programas, módulos 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.

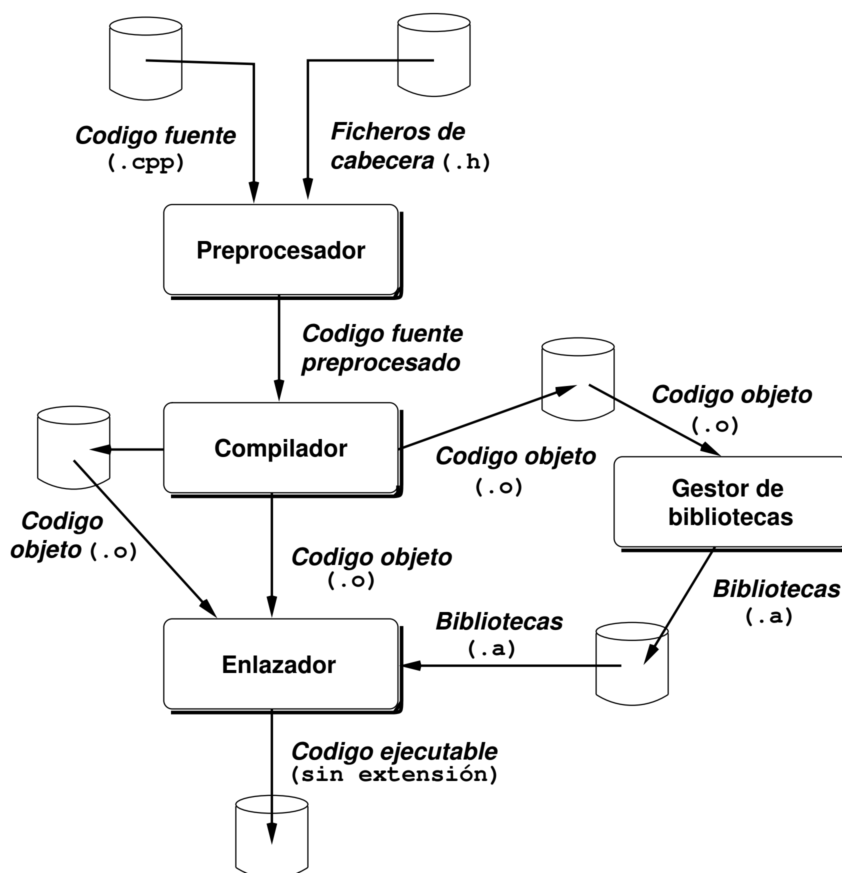


Figura 1: Esquema del proceso de compilación (generación de programas ejecutables) en C++

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 por el procesador. 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 biblioteca.

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 estos ficheros de biblioteca también está contemplada en el esquema. Así, es posible generar únicamente ficheros objeto para:

1. Enlazarlos con otros ficheros objeto para generar un ejecutable.

Esta primera alternativa 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, ya que normalmente no tiene interés su permanencia. Si realmente se desean conservar algunos de los ficheros objeto se opta por la siguiente alternativa.

2. Incorporarlos a una biblioteca (del inglés *library*).

   Una biblioteca, en la terminología de C++, será 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()`.

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++.

1.2. El Preprocesador

El preprocesador acepta como entrada código fuente y se encarga de:

1. Eliminar los comentarios.
2. Interpretar y procesar las directivas de preprocesamiento, precedidas *siempre* por el símbolo `#`.

Dos de las directivas más comúnmente empleadas en C++ son `#include` y `#define`:

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

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

- `#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 aparición de `MAX_SIZE` por el literal 100 (de tipo `int`).

En la sección 3 tratamos con más profundidad estas directivas y algunas otras más complejas. En cualquier caso, retomando el esquema mostrado en la figura 1 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 2 mostramos el esquema de la fase de compilación. Este tema lo desarrollamos en las secciones 1.3 y 4, donde describimos el funcionamiento del compilador de C++ implementado por GNU (`g++`).

1.3. El compilador

El compilador (del inglés, *compiler*) analiza la sintaxis y la semántica del código fuente preprocesado y lo traduce, generando un fichero que contiene el código objeto. La extensión por defecto de estos ficheros es `.o` (GNU) o `.OBJ` (Borland C).

El programa proporcionado por GNU para realizar la compilación es `gcc` (las siglas `gcc` son el acrónimo de ***G****NU* ***C*** ***C****ompiler*) para el caso del lenguaje de programación C o `g++` para el caso del lenguaje C++. En el resto de este documento las indicaciones se harán con respecto a `g++` o para `gcc`

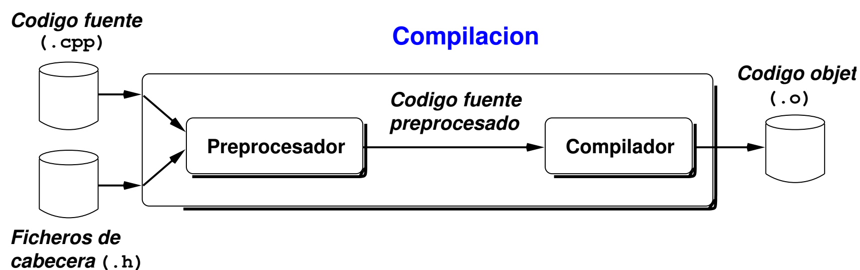


Figura 2: Fase de compilación

indistintamente, entendiendo que son igualmente válidas para ambos compiladores. En la sección 4 discutimos con detalle el funcionamiento y las opciones más relevantes de `g++`.

Si bien, como veremos en la siguiente sección, el comportamiento por defecto de `g++` es crear un ejecutable, esto es, compilar y enlazar invocando de forma *impícita* al programa enlazador, este programa puede usarse para realizar únicamente la compilación (construcción de módulos objeto) si en su llamada se especifica el parámetro `-c`. Por ejemplo, dado un módulo fuente llamado `prueba.cpp`, el correspondiente módulo objeto se construye ejecutando:

```
g++ -c -o prueba.o prueba.cpp
```

Antes de continuar, debemos hacer una importante observación. En el proceso de compilación se traduce el código fuente del fichero que se está compilando salvo las referencias a identificadores externos (funciones o variables). Estos objetos externos, especialmente funciones, se declaran en este fichero, pero se definen en otros. La declaración servirá al compilador para comprobar que las referencias externas son sintácticamente correctas.

Veamos con un ejemplo muy simple e ilustrativo de que esta situación la encontramos muy a menudo. Supongamos un fichero `.cpp` que contiene la función `main()`. En ella se usa una instrucción `sin()`. Para comprobar si la sintaxis es correcta, el compilador necesita conocer el prototipo de esta función: de ahí que nuestro programa necesite incluir el fichero de cabecera `cmath`. Sin embargo, el código objeto asociado a esta función se encuentra en otro fichero (una biblioteca del sistema) por lo que la traducción de nuestro programa se hará dejando el "hueco" correspondiente a la función `sin()`. Este hueco lo completará el enlazador a partir de una biblioteca del sistema, generando definitivamente un código ejecutable.

1.4. El enlazador

El enlazador (del inglés, *linker*) resuelve las referencias a objetos externos. Estas referencias son a objetos que se encuentran en otros módulos compilados, ya sea en forma de ficheros objeto o incorporados en alguna biblioteca.

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 el compilador `g++`: cuando se llama a `g++` y uno de los ficheros involucrados es un módulo objeto o un fichero de biblioteca, siendo el último paso que realiza `g++` la llamada al enlazador `ld`. 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.

Para generar un fichero ejecutable se requiere que uno de los módulos objeto que se están enlazando contenga la función `main()`. A modo de ejemplo, supongamos un fichero llamado `principal.cpp` que contiene la función `main()`. En este fichero se hace uso de la función que calcula el seno, cuyo prototipo (declarado en `cmath`) es el siguiente: `double sin(double x)`. Se quiere generar un fichero ejecutable a partir de este módulo fuente.

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
  ...
int main ()
{
      ...
   double s, x;
      ...
   cin >> x;
      ...
   s = sin (x);
   cout << "El seno de " << x << " es " << s;
      ...
}
```

Cuando el compilador procesa este código no puede generar el código ejecutable ya que no puede completar la referencia a la función `sin()`: lo único que encuentra en `math.h` es el prototipo o declaración de la función `sin()`. Así, el código ejecutable asociado a este cálculo no puede completarse y queda "pendiente". Puede comprobarse empíricamente. La ejecución del comando

```
g++ -o principal principal.cpp
```

produce un error de enlace ya que el enlazador no encuentra el código asociado a la función `sin()`. Para que se genere el código ejecutable habrá que indicar *explícitamente* que se enlace con la biblioteca matemática:

```
g++ -o principal principal.cpp -lm
```

1.5. Uso 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, gestión dinámica de memoria, 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, como indicamos anteriormente, algunas bibliotecas se enlazan automáticamente al generar un programa ejecutable, lo que induce al error de pensar que alguna función es propia 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 ISO C++* debe asegurar que las funciones proporcionadas en esas bibliotecas se comportan de forma similar a como especifica el comité ISO. A efectos prácticos, las funciones de la biblioteca estándar pueden considerarse parte del lenguaje C++.

Aún teniendo en cuenta estas consideraciones, 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 de una forma completamente estandarizada. En la figura 1 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.

El programa de GNU que se encarga de incorporar, eliminar y sustituir módulos objeto en una biblioteca es `ar`. En realidad, `ar` es un programa de aplicación más general que la de gestionar bibliotecas de módulos objeto pero no entraremos en ello ya que no es de nuestro interés. En la sección 7 explicamos

con detalle la forma en que se gestionan y enlazan bibliotecas, profundizando en el funcionamiento del programa `ar`.

2. La modularización del software en C++

2.1. 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 hace es 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 los módulos objeto correspondientes y/o las bibliotecas necesarias.

2.2. 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, que 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` (sección 6) nos será muy útil para esta tarea.
5. En lenguajes como Pascal, empleamos el anidamiento de funciones y las unidades para conseguir el ocultamiento de información, herramienta indispensable para conseguir la abstracción. En C++, las funciones no se pueden anidar. La única herramienta de la que disponemos es 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 interfaz necesaria para poder usar estas funciones y objetos.

2.3. 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 implementarán 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.

2.4. 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`, constantes globales, 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 identificador 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 incluye una linea de declaración (prototipo) al principio del fichero, con lo que la definición se puede situar en cualquier punto del fichero, incluso después de las llamadas a la función.
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 fichero que contiene la llamada

   una línea de declaración (prototipo) de la función. Normalmente se realiza incluyendo (mediante la directiva de preprocesamiento `#include`) el fichero de cabecera asociado al módulo que contiene la definición de la función.

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

2.5. Ocultamiento de información

Como sabemos, el ocultamiento de información es un objetivo fundamental en la Metodología de la Programación, ya que nos proporciona un gran número de ventajas. Como hemos comentado anteriormente, la modularización del software es la herramienta de que disponemos cuando programamos en C++ para conseguir el ocultamiento.

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.

2.5.1. Un fichero de cabecera para nuestras bibliotecas

El uso que hemos comentado de los ficheros de cabecera está orientado a la compilación de nuestro software. Pero también podemos dar o vender nuestro software a otros programadores en forma de módulos objeto o, casi siempre, en forma de bibliotecas. Por motivos de privacidad o, simplemente, para dar al usuario sólo la información que le interesa, podemos crear un fichero de cabecera con este fin. En este fichero de cabecera sólo incluiremos los objetos y funciones que el usuario puede llamar (públicas), ocultándole todas las funciones auxiliares (privadas) empleadas en la biblioteca.

De esta forma, conseguimos un segundo nivel de ocultamiento de información que abre una posibilidad de ceder o vender nuestro software, no como productos finales (ejecutables), sino también como productos para programadores, de la misma forma que el creador de nuestro compilador nos proporciona bibliotecas adicionales a la biblioteca estándar de C++.

2.6. Compilación de programas compuestos por varios ficheros fuente. Compilación separada

Los ejemplos que vamos a citar hacen uso del compilador en línea de GNU, `g++`. En general, todos los compiladores en línea tienen las mismas opciones (al menos las más importantes), aunque probablemente cambie la forma de invocarlas en la línea de órdenes. En la sección 4 discutimos con detalle el funcionamiento y las opciones más relevantes de `g++`.

En general, el proceso de compilación se vuelve algo más complicado cuando un programa está compuesto por varios ficheros. Por ejemplo, supongamos un programa que está compuesto por tres ficheros: `programa.cpp`, que contiene la función `main()` y los ficheros `func1.cpp` y `func2.cpp` que contienen funciones que necesita nuestro programa. La forma más simple de generar un ejecutable llamado `programa` a partir de estos ficheros es la siguiente:

```
g++ -o programa programa.cpp func1.cpp func2.cpp
```

En este ejemplo, `g++` se encarga de llamar al preprocesador, de compilar cada uno de los ficheros fuente y de llamar al enlazador para que enlace los ficheros objeto resultantes del proceso de compilación, generando el ejecutable `programa`. En este caso, los ficheros objeto *no* se conservan después del enlace, constituyen ficheros temporales.

Podemos también compilar cada fichero fuente por separado utilizando la opción `-c`. Esta opción hace que se ejecute únicamente el compilador, generando un fichero objeto y sin llamar al enlazador. Si no se especifica el nombre del fichero objeto destino, éste tendrá el mismo nombre que el fichero fuente, pero con la extensión `.o`. En este ejemplo:

```
g++ -c programa.cpp
g++ -c func1.cpp
g++ -c func2.cpp
```

Ahora, sólo falta enlazarlos para generar el programa ejecutable. Usaremos `g++` para que éste llame implícitamente al enlazador:

```
g++ -o programa programa.o func1.o func2.o
```

Siguiendo este proceso, además del ejecutable tendremos los ficheros `programa.o`, `func1.o` y `func2.o`. Los dos últimos podrían usarse para incorporarse a una biblioteca.

Finalmente, mostraremos con un ejemplo más complejo cómo construir un programa ejecutable a partir de ficheros objeto y bibliotecas, y cómo construir éstas a partir de una serie de ficheros con código fuente. Este programa se construye a partir de los siguientes ficheros:

1. `programa.cpp`: Contiene la función `main()` y, posiblemente, otras funciones del programa.
2. `funcs1.cpp` y `funcs2.cpp`: Contienen funciones y/o objetos requeridos en el programa. No tenemos interés en que formen parte de alguna biblioteca.
3. Por un lado, `liba1.cpp`, `liba2.cpp` y `liba3.cpp`. Por otro, `libb1.cpp` y `libb2.cpp`. Contienen funciones relacionadas y generalizadas para su uso por cualquier otro programa. Formarán las bibliotecas `liba.a` y `libb.a`.

El esquema del proceso que debe seguirse para generar el ejecutable `programa` a partir de estos ficheros se describe en la figura 3.

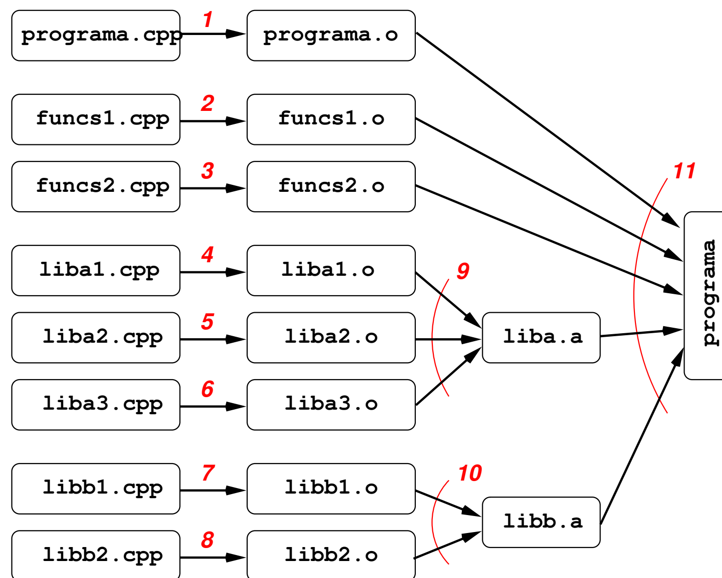


Figura 3: Esquema completo del proceso de generación del ejecutable

Sobre la figura 3 hemos numerado las acciones a realizar. En primer lugar, se generan los módulos objeto correspondientes a cada uno de los módulos fuente. El orden de estas ocho acciones no es importante, pudiendose intercambiar sin problemas:

```
1. g++ -c -o programa.o programa.cpp
2. g++ -c -o funcs1.o funcs1.cpp
3. g++ -c -o funcs2.o funcs2.cpp
4. g++ -c -o liba1.o liba1.cpp
5. g++ -c -o liba2.o liba2.cpp
6. g++ -c -o liba3.o liba3.cpp
7. g++ -c -o libb1.o libb1.cpp
8. g++ -c -o libb2.o libb2.cpp
```

Antes de generar el ejecutable deben estar creadas las dos bibliotecas, por lo que es el momento de crearlas y se puede hacer ahora porque los módulos objeto que las componen ya se han creado. El orden de creación entre ellas es irrelevante.

```
 9. ar -rvs liba.a liba1.o liba2.o liba3.o
10. ar -rvs libb.a libb1.o libb2.o
```

Las bibliotecas se crean con el programa `ar`, el gestor de bibliotecas de GNU. En la sección 7 discutimos con detalle el funcionamiento y las opciones más relevantes de `ar`. Una vez creadas las bibliotecas ya puede crearse el ejecutable, enlazando `programa.o` (que contiene la función `main()`) con los módulos objeto `funcs1.o` y `funcs2.o` y con las bibliotecas `liba.a` y `libb.a`.

```
11. g++ -o programa programa.o funcs1.o funcs2.o liba.a libb.a
```

Con la ayuda de un esquema como el mostrado en la figura 3 puede construirse cualquier proyecto complejo. Sin embargo, una modificación en un módulo del proyecto puede hacer que el proceso de actualización sea muy tedioso. Por ejemplo, si se detecta un fallo en una función de `liba1.cpp` para actualizar correctamente, la biblioteca en la que está el módulo objeto asociado y el ejecutable que usa la función modificada será necesario repetir los pasos 4, 9 y 11 de la figura 3.
Las acciones a realizar, en este orden, son:

```
 4. g++ -c -o liba1.o liba1.cpp
 9. ar -rvs liba.a liba1.o liba2.o liba3.o
11. g++ -o programa programa.o funcs1.o funcs2.o liba.a libb.a
```

Resulta evidente que es necesaria mucha disciplina y ser muy metódico para que, en un caso real, gestionar adecuadamente las acciones a realizar y el orden en que se deben acometer. Afortunadamente, existe una utilidad (`make`) que gestiona todo esto: se encarga de comprobar qué ficheros se han modificado y de realizar las acciones oportunas para que todo lo que depende de estos ficheros se reconstruya ordenadamente de forma que se incorporen las modificaciones introducidas. El programador tan sólo debe especificar, en un formato adecuado, las dependencias entre ficheros y las acciones que deben ejecutarse si se realiza alguna modificación. Esta especificación se realiza mediante un fichero especial denominado genericamente *makefile*. En la sección 6 describimos con detalle la utilidad `make` y el formato de los ficheros makefile.

Para finalizar presentaremos un fichero makefile asociado al proyecto que nos ocupa. Ésta es la primera aproximación, por lo que el fichero makefile es muy explícito. Se deja al lector como ejercicio que una vez haya estudiado la sección 6 reescriba este fichero de forma más compacta.

```
# Fichero: makefile
# Fichero makefile para la construccion del ejecutable "programa".

programa: programa.o funcs1.o funcs2.o liba.a libb.a
   g++ -o programa programa.o funcs1.o funcs2.o liba.a libb.a

funcs1.o: funcs1.cpp
   g++ -c -o funcs1.o funcs1.cpp
funcs2.o: funcs2.cpp
   g++ -c -o funcs2.o funcs2.cpp

liba.a: liba1.o liba2.o liba3.o
   ar -rvs liba.a liba1.o liba2.o liba3.o
libb.a: libb1.o libb2.o
   ar -rvs libb.a libb1.o libb2.o

liba1.o: liba1.cpp
   g++ -c -o liba1.o liba1.cpp
liba2.o: liba2.cpp
   g++ -c -o liba2.o liba2.cpp
liba3.o: liba3.cpp
   g++ -c -o liba3.o liba3.cpp

libb1.o: libb1.cpp
   g++ -c -o libb1.o libb1.cpp
libb2.o: libb2.cpp
   g++ -c -o libb2.o libb2.cpp
```

La sintaxis de este fichero es muy sencilla. Por ejemplo, las líneas

```
liba1.o: liba1.cpp
  g++ -c -o liba1.o liba1.cpp
```

se interpretan como sigue: `liba1.o` *depende de* `liba1.cpp`. Si se modificara `liba1.cpp` debe reconstruirse `liba1.o`. La forma de hacerlo se especifica en la línea que sigue a la especificación de la dependencia: `g++ -c -o liba1.o liba1.cpp`

Ahora, si se modificara ese módulo bastará con ejecutar `make`. Detectará que `liba1.cpp` es más actual que `liba1.o` (comparando la fecha y hora asociada a ambos) y se reconstruirá `liba1.o`. Después, encuentra que debe reconstruir `liba.a` y finalmente, reconstruye el nuevo ejecutable `programa`. Es indudable que esta forma de gestionar proyectos de software resulta muy atractiva.

3. El preprocesador de C++

3.1. Introducción

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 (figura 4). El preprocesador 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 `#`.

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 esta sección 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.

3.2. Creación de constantes simbólicas y macros funcionales

3.2.1. Constantes simbólicas

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

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

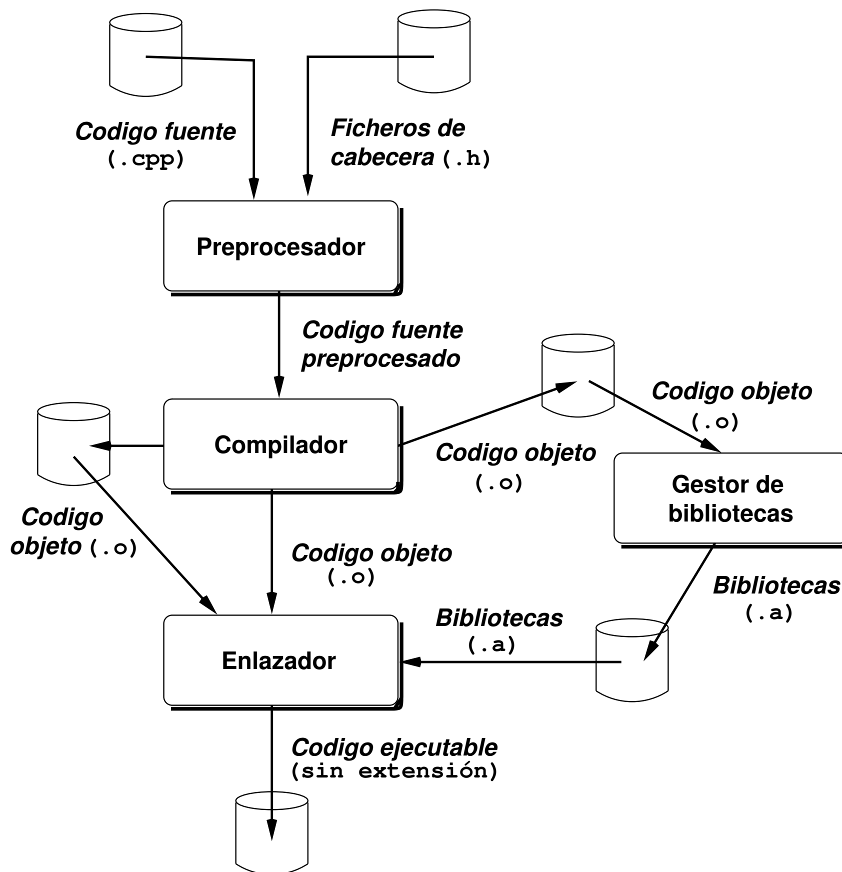


Figura 4: El preprocesamiento constituye la primera etapa de la compilación

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 o se eliminen explícitamente (ver directiva `#undef` en la sección 3.3.
Veamos algunos ejemplos sencillos.

```
#define TAMMAX 256
```

hace que sustituya todas las apariciones del identificador `TAMMAX` por la constante numérica (entera) `256`. Normalmente se prefiere usar una constante entera, definida con `const int` a las constantes simbólicas.

```
#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 sección 3.5).

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

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

3.2.2. 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 ya que no se produce una llamada.
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 pero un gran inconveniente porque no se realiza comprobación de tipos.

Normalmente preferiremos usar las funciones en línea (`inline`) a las macros fucnionales.

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 evalúa 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.

Para finalizar, dos variantes sobre esta directiva:

1. Si se precede el nombre de un argumento por el símbolo `#`, la expansión se incluye entre comillas. Por ejemplo, la expansión de la macro

   ```
   #define dprint(expr) cout << #expr << " = " <<expr <<
   ```

   cuando escribimos la sentencia:

   ```
   dprint(x/y);
   ```

es la siguiente:

```
cout << "x/y" << " = " << x/y;
```

2. Podemos utilizar el símbolo `##` para concatenar los argumentos de una macro funcional en su expansión:

```
#define pega(A,B) A##B
```

   La expansión de la expresión `pega(nombre,1)` es `nombre1`

3.3. 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`.

3.4. 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 sección 4), mientras que "*fichero*" indica que se encuentra en el directorio donde se esta realizando la compilación o en el directorio especificado en el nombre del fichero.

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 en los directorios de ficheros de cabecera del sistema.

Ejemplo 1 *El fichero* `ppal.cpp` *contiene dos directivas* `#include` *y una* `#define`*:*

```
/*********************************************************/
// Fichero : ppal.cpp
/*********************************************************/

#include <iostream>
#include "cabecera.h"
using namespace std;

#define TAMMAX 256

int main ()
{
   float notas[TAMMAX];  // vector de calificaciones
   int   nota1, nota2;   // notas de los parciales
   int   mayor_nota;     // calificacion mayor
```

```
      ......
      cin >> nota1;
      cin >> nota2;
         ......
      mayor_nota = MAX(nota1,nota2);
         ......
   imprime_resumen (notas, TAMMAX);
      ......
}
```

En este fichero no está especificado como resolver `MAX` *ni el prototipo de* `imprime_resumen()`*. Sin embargo, la línea* `#include "cabecera.h"` *puede darnos la pista de que están definidas en él:*

```
/***********************************************************/
// Fichero : cabecera.h
/***********************************************************/

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

/***********************************************************/
// Funcion: imprime_resumen
         ......
/***********************************************************/
void imprime_resumen (int *v, int num_elems);
      ......
```

Cuando se copila el fichero `ppal.cpp`*, el primer paso es el preprocesamiento de* `ppal.cpp`*. En primera instancia, se incluyen los ficheros especificados y se eliminan los comentarios, generando este código:*

```
(Contenido de iostream)
          ......
#define MAX(A,B) ((A)>(B)?(A):(B))
void imprime_resumen (int *v, int num_elems);
using namespace std;

#define TAMMAX 256

int main ()
{
   float notas[TAMMAX];
   int   nota1, nota2;
   int   mayor_nota;
      ......
      cin >> nota1;
      cin >> nota2;
         ......
      mayor_nota = MAX(nota1,nota2);
         ......
   imprime_resumen (notas, TAMMAX);
      ......
}
```

y a continuación se procesan las directivas `#define`*:*

```
(prototipos de iostream)
```

```
        ......
void imprime_resumen (int *v, int num_elems);
using namespace std;

int main ()
{
   float notas[256];
   int   nota1, nota2;
   int   mayor_nota;
      ......
      cin >> nota1;  // lectura de nota 1
      cin >> nota2;  // lectura de nota 2
         ......
      mayor_nota = ((nota1)>(nota2)?(nota1):(nota2));
         ......
   imprime_resumen (notas, 256);
      ......
}
```

y este es el código preprocesado que es usado por el compilador para generar el código objeto.

3.5. 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 sección 4).

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 que:

```
#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 (NOMBRE_FICHERO_CABECERA)
#define NOMBRE_FICHERO_CABECERA

    Resto del contenido del fichero de cabecera

#endif
```

En este ejemplo, la constante simbólica `NOMBRE_FICHERO_CABECERA` 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 `NOMBRE_FICHERO_CABECERA` no está definida, por lo que la evaluación de `#ifndef (NOMBRE_FICHERO_CABECERA)` es cierta, se define y se procesa (incluye) el resto del fichero de cabecera. Cuando se intenta incluir de nuevo, como `NOMBRE_FICHERO_CABECERA` ya está definida la evaluación de `#ifndef (NOMBRE_FICHERO_CABECERA)` 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.

4. El compilador *g++*

4.1. Introducción

La compilación tiene como objetivo analizar la sintaxis y la semántica del código fuente preprocesado y traducirlo a código objeto. Formalmente, el preprocesamiento es la primera fase de la compilación, aunque comúnmente se conoce como compilación al análisis y generación de código objeto (ver figura 5).

Es posible realizar la compilación de un programa C++ desde un entorno de desarrollo integrado (IDE) como los que proporcionan los compiladores de Borland o Dev-C++ o RHIDE de DJGPP, o bien empleando un compilador desde la línea de órdenes.

La compilación desde un entorno integrado resulta, en principio, más cómoda. Pero, ¿qué ocurre cuando queramos compilar en un sistema que no sea un PC y no tengamos instalado un entorno de este tipo? De hecho, la forma usual de compilar programas C++ no es a través de un entorno de desarrollo, sino empleando un compilador desde la línea de órdenes (en última instancia, la compilación desde un entorno de desarrollo invoca la ejecución de este tipo de compilador).

Este tipo de compiladores sí están disponibles en prácticamente todos los sistemas. Puesto que nuestro objetivo en todo momento es buscar la solución más general y estandarizada, vamos a estudiar cómo compilar desde la línea de órdenes empleando el compilador `g++`, desarrollado por GNU.

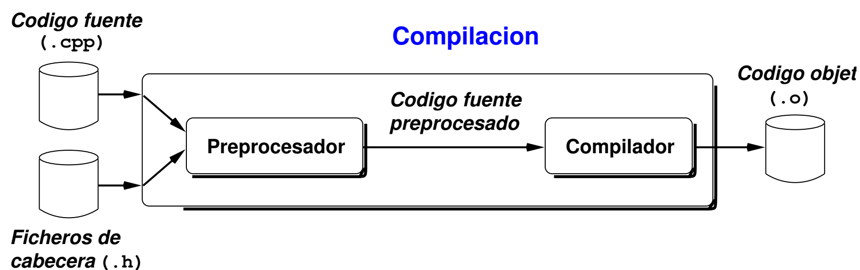


Figura 5: La compilación como generación de código objeto

4.2. Sintaxis

La sintaxis de la llamada al compilador desde la línea órdenes de GNU es la siguiente:

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

donde:

- Cada **opción** va precedida por el signo menos (-) Algunas opciones **no** están acompañadas de argumentos (por ejemplo, `-c` o `-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 4.3.

- *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), `.cc` y `.cpp` (código fuente C++).

  Por defecto, el compilador realizará diferentes etapas del proceso de compilación 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.

4.3. Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

`-ansi` considera únicamente código fuente escrito en C/C++ estándar (ISO C89 y ISO C++) 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.

`-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 ejecutable (su tendencia es la de realizar el trabajo completo: compilar y enlazar) y le asignará un nombre por defecto: `a.exe` (MS-DOS) o `a.out` (Linux/Unix). Si se especifica la opción `-c` generará el fichero objeto *nombre_fichero*`.o`

`-I` *camino* añade a la lista de directorios donde se buscan los ficheros de cabecera. Se puede utilizar esta opción varias veces para añadirr distintos directorios.

`-L` *camino* añade el directorio especificado a la lista de directorios 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. Los ficheros de biblioteca deben especificarse mediante la opción `-l`*fichero*.

`-l` fichero hace que el enlazador busque en los directorios de bibliotecas (en los que están los especificados con `-L` y los del sistema) un fichero de biblioteca llamado `lib`*fichero*`.a` y lo usa para enlazarlo.

`-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.

`-Wall` 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 `stderr` las órdenes ejecutadas por `g++`.

`-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).

4.4. Ejemplos

Ejemplo 2 *El caso más simple es la ejecución de* `g++` *sin especificar ninguna opción.*

```
% g++ saludo.cpp
```

Esta orden realiza las siguientes tareas:

1. *Compila el fichero fuente* `saludo.cpp` *generando un fichero objeto temporal*
2. *Enlaza ese fichero temporal con las bibliotecas del sistema y genera un fichero ejecutable llamado* `a.exe` *(MS-DOS) ó* `a.out` *(Linux/Unix).*
3. *Borra el fichero temporal.*

Si lo que deseamos es que el ejecutable tenga un nombre más adecuado, por ejemplo `saludo`*, especificaremos el nombre con la opción* `-o`*:*

```
% g++ -o saludo saludo.cpp
```

Esta orden hace las mismas tareas que la anterior, salvo que se da el nombre de `saludo` *al ejecutable que se genera.*

*Se puede especificar un camino (*path*) en la opción* `-o`*. Por ejemplo, suponiendo que* `g++` *se ejecuta bajo MS-DOS, la siguiente orden:*

```
% g++ -o /programs/bin/saludo saludo.cpp
```

guarda el fichero ejecutable `saludo` *en el directorio* `\programs\bin`*. Obsérvese el uso de la barra de división (*`/`*) para especificar directorios, independientemente de que se pueda estar trabajando en el sistema operativo MS-DOS. Esta es una característica heredada del origen Unix del compilador. Si se ejecutara bajo Linux/Unix, el ejecutable se guardará el el directorio* `/programs/bin` *(si se tuviera permiso para escritura en ese directorio, por supuesto).*

Ejemplo 3 *En este ejemplo mostraremos cómo generar (únicamente) ficheros objeto. En este caso se trata de indicar a* `g++` *que compile solamente, sin llamar al enlazador. La siguiente orden:*

```
% g++ -c saludo.cpp
```

compila el fichero fuente `saludo.cpp`*, generando el fichero objeto* `saludo.o`*. No realiza la etapa de enlazado: la opción* `-c` *hace que* `g++` *se comporte como un compilador.*

Observar que no se ha especificado el nombre del objeto con la opción `-o`*. Por defecto crea un fichero objeto con el mismo nombre que el fuente, cambiando la extensión* `.cpp` *por* `.o`*. Si quisiéramos especificar el nombre del fichero objeto resultante, podríamos hacer, por ejemplo,*

```
% g++ -c -o objeto saludo.cpp
```

y se generaría el módulo objeto llamado `objeto`*.*

Ejemplo 4 *En este ejemplo mostraremos cómo especificar directorios de ficheros de cabecera o bibliotecas y cómo se especifican las bibliotecas que han de enlazarse.*

La siguiente orden:

```
% g++ -c -I/usr/local/include -o saludo.o saludo.cpp
```

añade el directorio `/usr/local/include` *a la lista de directorios en los que buscar los ficheros de cabecera. Si el fichero* `saludo.cpp` *incluye (con* `#include`*) un fichero de cabecera que se encuentra en el directorio* `/usr/local/include` *(suponiendo la ejecución bajo Linux/Unix), la orden anterior hace que* `g++` *pueda encontrarlo para el preprocesador. Pueden incluirse cuantos directorios se deseen, por ejemplo:*

```
% g++ -c -I/usr/local/include -I../include -I. -o saludo.o saludo.cpp
```

Los directorios de bibliotecas se especifican de forma análoga. Por ejemplo, la siguiente orden:

```
% g++ -o saludo -L/usr/local/lib saludo.o -lutils
```

llama al enlazador para que enlace el objeto `saludo.o` *con la biblioteca* `libutils.a` *y obtenga el ejecutable* `saludo`*. Busca el fichero de biblioteca* `libutils.a` *en los directorios del sistema y en el directorio* `/usr/local/lib` *(suponiendo la ejecución bajo Linux/Unix).*

Ejemplo 5 *En este ejemplo mostraremos cómo usar las macros en las llamadas a* `g++`*.*

La ejecución de la siguiente orden:

```
% g++ -DMAXIMO=100 saludo.cpp -o saludo
```

compila y enlaza el fichero `saludo.cpp` *generando el ejecutable* `saludo`*. Además, se define una constante simbólica llamada* `MAXIMO` *con valor* `100`*. Esta definición tiene el mismo efecto que haber incluido la línea* `#define MAXIMO 100` *en* `saludo.cpp`*.*

El ámbito de la definición se extiende a todos *los ficheros fuente implicados en la llamada a* `g++`*. Por ejemplo, dados los siguientes ficheros fuente:*

```
/**************************************************/
// Fichero: ppal.cpp
/**************************************************/
#include <iostream>
using namespace std;

void f ();

int main ()
{
```

```
   cout << "MACRO en main(): " << MACRO << endl;
   f ();

   return (0);
}

/**************************************************/
// Fichero: funcion.cpp
/**************************************************/
#include <iostream>
using namespace std;

void f ()
{
   cout << "MACRO en f(): " << MACRO << endl;
}
```

se observa que en ambos se utiliza `MACRO`*, que no está definida. Si se ejecuta la siguiente orden:*

```
% g++ -DMACRO=10 ppal.cpp funcion.cpp -o ejecutable
```

se crea el fichero ejecutable llamado `ejecutable` *a partir de* `ppal.cpp` *y* `funcion.cpp`*. Para este fin, las tareas que desencadena* `g++` *son las siguientes:*

1. *Crear los objetos correspondientes a* `ppal.cpp` *y* `funcion.cpp` *(son ficheros temporales).*
2. *LLamar al enlazador para que enlace estos objetos y las bibliotecas adecuadas del sistema para generar* `ejecutable`*. Puede generarse un fichero ejecutable porque uno de los fuentes (*`ppal.cpp`*) contiene la función* `main()`*.*
3. *Borrar los objetos temporales.*

La ejecución del programa `ejecutable` *demuestra que la definición de* `MACRO` *se conoce en los dos ficheros fuente involucrados en la llamada a* `g++`*:*

```
MACRO en main(): 10
MACRO en f(): 10
```

Recordar que no es necesario especificar un valor asociado a las macros en la llamada a `g++`*. Si así se hace, se considera que las macros están, simplemente, definidas. Por defecto se les asigna el valor 1. Por ejemplo, si después de llamar a* `g++` *así:*

```
% g++ -DMACRO ppal.cpp funcion.cpp -o ejecutable
```

se ejecuta el programa `ejecutable`*, obtenemos el siguiente resultado:*

```
MACRO en main(): 1
MACRO en f(): 1
```

Ejemplo 6 *Supongamos que se dispone de un módulo (*`tipos.cpp`*) que define y usa vectores genéricos y matrices (cuadradas) genéricas. La generalidad debe entenderse relativa al tipo base de los elementos que se almacenen y al número de casillas de la matriz. En definitiva, el fichero fuente utiliza la constante* `N`*, que no se define en él y el tipo* `tipo_base`*, definido a partir de la existencia de una constante (*`INT` *o* `FLOAT`*) como se muestra en el siguiente trozo de código.*

```
#if defined(INT)
  typedef int tipo_base;
#elif defined(FLOAT)
  typedef float tipo_base;
```

```
#endif
.......
tipobase v[N];
tipobase m[N][N];
.......
```

A la hora de generar el ejecutable hay que especificar estas macros en la llamada a `g++`*. Así, si queremos, por ejemplo, un ejecutable (*`prog1`*) que gestione vectores de enteros (*`int`*) de tamaño 10 y matrices cuadradas de enteros de dimensión* 10×10*, llamaremos a* `g++` *de la siguiente manera:*

```
% g++ -o prog1 -DN=10 -DINT tipos.cpp
```

Si lo que queremos es, por ejemplo, un ejecutable (`prog2`*) que gestione vectores de reales (*`float`*) de tamaño 20 y matrices cuadradas de reales de dimensión* 20×20*, llamaremos a* `g++` *así:*

```
% g++ -o prog2 -DN=20 -DFLOAT tipos.cpp
```

El lector habrá deducido como el uso de macros nos permite construir diferentes ejecutables utilizando el mismo fichero fuente como base.

Ejercicio

1. Crear el fichero `saludo.cpp` que imprima en la pantalla un mensaje de bienvenida, compilarlo y enlazarlo según los ejemplos vistos anteriormente.

2. Crear el fichero `calculos.cpp` a partir del listado de la sección 8.3 (el objetivo de este programa es muy sencillo y se puede entender sin más que mirar el código). Corregir los errores sintácticos y semánticos que pueda tener y completar el código que falta por implementar (señalado entre comentarios en el mismo código). Algunos datos de prueba para el programa

   a) $1! = 1, 4! = 24$

   b) $1^0 = 1, 2^{10} = 1024$

   c) $\binom{6}{0} = 1, \binom{6}{3} = 20$

5. Introducción al depurador DDD

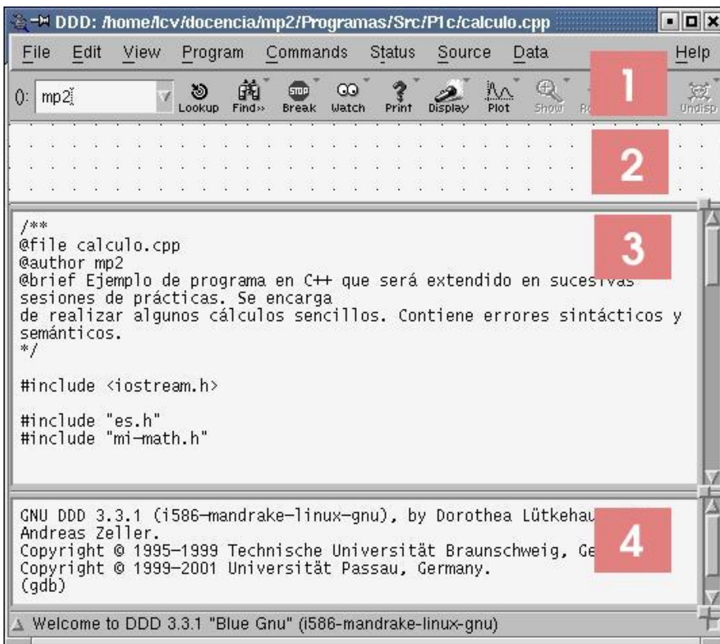
En esta sección se supone que se conocen los fundamentos de la depuración de programas (explicados en la asignatura Metodología de la Programación I) y se centrará en explicar el funcionamiento básico del depurador `ddd`.

5.1. 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` (ver sección 9) para ejecutables binarios, aunque se pueden utilizar otros depuradores.

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`*



(a)

(b)

Figura 6: Pantalla principal de ddd

5.2. Pantalla principal

La pantalla principal del depurador se muestra en la figura 6.a).
En ella se pueden apreciar las siguientes partes.

1. Zona de menú y barra de herramientas. Con los componentes típicos de cualquier programa y una ventana a la izquierda señalada por (), donde aparecerán los identificadores considerados, números de línea, . . .

2. Zona de visualización de datos. En esta parte de la ventana se mostrarán las variables que se hayan elegido y sus valores asociados.

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.

4. Zona de visualización de mensajes de gdb. Muestra los mensajes del verdadero depurador, en este caso, gdb.

Sobre la ventana principal aparece una ventana flotante de herramientas que se muestra en la figura 6.b) desde la que se pueden hacer, de forma simplificada, las operaciones de depuración más frecuentes.

5.3. Opciones fundamentales del menú del depurador

A continuación se comentan las opciones más importantes de los menús del programa ddd.

- FILE: Permite la realización de diversas operaciones con los programas que se desean depurar
  - Open Program: Abre un programa ejecutable compilado con la opción -g

- Open Recent: Recarga el programa que ha sido depurado recientemente.
- Open Core Dump: Abre un fichero core
- Open Source: Permite abrir los fuentes que intervienen en el programa.
- Open Session: Abre una sesión, anteriormente grabada.
- Save Session as: Graba una sesión, para abrirla después.
- Change Directory: Cambia de directorio de trabajo.
- Make: Ejecuta el make que exista en el directorio de trabajo.
- Restart: Recarga el ddd
- Exit: Sale del ddd

■ EDIT: Permite la edicción entre diferentes ventanas.

- Undo: Anula la acción anterior
- Redo: Vuelve a ejecutar la acción anterior
- Cut: Elimina el texto marcado y lo guarda en le portapapeles.
- Copy: Copia en el portapapeles
- Paste: Inserta lo que hay en el portapapeles
- Clear: Limpia el área de texto seleccionada más recientemente.
- Delete: Elimina el texto señalado, pero no lo pone en el portapapeles
- Select All: Selecciona todos los caracteres, a partir del área de texto seleccionada más recientemente
- Preferences: Permite adptar las preferencias del ddd
- GDB Settings: Adapta las preferencias del depurador
- Save Options: Graba las opciones para la próxima sesión

■ VIEW: Despliegue de ventanas específicas

- Command Tool: Abre, si no está abierta, la ventana de herramientas.
- Execution Window: Abre una ventana independiente, donde se ejecuta el programa.
- Debuger Console: Abre la ventana del depurador gdb (ventana inferior).
- Source Window: Abre la ventana donde se sitúa el código fuente
- Data Window: Abre la ventana de datos (encima de la ventana del fuente).
- Machine Code Window: Abre la ventana de código máquina (entre las ventanas fuente y gdb).

■ PROGRAM: Permite la realizar varias operaciones de depuración.

- Run: Empieza a ejecutar el programa, permitiendo proporcionar argumentos.
- Run Again: Ejecuta el programa con los argumentos más recientes.
- Run in Execution Window: Si está activada la ventana de ejecución, se ejecuta en dicha ventana.
- Step: Continua la ejecución, hasta la siguiente línea, que puede pertenecer a una función que halla sido llamada, siempre que el módulo donde aparezca dicha función halla sido compilado con la opción -g, en caso contrario, no entrará en la función.
- Step Instruction: Realiza la misma función que Step sobre el código máquina.

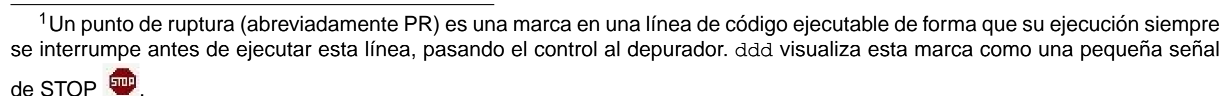
  - Next: Continúa la ejecución hasta la siguiente línea fuente, pero no se detiene en las líneas de las funciones llamadas, a no ser que contengan puntos de ruptura[1].
  - Next Instruction: Realiza la misma función que Next sobre el código máquina.
  - Until: Continúa hasta la siguiente línea, o hasta el número de línea indicada como parámetro en la ventana gdb.
  - Finish: Se ejecuta el programa hasta el final de la función actual, situándose en la siguiente línea de la llamada a la función, y mostrando el valor devuelto por return.
  - Continue: Continúa hasta después de que llega a una línea señalada con un punto de ruptura.
  - Kill: Mata el proceso que está siendo depurado.
  - Interrupt: Para la ejecución del programa o proceso, indicando el lugar en el que se ha detenido.
  - Abort: Aborta el programa.

- COMMANDS: Permite realizar diversas acciones sobre el historial de órdenes y la consola gdb.
  - Command History: Visualiza la historia de órdenes.
  - Previous: Visualiza la orden anterior en la historia.
  - Next: Visualiza la siguiente orden en la historia.
  - Find Backward, Forward: Realiza una búsqueda hacia adelante o hacia atrás de una orden, en la historia.

- STATUS: Permite examinar el estado de la pila, registros, hebras etc.
  - Backtrace: Muestra el orden de las funciones en la pila, desde main hasta el punto de ejecución actual.
  - Up: Muestra el lugar y la función desde la que se ha llamado a la función actual.
  - Down: Muestra la función a la que llamó la función actual.

- SOURCE: Permite realizar diversas acciones sobre el código fuente.
  - Breakpoints: Edita todos los puntos de ruptura.
  - Lookup(): Muestra la línea donde está declarado el identificador que aparezca en la ventana ().
  - Find(): Busca la siguiente aparición del argumento de la ventana ().
  - Find(): Busca la anterior aparición del argumento de la ventana ().
  - FindWords Only: Considera sólo palabras completas en las búsquedas.
  - Find Case Sensitive: Si está activado, distingue entre mayúsculas y minúsculas.
  - Display Line Numbers: Muestra el número de línea en el código fuente.
  - Display Machine Code: Si está activado muestra en una ventana el código máquina.
  - Edit Source: Activa un editor con el código fuente con el vi por defecto, pero si queremos utilizar otro editor, debemos poner en la consola, antes de ejecutar el ddd: XEDITOR=nombre-editor, si utilizamos bash, por ejemplo, XEDITOR=kwrite, o setenv XEDITOR nombre-editor, si utilizamos tcsh.
  - Reload Source: Vuelve a cargar el fuente.

- DATA: Permite realizar distintas operaciones sobre los datos.

[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 .

- Displays: Presenta en una ventana, el valor de las variables, punteros, etc. XXXX
- Watchpoints: Edita los puntos de control (Watch)[2].
- Memory: Presenta un volcado de la memoria
- Print(): Presenta el valor de () en la consola del depurador.
- Display(): Presenta el valor de () en la ventana de datos
- Display Local Variables: Muestra todas las veriables locales en la ventana de datos.
- Display Arguments: Muestra todos los argumentos.
- Status Displays: Muestra distintas opciones del gdb.
- Refresh: Refresca los valores de la ventana de datos.

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

[2] Un punto de control, es una especie de punto de ruptura, que detiene el programa cuando el valor de () cambia

5.4. Ejecución de un programa paso a paso

Una vez cargado un programa binario, se comienza la ejecución mediante la orden *run*. Hay que tener en cuenta que esta orden inicia la ejecución del programa de la misma forma que si se hubiese llamado desde la línea de argumentos, de forma que 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`

Algunas órdenes pueden ser introducidas directamente en la línea de órdenes de la ventana del gdb. En cualquier momento se puede terminar la ejecución de un programa de distintas formas, la más rápida es mediante la orden `kill`. Se pueden pasar argumentos a la función `main` desde la ventana que aparece en la figura 7 que aparece al pulsar *Program - Run*.

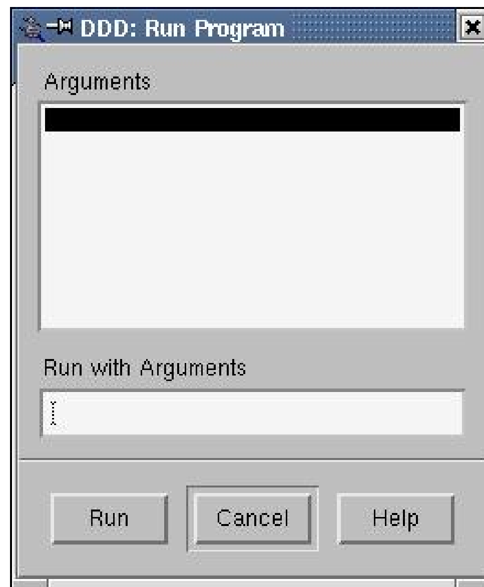


Figura 7: Ventana para pasar argumentos a `main`

Para comenzar a ejecutar un programa bajo control del depurador es conveniente colocar un punto de ruptura 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 programa en una ventana independiente (`DDD: Execution window`). Un vez que el programa se detiene, se pueden utilizar las órdenes descritas en el cuadro 1 para seguir la depuración.

5.5. 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 (mediante *display*, de forma que sus valores se visualicen durante toda la ejecución. Es necesario aclarar que sólo se puede visualizar el valor de una variable cuando la línea de ejecución activa se encuentre en el ámbito de esta variable. Así mismo, `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.

5.6. 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 y analizar qué llamadas se están resolviendo en un momendo dado de la ejecución de un programa.

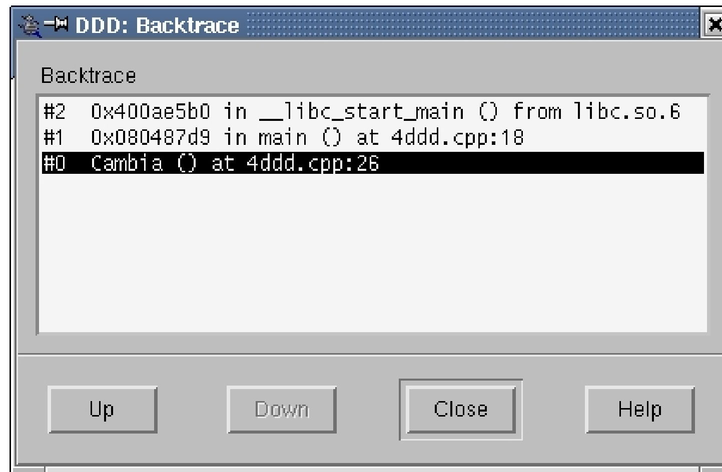


Figura 8: Ventana que muestra el estado de la pilla de llamadas a módulos

5.7. 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.

5.8. 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 y recompilar los módulos que fuese necesarios. *ddd* recargará el programa automáticamente cuando se pulsee *Run*.

Ejercicio

1. Escribir y compilar el programa de la sección 8.1 ¿Funciona correctamente?
2. Depurarlo paso a paso, encontrar los errores y repararlos.

6. El programa *make* y la construcción de ficheros *makefile*

6.1. Introducción

La gestión y mantenimiento del software durante el proceso de desarrollo puede ser una tarea ardua si 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 sotware y las modificaciones incorporadas pueden afectar a otros módulos (recordemos el ejemplo presentado en la sección 2.6: 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 y se crea mediante cualquier editor.

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 centranos en esta utilidad y descartemos el uso de gestores de proyectos tales como el que proporciona Borland, que, aunque recuerda a la utilidad `make` y puede resultar más amigable de cara al usuario, nos encasilla en una visión muy particular de la gestión de proyectos software. En este apéndice 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 proporciona el mecanismo para realizar una gestión inteligente, sencilla y precisa de un proyecto software, ya que permite:

1. Una forma sencilla de especificar las dependencias entre los módulos de un proyecto software,

2. La recompilación únicamente de los módulos que han de actualizarse,

3. Obtener siempre la última versión 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.

6.2. 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 comparando la fecha y hora asociada a cada fichero: si el fichero fuente es más reciente que el fichero destino lo reconstruye. Este sencillo mecanismo (suponiendo que se ha especificado correctamente la dependencia entre módulos) hace posible mantener siempre actualizada la última versión.

6.2.1. 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 `/` dependiendo del sistema operativo y/o versión de make. En la sección 6.2.2 de este apéndice enumeramos las opciones más importantes.
- ***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 (sección 6.3.3 de este apéndice). Una explicación detallada de los destinos puede encontrarse en las secciones 6.3.2 y 6.3.4.

Obsérvese que tanto las opciones como los destinos son opcionales, por lo que podría ejecutarse `make` sin más. El efecto de esta ejecución, así como el funcionamiento detallado de `make` cuando se especifican destinos se explica en la sección 6.2.3 de este apéndice.

6.2.2. Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

`-h` o `--help`: Proporciona ayuda acerca de `make`.

`-f` *fichero*: Utilizaremos esta opción si se proporciona a `make` un fichero makefile que no se llame `makefile` ni `Makefile`. Se toma el fichero llamado `fichero` como el fichero makefile. Por esta razón, hablaremos de fichero makefile cuando nos refiramos a un fichero de este tipo, mientras que escribiremos `makefile` cuando queramos referirnos a un fichero makefile que se llame así.

NombreMacro[=***cadena]*** define una constante simbólica (***NombreMacro***) con el nombre especificado como la cadena indicada después del signo `=`. Si ***cadena*** contiene espacios, será necesario encerrar ***cadena*** entre comillas. En la sección 6.5.1 se detallará su funcionamiento. Si la constante simbólica ya está definida dentro del fichero makefile, se ignorará el valor de la constante simbólica del fichero makefile.

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

6.2.3. 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: el destino especificado depende de otros que no están construidos. Si es así, los construye examinando 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.

Poco más se puede decir en estos momentos sobre la utilidad `make` si no se conoce la estructura de un fichero makefile. En la siguiente sección mostraremos ejemplos de uso de `make`, una vez conocido cómo escribir ficheros makefile.

6.3. 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.

La sintaxis de un fichero makefile varía ligeramente de un sistema a otro, si bien las líneas básicas son similares. En esta sección describiremos la sintaxis de un fichero makefile para la utilidad `make` de GNU, si bien los elementos que describimos pueden encontrarse en la especificación de ficheros makefile en cualquier sistema. Los elementos comunes entre diferentes sistemas que pueden incluirse en un fichero makefile son los siguientes:

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

Antes de especificar con detalle la sintaxis de los ficheros makefile, es conveniente que el lector recuerde de nuevo el ejemplo empleado en la sección 2.6. En este ejemplo presentamos la idea de lo que es un fichero makefile y su utilidad para gestionar proyectos de software en los que intervienen diferentes módulos. Este ejemplo puede servirle como ilustración adicional del contenido de la sección en la que nos encontramos.

6.3.1. 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.

Ejemplo 7 *En el siguiente fichero makefile (llamado* `makefile`*):*

```
# Fichero: makefile
# Construye el ejecutable "saludo" a partir de "saludo.cpp"

saludo : saludo.cpp
        g++ saludo.cpp -o saludo
```

se incluyen dos líneas de comentario al principio del fichero `makefile` *que indican las tareas que realizará la utilidad* `make`*. La descripción del resto del fichero la realizamos en el ejemplo 8.*

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
```

6.3.2. Reglas. Reglas explícitas

Las reglas constituyen el mecanismo por el que se indica a la utilidad `make` los destinos, 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.

Las reglas tienen este formato general:

Línea de dependencia
orden(es)

La línea de dependencia es diferente para las reglas explícitas e implícitas, pero las instrucciones aplicables son las mismas. En esta sección describiremos con detalle las **reglas explícitas**, dejando para la sección 6.5.2 la descripción de las **reglas implícitas**.

Reglas explícitas

El formato habitual de una regla explícita es el siguiente:

destino: *lista de dependencia*
orden(es)

donde:

- **destino** especifica el fichero a crear.
- ***lista de dependencia*** especifica los ficheros de los que depende **destino**. La lista se especifica separando los nombres de los ficheros con 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 **destino**.
- ***órden(es)*** son órdenes válidas para el sistema operativo en el que se ejecute la utilidad `make` (en nuestro caso serán habitualmente llamadas al compilador en línea `g++`). Pueden incluirse cuantas instrucciones se requieran como parte de una regla, cada uno en una línea distinta. Usualmente estas instrucciones sirven para construir el **destino**, 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.

Ejemplo 8 *En el ejemplo anterior (fichero* `makefile`*) encontramos una única regla:*

```
saludo : saludo.cpp
	g++ saludo.cpp -o saludo
```

que indica que para construir el destino `saludo` *se requiere la existencia de* `saludo.cpp` *(*`saludo` depende de `saludo.cpp`*) Ese destino se construye ejecutando la orden:*

```
g++ saludo.cpp -o saludo
```

que compila el fichero `saludo.cpp` *generando un fichero objeto temporal y lo enlaza con las bibliotecas adecuadas para generar finalmente* `saludo`*. Como hay una única regla y el fichero se llama* `makefile`*, las siguientes órdenes:*

```
% make
% make -f makefile
% make saludo
% make -f makefile saludo
```

tienen el mismo efecto. En los dos últimos casos se ha especificado el destino a crear, aunque no hace falta ya que es el primero del fichero `makefile`*.*

Ejemplo 9 *En este ejemplo mostramos un fichero makefile llamado* `makefil2.mak` *en el que se especifican dos reglas. La primera es la misma regla que la del ejemplo anterior y la segunda especifica un destino que no es un fichero ejecutable.*

```
# Fichero: makefil2.mak
# Por defecto, construye el ejecutable "saludo" a partir de "saludo.cpp"
# Tambien puede construirse el objeto "saludo.o" a partir de "saludo.cpp"

saludo : saludo.cpp
	g++ saludo.cpp -o saludo

# Esta regla especifica un destino que no es un fichero ejecutable.

saludo.o : saludo.cpp
	g++ -c saludo.cpp -o saludo.o
```

Estas dos órdenes:

```
% make -f makefil2.mak saludo
% make -f makefil2.mak
```

tendrán el mismo efecto ya que `saludo` *es el primer destino de* `makefil2.mak`*. Ahora, para construir el segundo destino se ejecutará:*

```
% make -f makefil2.mak saludo.o
```

6.3.3. 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`.

Prefijos de órdenes

Las órdenes pueden ir precedidas por prefijos. Los más importantes son los que enumeramos a continuación, de los que mostraremos ejemplos a lo largo de este capítulo

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

Ejemplo 10 *En este fichero makefile se especifican dos órdenes en cada una de las reglas. Además se incluye una orden para mostrar un mensaje en pantalla (*`echo`*), indicando qué acción se desencadena en cada caso. Esta orden puede usarse indistintamente en MS-DOS o Linux/Unix. Las órdenes* `echo` *van precedidos por el prefijo* `@` *en el fichero makefile para indicar que no debe mostrarse esa orden por la pantalla, sino solo su resultado.*

```
# Fichero: makefil3.mak
# Construye el ejecutable "saludo" a partir de "saludo.cpp"
# Tambien puede construirse el objeto "saludo.o" a partir de "saludo.cpp"

saludo : saludo.cpp
        @echo Creando saludo...
        g++ saludo.cpp -o saludo

# Esta regla especifica un destino que no es un fichero ejecutable.

saludo.o : saludo.cpp
        @echo Creando saludo.o solamente...
        g++ -c saludo.cpp -o saludo.o
```

Si se ejecuta la orden: `make -f makefil2.mak` *se realizan las siguientes tareas:*

```
Creando saludo...
g++ saludo.cpp -o saludo
```

Si no se hubiera usado el prefijo `@` *delante de la orden* `eco`*, el resultado hubiera sido:*

```
echo Creando saludo...
Creando saludo...
g++ saludo.cpp -o saludo
```

ya que por defecto se muestran en la consola las órdenes que se van ejecutando. Podemos, incluso, poner el prefijo `@` *delante de la llamada a* `g++` *y el resultado sería:*

```
Creando saludo...
```

esto es, no se muestra qué orden se está ejecutando.

Ejemplo 11 *Este ejemplo es una extensión del anterior en el que se ha incorporado una nueva regla, cuyo destino es* `clean` *que no tiene asociada una lista de dependencia.*

```
# Fichero: makefil4.mak
# Por defecto, construye el ejecutable "saludo" a partir de "saludo.cpp"
# Incorpora dos reglas mas:
#   1) Crear el objeto "saludo.o" a partir de "saludo.cpp"
#   2) Novedad: Regla sin lista de dependencia.

saludo : saludo.cpp
        @echo Creando saludo...
        g++ saludo.cpp -o saludo

# Esta regla especifica un destino que no es un fichero ejecutable.

saludo.o : saludo.cpp
        @echo Creando saludo.o solamente...
        g++ -c saludo.cpp -o saludo.o

# Esta regla especifica un destino sin lista de dependencia

clean :
        @echo Borrando ficheros .o...
        del *.o
```

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 makefil4.mak clean
```

6.3.4. Destinos Simbólicos

Un destino simbólico se especifica en un fichero makefile en la primera línea operativa del mismo o, en cualquier caso, antes de cualquier otro destino. 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 ningúna 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.

Ejemplo 12 *El fichero makefile* `makefil5.mak` *contiene en su primera línea operativa el destino simbólico llamado* `saludos`*. La lista de destinos asociada a éste es:* `saludo`*,* `saludo2`*, y* `saludo3`

```
# Fichero: makefil5.mak
# Ejemplo de fichero makefile con un destino simbolico llamado "saludos"

saludos: saludo saludo2 saludo3

saludo : saludo.cpp
	@echo Creando saludo...
	g++ saludo.cpp -o saludo

saludo2 : saludo2.cpp
	@echo Creando saludo2...
	g++ saludo2.cpp -o saludo2

saludo3 : saludo3.cpp
	@echo Creando saludo3...
	g++ saludo3.cpp -o saludo3

clean :
	@echo Borrando ficheros .o...
	del *.o
```

Con este ejemplo, si se ejecuta:

```
make -f makefil5.mak
```

como el fichero `makefil5.mak` *contiene varios destinos posibles y no se especifica ninguno,* `make` *intentará construir el primero, llamado* `saludos`*. Antes de plantearse la construcción de* `saludos` *debe construir (si no lo están) todos los que aparecen en la lista de dependencia asociada (*lista de destinos*):* `saludo`*,* `saludo2` *y* `saludo3`*. Una vez construidos, se plantea la construcción de* `saludos`*, pero al no tener ninguna orden para ello, termina.*

Obsérvese que el destino `clean` *no se construye, a no ser que se indique* explícitamente *con la orden* `make -f makefil5.mak clean`

Ejemplo 13 *En este ejemplo se incorpora el destino* `clean` *a la lista de dependencia del destino simbólico* `saludos`*, y lo que es más importante, se incorpora un destino llamado* `salva` *que a su vez es destino en una regla nueva con una lista de dependencia asociada. Cuando se intenta construir el destino llamado* `salva` *ya se han construido* `saludo`, `saludo2` *y* `saludo3` *por lo que se verifica la lista de dependencia y pasan a ejecutarse las órdenes asociadas a esta regla.*

```
# Fichero: makefil6.mak
# Ejemplo de fichero makefile con un destino simbolico llamado "saludos"
# y uso de una macro predefinida ($^).

saludos: saludo saludo2 saludo3 clean salva

saludo : saludo.cpp
	@echo Creando saludo...
	g++ saludo.cpp -o saludo

saludo2 : saludo2.cpp
	@echo Creando saludo2...
	g++ saludo2.cpp -o saludo2

saludo3 : saludo3.cpp
	@echo Creando saludo3...
	g++ saludo3.cpp -o saludo3
clean :
	@echo Borrando ficheros .o...
	rm *.o

salva : saludo saludo2 saludo3
	@echo Creando directorio resultado
	mkdir resultado
	@echo Moviendo los  a resultado
	mv $^ resultado
```

Por último, observe la última orden asociada a la última regla:

```
	move $^ resultado
```

En este orden 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 saludo2 saludo3 resultado
```

Ejercicio

1. *Copiar el makefile mostrado en la sección 8.2 y probar su funcionamiento por defecto. Analizar el orden de ejecución de las reglas.*

   ```
   make -f makefile.ej1
   ```

2. *Probar a pasar el argumento* `bigking` *y comprobar su efecto.*

   ```
   make -f makefile.ej1 bigking
   ```

3. *Extenderlo con una regla más que meta el producto en una caja en el caso de que sea para llevar. Esta regla será complementaria a las dos reglas principales del makefile (whooper y bigking).*

   ```
   make -f makefile.ej1 bigking parallevar
   ```

4. *Probar los makefiles mostrados en esta sección para el programa de saludo que se creó en la sección 4*

A continuación presentaremos las macros predefinidas para ficheros makefile.

Macros predefinidas

Las macros predefinidas utilizadas habitualmente en ficheros makefile son las que enumeramos a continuación. En el ejemplo anterior hemos utilizado la macro `$^` y a lo largo de este capítulo mostraremos en diversos ejemplos cómo se usan las demás.

`$^` Equivale a *todas* las dependencias de la regla, con un espacio entre ellas.
`$<` Nombre de la *primera dependencia* de la regla.
`$@` Nombre del fichero *destino* de la regla.

Destinos .PHONY

En los ejemplos 12 y 13 podría presentarse el siguiente problema: si alguien crea un fichero en el directorio actual con el nombre `clean`, la orden:

```
make -f makefil5.mak clean
```

no funcionará (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 solucionar esto 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 -f makefil5.mak 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.

6.4. Ejemplo: Gestión de un proyecto software

En esta sección mostraremos un ejemplo detallado de modularización del software en C++, en el que partiremos de una primera versión en la que todo el código está en un solo fichero hasta una versión en la que se distribuye en varios ficheros, con sus correspondientes ficheros de cabecera. Es importante destacar que, en cualquier caso se trata del *mismo código* en todas las versiones, tan solo se diferencian en la distribución del mismo en diferentes ficheros. Además, cada versión del programa viene acompañada del correspondiente fichero makefile.

Descripción del problema

El problema que nos ocupa es rellenar de forma aleatoria un vector de 25 enteros y ordenarlos de forma ascendente mediante el método de la burbuja.

6.4.1. Versión 1

En la primera versión del programa, el código se presenta en un solo fichero fuente (`ordena1.cpp`). El listado de los ficheros de este ejemplo aparece en la sección 8.4.

```
# Fichero: makefile.v1
# Generacion del ejecutable a partir de un unico fuente: "ordena1.cpp

destinos: ordena1

ordena1 : ordena1.cpp
   @echo Compilando ordena1.cpp ...
   g++ ordena1.cpp -o ordena1
```

El destino simbólico llamado `destinos` hace que se genere únicamente el ejecutable `ordena1`.

6.4.2. Versión 2

En esta versión del programa, se estructura el código en dos ficheros fuente y en un fichero de cabecera (ver sección 8.5).

1. Uno de los ficheros fuente (`ppal.cpp`) contiene únicamente la función `main()`.
2. El otro fichero fuente (`funcsvec.cpp`) contiene las restantes funciones y las definiciones de las constantes `MAX_LINE` y `MY_MAX_RAND`. Estas constantes se usan únicamente por las funciones definidas en este módulo, de ahí que se oculten a la función `main()` haciéndolas locales a este módulo.
3. El fichero de cabecera `funcsvec.h` incluye la definición de la constante `MAX` (la única que necesita conocer la función `main()`) y los prototipos de las funciones `llena_vector()`, `pinta_vector()` y `ordena_vector()`, que son las que invoca `main()`. Observar que hemos logrado un aceptable grado de ocultamiento de información:
   a) Cuando `main()` incluye este fichero **no** conoce la función `swap()`. De hecho, no tiene por qué conocerla ya que no la va a usar. Al no haber incluido su prototipo, nos vemos obligados a especificarlo en la función `ordena_vector()`. Esta función es **local** a `funcsvec.c`.
   b) Tampoco conoce las constantes `MAX_LINE` y `MY_MAX_RAND` ya que se usan únicamente por las funciones de `funcsvec.cpp` y por eso las hacemos **locales** a este módulo.

El fichero makefile necesario para la obtención del programa ejecutable puede ser el siguiente:

```
# Fichero: makefile.v2
# Ejemplo de makefile que genera un ejecutable a partir de dos ficheros objeto.
#   1) "ppal.o":codigo objeto del programa principal (de "ppal.cpp.
#   2) "funcsvec.o": codigo objeto de las funciones auxiliares (de "funcsvec.cpp.

destinos: ordena1 clean

ordena1 : ppal.o funcsvec.o
   g++ -o ordena1 ppal.o funcsvec.o
```

```
ppal.o : ppal.cpp funcsvec.h
   g++ -c -o ppal.o -I. ppal.cpp

funcsvec.o : funcsvec.cpp funcsvec.h
   g++ -c -o funcsvec.o -I. funcsvec.cpp

clean :
   rm ppal.o funcsvec.o
```

En este caso, el ejecutable `ordena1` se construye a partir de los ficheros objeto `ppal.o` y `funcsvec.o`. Ambos ficheros objeto se han generado utilizando la opción `-c` en la llamada a `g++` que genera únicamente el módulo objeto, esto es, sin llamar al enlazador. Así mismo, como los ficheros requeridos para generar el ejecutable son dos ficheros objeto, `g++` llamará únicamente al enlazador.

Observar que en las listas de dependencia para generar los ficheros objeto se incluye al fichero de cabecera `funcsvec.h`, de forma que cualquier modificación en él hará que se reconstruyan los objetos. Además, en la llamada a `g++` se incluye la opción `-I` con el argumento `.`, indicando de esta forma a `g++` que incluya el directorio actual en la lista de búsqueda de los ficheros de cabecera. Esta forma de especificar la dependencia respecto a los ficheros de cabecera resulta, en la práctica, muy útil ya que si no se especificara, cualquier modificación en los ficheros de cabecera no se podría propagar a los ficheros que dependen de éstos. Los cambios que pueden provocar la inconsistencia suelen ser muy sutiles, como la modificación de una constante simbólica. Otros cambios más evidentes, como el cambio de la cabecera de una función conlleva la modificación del prototipo (en el fichero `.h`) y de la definición (en el fichero `.c`) que hace que se reconstruya adecuadamente el destino final ya que seguramente existirá una regla con una línea de dependencia que incluya al fichero `.c` que se ha actualizado.

6.4.3. Versión 3

En esta versión del programa, se estructura el código en tres ficheros fuente y en dos ficheros de cabecera. El anterior fichero `funcsvec.cpp` se ha dividido en dos ficheros fuente, uno conteniendo las funciones que rellenan y muestran el vector (`vec_ES.cpp`), y el otro conteniendo la función de ordenación (`ordena.cpp`), pensando en un posible uso más general de esta función. Por coherencia, el anterior fichero `funcsvec.h` se ha dividido en dos ficheros de cabecera: `vec_ES.h` y `ordena.h` (ver código en sección 8.6).

1. Como antes, uno de los ficheros fuente (`ppal.cpp`) contiene únicamente la función `main()`. Se diferencia con el anterior en que se incluyen los dos ficheros de cabecera.

2. Otro fichero fuente (`vec_ES.cpp`) contiene las funciones de entrada/salida sobre el vector, concretamente, `llena_vector()` y `pinta_vector()`. Este fichero tiene asociado el correspondiente fichero cabecera (`vec_ES.h`).

3. El último fichero fuente (`ordena.cpp`) contiene la función de ordenación `ordena_vector()` y la función auxiliar `swap()`, local a este módulo. Este fichero tiene asociado el correspondiente fichero cabecera (`ordena.h`), que sólo ofrece la función `ordena_vector()`, ya que la función auxiliar `swap()` sólo la usa la primera.

El fichero makefile necesario para la obtención del programa ejecutable será ahora `makefile.v3`. Su estructura será similar a `makefile.v2` sólo que ahora intervienen tres módulos objeto para formar el ejecutable y dos ficheros de cabecera. Sobre `makefile.v3` pueden hacerse las mismas consideraciones que sobre `makefile.v2` acerca de la dependencia respecto de los ficheros de cabecera.

```
# Fichero: makefile.v3
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
```

```
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").

destinos: ordena1 clean

ordena1 : ppal.o vec_ES.o ordena.o
   g++ -o ordena1 ppal.o vec_ES.o ordena.o

ppal.o : ppal.cpp vec_ES.h ordena.h
   g++ -c -o ppal.o -I. ppal.cpp

vec_ES.o : vec_ES.cpp vec_ES.h
   g++ -c -o vec_ES.o -I. vec_ES.cpp

ordena.o : ordena.cpp ordena.h
   g++ -c -o ordena.o -I. ordena.cpp

clean :
   rm ppal.o vec_ES.o ordena.o
```

Ejercicio

1. Construir las tres implementaciones que se han comentado anteriormente, los distintos ficheros y sus makefiles.
2. Descomponer el programa mostrado en el listado de la sección 8.3 en varios ficheros, construir los ficheros de cabeceras que se estimen oportunos, el fichero makefile correspondiente y obtener el ejecutable final.

6.5. Macros, reglas implícitas y directivas condicionales

6.5.1. Macros en ficheros *makefile*

Una **macro o variable MAKE** es una *cadena* que se expande cuando se llama desde 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:

Nombre = texto a expandir

donde:

- **Nombre** 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, `NOMLIB` que representa a la cadena `libejemplo.a` se especificará de la siguiente manera:

```
NOMLIB = libejemplo.a
```

Si esta línea se incluye en el fichero makefile, cuando `make` encuentra la construcción `$(NOMLIB)` en él, sustituye dicha construcción por `libejemplo.a`. Cada macro debe estar en una línea separada en un 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:

```
CFLAGS = $(DEBUG) -c
DEBUG = -g
```

En este ejemplo `CFLAGS` se expandirá a `-g -c`

Ejemplo 14 *El primer ejemplo que ilustra la definición y uso de macros en ficheros makefile es una modificación del fichero* `makefile.v3` *asociado a la versión 3 del proyecto presentado en la sección anterior.*

```
# Fichero: makefil2.v3 (Version 2 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").
#
# Novedad: uso de macros (OBJ e INCLUDE).

OBJ = ppal.o vec_ES.o ordena.o
INCLUDE = .

destinos: ordena1 clean

ordena1 : $(OBJ)
   g++ -o ordena1 $(OBJ)

ppal.o : ppal.cpp vec_ES.h ordena.h
   g++ -c -o ppal.o -I$(INCLUDE) ppal.cpp

vec_ES.o : vec_ES.cpp vec_ES.h
   g++ -c -o vec_ES.o -I$(INCLUDE) vec_ES.cpp

ordena.o : ordena.cpp ordena.h
   g++ -c -o ordena.o -I$(INCLUDE) ordena.cpp

clean :
   rm $(OBJ)
```

En el ejemplo anterior se define, al principio de las líneas operativas del fichero `makefil2.v3`*, la macro llamada* `OBJ` *cuyo valor es la cadena* `ppal.o vec_ES.o ordena.o` *. Cuando* `make` *procesa* `makefil2.v3` *sustituirá las apariciones de* `$(OBJ)` *por el valor de la macro* `OBJ`*, esto es, la regla:*

```
ordena1 : $(OBJ)
        g++ -o ordena1 $(OBJ)
```

la procesa como:

```
ordena1 : ppal.o vec_ES.o ordena.o
        g++ -o ordena1 ppal.o vec_ES.o ordena.o
```

Además de `OBJ` *se define otra macro llamada* `INCLUDE` *a la que se asigna la cadena* `.` *(el directorio actual). Cuando* `make` *la sustituye, por ejemplo en la regla:*

```
ordena.o : ordena.cpp ordena.h
        g++ -c -o ordena.o -I$(INCLUDE) ordena.cpp
```

la procesa como:

```
ordena.o : ordena.cpp ordena.h
        g++ -c -o ordena.o -I. ordena.cpp
```

exactamente como en `makefile.v3`*. Sin embargo, si se cambiara el directorio de los ficheros de cabecera, tan sólo habrá que modificar el valor de la macro* `INCLUDE` *del fichero makefile, evitando de esta forma tener que cambiar el argumento de la opción* `-I` *en cada llamada a* `g++`*, tarea que conlleva la posibilidad de cometer algún error. Sin duda, esta forma de utilizar las macros en ficheros makefile es muy recomendable.*

Sustituciones de cadenas en macros

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

$(Nombre:TextoOriginal = TextoNuevo)

que se interpreta como: sustituir en la cadena asociada a **Nombre** 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 **Nombre**, se trata de una sustitución *temporal*, por lo que **Nombre** mantiene el valor dado en su definición.

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

```
FUENTE = f1.cpp f2.cpp f3.cpp
```

se pueden sustituir temporalmente los caracteres `.cpp` por .o escribiendo `$(FUENTE:.cpp=.o)`. El valor de la macro `FUENTE` no se modifica, ya que la sustitución es temporal.

Ejemplo 15 *El segundo ejemplo que ilustra la definición y uso de macros en ficheros makefile es una nueva modificación de* `makefile.v3`*. Obsérvese el uso intensivo de macros y de sustituciones en macros que presentamos en* `makefil3.v3`*:*

```
# Fichero: makefil3.v3 (Version 3 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").
# Utiliza las macros OBJ e INCLUDE (version 2).
# Novedad: uso de macros con sustitucion.

OBJ  = vec_ES.o ordena.o
INCLUDE = .
MAIN = ppal.o
EXE  = ordena1
VES  = vec_ES

destinos: ordena1 clean
```

```
$(EXE): $(MAIN) $(OBJ)
   g++ $(MAIN) $(OBJ) -o $(EXE)

$(MAIN): $(MAIN:.o=.cpp) $(VES).h $(VES:vec_ES=ordena).h
   g++ -c -o $(MAIN) -I$(INCLUDE) $(MAIN:.o=.cpp)

$(VES).o: $(VES).cpp $(VES).h
   g++ -c -o $(VES).o -I$(INCLUDE) $(VES).cpp

$(VES:vec_ES=ordena).o: $(VES:vec_ES=ordena).cpp $(VES:vec_ES=ordena).h
   g++ -c -o $(VES:vec_ES=ordena).o -I$(INCLUDE) $(VES:vec_ES=ordena).cpp

clean :
   rm $(OBJ)
```

En este ejemplo se definen, al principio de las líneas operativas del fichero `makefile`*, las macros* `OBJ`*,* `INCLUDE`*,* `MAIN`*,* `EXE` *y* `VES`*. Las sustituciones en macros se especifican en las reglas 2 y 4:*

- **Regla 2**.

  En este caso, las líneas siguientes:

  ```
  $(MAIN): $(MAIN:.o=.cpp) $(VES).h $(VES:vec_ES=ordena).h
          g++ -c -o $(MAIN) -I$(INCLUDE) $(MAIN:.o=.cpp)
  ```

  son interpretadas por `make` *tomando en cuenta el valor de las macro* `MAIN` *(*`ppal.o`*) y* `VES` *(*`vec_ES`*). Las sustituciones dan como resultado:*

  ```
  ppal.o: ppal.cpp vec_ES.h ordena.h
        g++ -c -o ppal.o -I. ppal.cpp
  ```

  sin modificar el valor de las macros `MAIN` *y* `VES`*, que siguen teniendo los valores iniciales. Obsérvese que se ha utilizado la sustitución* `$(VES:vec_ES=ordena).h` *para sustituir el nombre del fichero, no la extensión (que es lo usual).*

- **Regla 4**.

  En este caso, las líneas siguientes:

  ```
  $(VES:vec_ES=ordena).o: $(VES:vec_ES=ordena).cpp $(VES:vec_ES=ordena).h
          g++ -c -o $(VES:vec_ES=ordena).o -I$(INCLUDE) $(VES:vec_ES=ordena).cpp
  ```

  son interpretadas por `make` *tomando en cuenta el valor de la macro* `VES` *(*`vec_ES`*) y da como resultado:*

  ```
  ordena.o: ordena.cpp ordena.h
        g++ -c -o ordena.o -I. ordena.cpp
  ```

Debemos indicar que no es habitual escribir unas reglas tan complejas para hacer algo tan simple. Este ejemplo debe entenderse como que cumple el objetivo de ilustrar la sustitución en macros y que ésta puede producirse en cualquier parte de la cadena.

Macros en llamadas a *make*

Puede especificarse el valor de una macro 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 ya que éste se le pasa como un parámetro más (ver sección 6.2.2). La sintaxis de la llamada a `make` con macros es la siguiente:

`make` **Nombre[= texto a expandir]** [*opciones...*] [*destino(s)*]]

donde **Nombre[=texto a expandir]** define la macro con el nombre **Nombre** con el valor **texto a expandir**.

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.

Ejemplo 16 *El ejemplo que ilustra la especificación de macros en la llamada a* `make` *es muy parecido al del ejemplo 1 de esta sección (*`makefil4.mak`*). La diferencia estriba en la segunda regla, donde se utiliza una macro llamada* `DESTDIR` *que no está definida en el fichero* `makefile`*, por lo que ésta debe definirse en la llamada a* `make`*.*

```
# Fichero: makefil4.v3 (Version 4 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").
# Utiliza las macros OBJ e INCLUDE (version 2).
# Novedad: uso de macros (DESTDIR) en la llamada a make.

OBJ = ppal.o vec_ES.o ordena.o
INCLUDE = .

destinos: ordena1 clean

ordena1 : $(OBJ)
   mkdir $(DESTDIR)
   g++ -o $(DESTDIR)/ordena1 $(OBJ)

ppal.o : ppal.cpp vec_ES.h ordena.h
   g++ -c -o ppal.o -I$(INCLUDE) ppal.cpp

vec_ES.o : vec_ES.cpp vec_ES.h
   g++ -c -o vec_ES.o -I$(INCLUDE) vec_ES.cpp

ordena.o : ordena.cpp ordena.h
   g++ -c -o ordena.o -I$(INCLUDE) ordena.cpp

clean :
   rm $(OBJ)
```

Para que la llamada a `make` *tenga éxito debemos especificar en la llamada el valor de la macro* `DESTDIR`*, esto es, el directorio en el que queremos guardar el fichero* `ordena1`*. Si ejecutamos:*

```
% make DESTDIR=/home/usr/x4563456/bin -f makefil4.v3
```

la segunda regla se interpretará como sigue:

```
ordena1 : ppal.o vec_ES.o ordena.o
   mkdir /home/usr/x4563456/bin
   g++ -o /home/usr/x4563456/bin/ordena1 ppal.o vec_ES.o ordena.o
```

Si no se especifica el valor de `DESTDIR`*,* `make` *mostrará un mensaje de error (número de argumentos insuficiente) y terminará de procesar el fichero makefile sin generar el ejecutable. También aparecerá un mensaje de error si el directorio especificado ya existe (No se puede crear el subdirectorio)*

Estos problemas pueden solucionarse usando el prefijo `-` *antes de la orden, para evitar que* `make` *deje de procesar el fichero makefile cuando la ejecución de la instrucción produce un error. De esta forma aunque la instrucción no se ejecute satisfactoriamente, el fichero makefile continuará procesándose. Por ejemplo, en el fichero makefile de este ejemplo podría ponerse delante de la orden* `mkdir` *el prefijo* `-` *para evitar que se aborte el procesamiento del fichero makefile si no se proporciona el nombre del directorio o éste ya estaba creado previamente. Esa regla quedaría:*

```
ordena1 : $(OBJ)
        -mkdir $(DESTDIR)
        g++ -o $(DESTDIR)/ordena1 $(OBJ)
```

Nota: *Si no se especifica* `DESTDIR`, `g++` *se ejecuta con la liguiente línea de órdenes:*

```
        g++ -o /ordena1 ppal.o vec_ES.o ordena.o
```

e intenta guardar el fichero `ordena1` *en el directorio raiz (*`/`*). Si el usuario no tiene permiso de escritura (habitual en sistemas Linux/Unix) no se guardará el ejecutable. Una solución elegante es indicar el directorio base con otra macro (por ejemplo,* `BASE`*) y escribir la regla como:*

```
BASE = /home/usr/x4563456/
   ......
ordena1 : $(OBJ)
        -mkdir $(BASE)$(DESTDIR)
        g++ -o $(BASE)$(DESTDIR)/ordena1 $(OBJ)
```

así, si no se especifica `DESTDIR`, `g++` *se ejecuta con la liguiente línea de órdenes:*

```
        g++ -o /home/usr/x4563456//ordena1 ppal.o vec_ES.o ordena.o
```

y el ejecutable `ordena1` *se guarda en el directorio base. No obstante, en la sección 6.5.3 mostraremos cómo controlar desde el fichero makefile (mediante directivas de condicionales) si se ha definido una macro y cómo actuar utilizando construcciones condicionales de flujo dentro de un fichero makefile (de forma similar a las directivas de compilación condicional del preprocesador).*

Ejercicio

1. Probar los ejemplos anteriores que involucran la definición y uso de macros.
2. Adaptar el fichero makefile construido para la versión descompuesta del código de la sección 8.3 para que contenga macros que hagan referencia a los distintos ficheros fuente, objeto y ejecutable, que utilice como directorio de salida el directorio `$HOME/mp2/bin` únicamente para los ficheros ejecutables finales, los ficheros objeto `.o` se crearán en el directorio de trabajo.
3. Extender el makefile anterior con una opción `rebuild` que permite reconstruir todo el proyecto independientemente de si los ficheros están actualizados o no (consejo: borrar los ficheros `.o`).
4. Extender el makefile anterior con una opción para incluir información de depuración o no (consejo: utilizar una macro en la llamada a make y usarla en las sucesivas llamadas al compilador).

6.5.2. Reglas implícitas

En los ficheros makefile aparecen, en la mayoría de los casos, reglas que se parecen mucho. En los ejemplos que hemos presentado vemos 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. 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 [3]

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

El ejemplo 17 ilustra el uso de esta regla. Las reglas implícitas que usa make utilizan una serie de macros predefinidas, tales como las del caso anterior (CXX, CPPFLAGS y CXXFLAGS). Estas macros pueden ser definidas:

1. Dentro del fichero makefile,
2. A través de argumentos pasados a make, o
3. Con variables del entorno del sistema operativo.

Estas macros/variables suelen tener un valor por defecto, que dependerá de cada sistema.

Si deseamos que make use reglas implícitas, en el fichero makefile escribiremos la regla *sin ninguna orden*. De esta forma, es posible añadir nuevas dependencias a la regla. Finalmente, también es posible no escribir nada (no escribir la regla) y make usará la regla implícita correspondiente al destino que esté intentando construir.

Ejemplo 17 *El ejemplo que ilustra el uso de reglas implícitas está basado en los ejemplos anteriores (versiones sobre el fichero* makefile.v3*).*

```
# Fichero: makefil5.v3 (Version 5 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").
#
# Novedad: uso reglas implicitas.
#
# En este ejemplo, al no existir reglas explicitas que digan como generar los
# ficheros .o a partir de los ficheros .cpp, se aplica la regla implicita:
#          $(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
# para cada uno de ellos.

CXX = g++
CPPFLAGS =
CXXFLAGS = -I$(INCLUDE)

FUENTES = ppal.cpp vec_ES.cpp ordena.cpp
OBJETOS = $(FUENTES:.cpp=.o)
EXE     = ordena1
INCLUDE = ./include

$(EXE): $(OBJETOS)
   g++ -o $(EXE) $(OBJETOS)
```

[3] Para el caso de programas escritos en lenguaje C, esta regla implícita sería `$(CC) -c $(CPPFLAGS) $(CFLAGS)`

```
ppal.o : $(INCLUDE)/vec_ES.h $(INCLUDE)/ordena.h

vec_ES.o: $(INCLUDE)/vec_ES.h

ordena.o: $(INCLUDE)/ordena.h
```

La primera regla es una regla explícita que indica como crear el ejecutable `ordena1` *a partir de los ficheros objeto* `ppal.o`, `vec_ES.o` *y* `ordena.o`*. Observar cómo se aplica la sustitución de cadenas en la macro* `FUENTES` *para dar valor a* `OBJETOS`*.*

No existen reglas explícitas que digan como construir los ficheros objeto `ppal.o`, `vec_ES.o` *y* `ordena.o` *a partir del fichero fuente correspondiente. No obstante, los módulos objeto dependen de forma* implícita *de los módulos fuente asociados (no es necesario indicar esa dependencia). En esta situación* `make` *utiliza una regla implícita, ejecutando:* `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)` *para cada uno de ellos. En este caso,* **no** *se ha escrito la regla implícita. Además, hemos añadido nuevas dependencias a estas reglas, indicado la dependencia de cada uno de ellos respecto de los ficheros de cabecera:*

- `ppal.o` *depende de los ficheros de cabecera* `vec_ES.h` *y* `ordena.h`*,*
- `vec_ES.o` *depende del fichero de cabecera* `vec_ES.h`*, y*
- `ordena.o` *depende del fichero de cabecera* `ordena.h`*.*

De esta forma, cuando se modifique algún fichero de cabecera se ejecutará la regla implícita que actualiza el o los ficheros objeto necesarios y finalmente el ejecutable. Observar que los ficheros de cabecera se encuentran en un subdirectorio del directorio actual llamado `include` *y que se ha modificado la variable* `CXXFLAGS` *con el valor* `-I$(INCLUDE)` *(se expande a* `-I./include` *para cada ejecución de la regla implícita).*

Por ejemplo, la ejecución (primera) de:

```
% make -f makefil5.v3
```

provoca la ejecución de las siguientes tareas:

```
g++ -I./include -c ppal.cpp -o ppal.o
g++ -I./include -c vec_ES.cpp -o vec_ES.o
g++ -I./include -c ordena.cpp -o ordena.o
g++ -o ordena1  ordena1.o vec_ES.o ordena.o
```

Las tres primeras son consecuencia de la regla implícita `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)`*, donde* `$(CXX)` *es* `g++` *y* `$(CXXFLAGS)` *es* `-I./include`*.*

Si se modificara cualquier fichero fuente (por ejemplo, `ordena.cpp`*) se aplicarán, por este orden, la regla implícita para reconstruir* `ordena.o` *y la explícita para reconstruir* `ordena1`*:*

```
% make -f makefil5.v3
g++ -I./include   -c ordena.cpp -o ordena.o
g++ -o ordena1  ppal.o vec_ES.o ordena.o
```

Si se modificara un fichero de cabecera (por ejemplo, `ordena.h`*) se fuerza a la reconstrucción de los objetos que dependen de él (en este caso,* `ordena.o` *y* `ppal.o`*) con la regla implícita y finalmente se reconstruye el ejecutable* `ordena1` *usando la regla explícita:*

```
% make -f makefil5.v3
g++ -I./include   -c ppal.cpp -o ppal.o
g++ -I./include   -c ordena.cpp -o ordena.o
g++ -o ordena1  ppal.o vec_ES.o ordena.o
```

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

```
$(CC) $(LDFLAGS) $(LOADLIBS)
```

se interpreta como sigue: `programa` se construirá a partir de `programa.o`. Esta regla funciona correctamente para programas *con un solo fichero fuente*. Funcionará también correctamente en programas con múltiples ficheros objeto si uno de los cuales tiene el mismo nombre que el ejecutable. En el siguiente ejemplo mostramos cómo hacerlo.

Ejemplo 18 *El ejemplo que ilustra el uso de la regla implícita para enlazar está basado en el ejemplo anterior. La única diferencia es que ahora renombramos* `ppal.cpp` *por* `ordena1.cpp`*: recordar que la regla implícita pare enlazar funciona correctamente en proyectos con múltiples ficheros objeto si uno de ellos tiene el mismo nombre que el ejecutable (queremos que el ejecutable se llame* `ordena1` *y no* `ppal`*). Observar que se ha sustituido la regla explícita para la generación del ejecutable por una regla implícita (sin órdenes) para el enlazador, especificando únicamente la lista de dependencias.*

```
# Fichero: makefil6.v3 (Version 6 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ordena1.o": codigo objeto del programa principal (de "ordena1.cpp").
#      (este fichero era antes ppal.cpp)
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").
#
# Novedad: uso reglas implicitas para enlazar.
#
# Para la fase de enlazado, se aplica la regla implicita:
#          $(CC) $(LDFLAGS) $(LOADLIBS)

CXX = g++
CC = $(CXX)
CPPFLAGS =
CXXFLAGS = -I$(INCLUDE)

LDFLAGS =
LOADLIBS =

INCLUDE = ./include

ordena1: ordena1.o vec_ES.o ordena.o

ordena1.o : $(INCLUDE)/vec_ES.h $(INCLUDE)/ordena.h
vec_ES.o: $(INCLUDE)/vec_ES.h
ordena.o: $(INCLUDE)/ordena.h
```

Ahora, la ejecución de: `make -f makefil6.v3` *provoca la ejecución, por este orden, de las siguientes tareas:*

```
g++ -I./include   -c ordena1.cpp -o ordena1.o
g++ -I./include   -c vec_ES.cpp -o vec_ES.o
g++ -I./include   -c ordena.cpp -o ordena.o
g++   ordena1.o vec_ES.o ordena.o   -o ordena1
```

Como ocurre con `makefil5.v3`*, cualquier modificación en un fichero cabecera desencadena la actualización del objeto dependiente y del ejecutable final.*

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
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -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.

Ejemplo 19 *El ejemplo que ilustra el uso de reglas patrón está basado en el ejemplo anterior: usa reglas implícitas para compilar y enlazar.*

```
# Fichero: makefil7.v3 (Version 7 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").
#
# Novedad: uso de reglas implicitas patron.

CXX = g++
CC = $(CXX)
CPPFLAGS =
CXXFLAGS = -I$(INCLUDE)

LDFLAGS =
LOADLIBS =

INCLUDE = ./include

ordena1 : ordena1.o vec_ES.o ordena.o

ordena1.o : $(INCLUDE)/vec_ES.h $(INCLUDE)/ordena.h
```

```
%.o : %.cpp $(INCLUDE)/%.h
   g++ -c $(CXXFLAGS) $< -o $@
```

Las dos primeras reglas son las del ejemplo anterior: la primera es una regla implícita para el enlazador y la segunda regla es una regla implícita para el compilador (ver discusión en el ejemplo anterior).

La tercera regla es una regla implícita patrón que indica que cada fichero con extensión `.o` *depende de los respectivos ficheros con extensión* `.cpp` *y* `.h` *(éstos últimos se encontrarán en el subdirectorio* `include`*). Nótese que el orden en que aparecen las dependencias en la regla es importante pues* `$<` sustituye a la primera dependencia. *El objeto de esta regla es especificar cómo se construirán los ficheros objeto con la regla anterior. La construcción de los ficheros destino se realiza ejecutando la orden asociada a la regla implícita patrón:*

```
g++ -c $(CXXFLAGS) $< -o $@
```

en el que se llama a `g++` *para que compile sin enlazar. En este caso se utiliza las macros* `$<` *y* `$@` *cuyo significado es:*

- `$<` *Sustituye completamente el nombre del fichero dependiente.*
- `$@` *Sustituye al nombre del fichero destino.*

Observar que `ordena1.o` *puede construirse con dos reglas diferentes. La que se aplica es la primera que aparece en el fichero makefile, en este caso:*

```
ordena1.o : $(INCLUDE)/vec_ES.h $(INCLUDE)/ordena.h
```

porque queremos expresar la dependencia de `ordena1.o` *respecto de los dos ficheros de cabecera (además de la dependencia implícita respecto del fuente* `ordena1.cpp`*).*

Tras esta discusión veamos qué tareas desencadena la ejecución de `make` *sobre* `makefil7.v3`*:*

```
g++ -I./include   -c ordena1.cpp -o ordena1.o
g++ -c -I./include vec_ES.cpp -o vec_ES.o
g++ -c -I./include ordena.cpp -o ordena.o
g++   ordena1.o vec_ES.o ordena.o   -o ordena1
```

La modificación de un fichero cabecera, por ejemplo `vec_ES.h` *hará que al llamar a* `make` *se ejecuten estas tareas:*

```
g++ -I./include   -c ordena1.cpp -o ordena1.o
g++ -c -I./include vec_ES.cpp -o vec_ES.o
g++   ordena1.o vec_ES.o ordena.o   -o ordena1
```

Observar que la primera se realiza al haber anulado la actualización de `ordena1.o` *por la última regla al haber introducido la segunda que utiliza la regla implícita de compilación:*

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

Ejercicio

1. Crear los ficheros make descritos anteriormente y probarlos.
2. Implementar nuevos ficheros make para el programa de la sección 8.3 (descompuesto en ejercicios anteriores) que contenga reglas implícitas.
3. Hacer que el makefile ordene la compilación de todos los módulos con `-Wall` y `-O3` (consejo: modificar la variable `CPPFLAGS`).

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*.

6.5.3. 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.
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`.

Ejemplo 20 *El ejemplo que ilustra el uso de directivas en un makefile es una ampliación del ejemplo 16, en el que se utilizaba una macro (*`DESTDIR`*) en la llamada a* `make` *en la que se especifica el directorio donde se guarda el ejecutable.*

En dicho ejemplo, la primera versión intenta crear el directorio con la orden `mkdir DESTDIR`*. Escrita de esta forma,* `make` *interrumpe el procesamiento del fichero makefile si: a) no se especificó la macro en la llamada a* `make` *ó b) se especificó la macro pero el directorio ya existe. Para remediar este problema se presentó la opción de preceder la orden con el prefijo* `-` *para que en el caso de que hubiera algún problema se continuara el procesamiento del fichero makefile. Finalmente, se presentó la opción de especificar mediante una macro (*`BASE`*) el directorio en el que se guarda el ejecutable si no se especifica el valor de la macro* `DESTDIR`*.*

Ahora podemos evaluar en un fichero makefile si una macro se encuentra definida y construir ficheros makefile que puedan dirigir el flujo de procesamiento de `make`*. Esta alternativa resulta, sin duda, la más profesional.*

Observar cómo en el fichero `makefil8.v3` *se utilizan* todos *los prefijos de órdenes y macros por defecto.*

```
# Fichero: makefil8.v3 (Version 8 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").
#
# Novedad: uso de directivas condicionales en ficheros makefile

CXXFLAGS = -I$(INCLUDE)

INCLUDE = ./include

ordena1 : ordena1.o vec_ES.o ordena.o
ifndef DESTDIR
   @echo  Error: Falta especificar opcion DESTDIR=directorio
else
   -mkdir $(DESTDIR)
   @echo Creando $@ a partir de: $^
   g++ $^ -o $(DESTDIR)/$@
   @echo
endif

ordena1.o : ordena1.cpp $(INCLUDE)/*.h
   @echo Creando $@ a partir de: $^
   g++ -c $(CXXFLAGS) ordena1.cpp -o $@
   @echo

%.o : %.cpp $(INCLUDE)/%.h
   @echo Creando $@ a partir de: $^
   g++ -c $(CXXFLAGS) $< -o $@
   @echo
```

Vamos a analizar con detenimiento este fichero makefile. Dos apuntes generales:

- *Se ha añadido una instrucción* `echo` *asociada a cada regla para informar sobre la tarea que está ejecutando, concretamente, para cada destino muestra el mensaje:* `Creando` destino `a partir de:` lista de dependencia completa asociada a destino
- *Todos los ficheros objetos se construyen llamando de forma explícita al compilador en lugar de utilizar la regla implícita* `$(CC) -c $(CPPFLAGS) $(CFLAGS)`*.*

La última regla, que indica cómo construir los objetos `ordena.o` *y* `vec_ES.o` *a partir de los ficheros fuente y cabecera correspondientes es idéntica a la presentada en el ejemplo anterior (ejemplo 19), por lo que no la vamos a comentar.*

La regla que indica cómo construir el objeto `ordena1.o` *se ha modificado respecto a la versión anterior. Ahora se llama de forma explícita al compilador en lugar de utilizar la regla implícita.*

La principal novedad de este makefile reside en las instrucciones asociadas a la regla que construye el ejecutable `ordena1`*. La regla es idéntica: construye* `ordena1` *a partir de los objetos* `ordena1.o`*,* `vec_ES.o` *y* `ordena.o`*. Sin ebargo, las tareas que se ejecutan dependen de la evaluación de la directiva* `ifndef` *sobre la macro* `DESTDIR`*. Así,*

a) Si `DESTDIR` *no está definida (*`ifndef DESTDIR` *es* verdad*), se muestra un mensaje de error (*el que nosotros especificamos*) y termina el procesamiento del fichero makefile, ya que ésta es la última regla que se procesa.*

b) Si está definida, (`ifndef DESTDIR` *es* falso*) ejecuta las tareas de la parte* `else`*: intenta crear el directorio y genera el ejecutable llamando a* `g++` *de forma explícita (éste invocará al enlazador).*

A continuación mostreremos unos ejemplos de ejecución de `make` *sobre este fichero makefile.*

En primer lugar, veamos las tareas que se ejecutan si no se define la macro `DESTDIR` *en la llamada a* `make`*. Deberá crear únicamente los ficheros objeto y mostrar un mensaje de error:*

```
% make -f makefil8.v3

Creando ordena1.o a partir de: ordena1.cpp include/ordena.h include/vec_ES.h
g++ -c -I./include ordena1.cpp -o ordena1.o

Creando vec_ES.o a partir de: vec_ES.cpp include/vec_ES.h
g++ -c -I./include vec_ES.cpp -o vec_ES.o

Creando ordena.o a partir de: ordena.cpp include/ordena.h
g++ -c -I./include ordena.cpp -o ordena.o

Error: Falta especificar opcion DESTDIR=directorio
```

Si se especifica el valor de la macro `DESTDIR` *y este directorio* **no existe***, todo funcionará como esperamos (creará el directorio y guardará el ejecutable en él):*

```
% make -f makefil8.v3 DESTDIR=bin

Creando ordena1.o a partir de: ordena1.cpp include/ordena.h include/vec_ES.h
g++ -c -I./include ordena1.cpp -o ordena1.o

Creando vec_ES.o a partir de: vec_ES.cpp include/vec_ES.h
g++ -c -I./include vec_ES.cpp -o vec_ES.o

Creando ordena.o a partir de: ordena.cpp include/ordena.h
g++ -c -I./include ordena.cpp -o ordena.o

mkdir bin
Creando ordena1 a partir de: ordena1.o vec_ES.o ordena.o
g++ ordena1.o vec_ES.o ordena.o -o bin/ordena1
```

mientras que si el directorio **existe***, al estar "protegida" la instrucción* `mkdir` *con el prefijo* `-`*,* `make` *sigue procesando tareas, aunque haya fallado al construir el directorio especificado. Por ejemplo, supongamos que el directorio* `bin2` *ya existe:*

```
% make -f makefil8.v3 DESTDIR=bin2
```

```
Creando ordena1.o a partir de: ordena1.cpp include/ordena.h include/vec_ES.h
g++ -c -I./include ordena1.cpp -o ordena1.o

Creando vec_ES.o a partir de: vec_ES.cpp include/vec_ES.h
g++ -c -I./include vec_ES.cpp -o vec_ES.o

Creando ordena.o a partir de: ordena.cpp include/ordena.h
g++ -c -I./include ordena.cpp -o ordena.o

mkdir bin2
mkdir: cannot make directory ‘bin2’: File exists
make: [ordena1] Error 1 (ignored)
Creando ordena1 a partir de: ordena1.o vec_ES.o ordena.o
g++ ordena1.o vec_ES.o ordena.o -o bin2/ordena1
```

Recordar que hemos insistido en la conveniencia de indicar expícitamente la depenencia respecto de los ficheros de cabecera*. Ilustraremos los beneficios de esta manera de indicar dependencias con un ejemplo. Si decidimos cambiar la constante* `MAX` *de* `vec_ES.h` *por 15, cambiaremos la línea de su definición por la siguiente:*

```
const int MAX=15;    // Tamanio del vector
```

La reconstrucción del ejecutable se hará con el fichero makefile anterior. Supongamos que se guarda de nuevo en el directorio `bin2`*. Observar cómo sólo se reconstruye lo estrictamente necesario (por ejemplo,* `ordena.o` *no se modifica porque no depende de* `vec_ES.h`*):*

```
% make -f makefil8.v3 DESTDIR=bin2

Creando ordena1.o a partir de: ordena1.cpp include/ordena.h include/vec_ES.h
g++ -c -I./include ordena1.cpp -o ordena1.o

Creando vec_ES.o a partir de: vec_ES.cpp include/vec_ES.h
g++ -c -I./include vec_ES.cpp -o vec_ES.o

mkdir bin2
mkdir: cannot make directory ‘bin2’: File exists
make: [ordena1] Error 1 (ignored)
Creando ordena1 a partir de: ordena1.o vec_ES.o ordena.o
g++ ordena1.o vec_ES.o ordena.o -o bin2/ordena1
```

Como alternativa al fichero makefile anterior, presentamos ahora una modificación que permite la generación del ejecutable aunque no se especifique el directorio de destino en la llamada a `make`*. Se establece un directorio por defecto (en este caso, el actual) donde guardar el ejecutable. Para ello utilizamos, al igual que en el ejemplo 16 una macro (*`BASE`*) que inicializamos con el directorio actual, que establecemos como directorio por defecto para guardar el ejecutable si no se especifica el valor de la macro* `DESTDIR`*.*

```
# Fichero: makefil9.v3 (Version 9 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").
#
# Novedad: uso de directivas condicionales en ficheros makefile
```

```
CXXFLAGS = -I$(INCLUDE)

INCLUDE = ./include
BASE = .

ordena1 : ordena1.o vec_ES.o ordena.o
ifndef DESTDIR
   @echo  Error: Falta especificar opcion DESTDIR=directorio
   @echo  Por defecto, se establece al directorio actual.
   @echo
endif

   -mkdir  $(BASE)$(DESTDIR)
   @echo Creando $@ a partir de: $^
   g++ $^ -o $(BASE)$(DESTDIR)/$@
   @echo

ordena1.o : ordena1.cpp $(INCLUDE)/*.h
   @echo Creando $@ a partir de: $^
   g++ -c $(CXXFLAGS) ordena1.cpp -o $@
   @echo

%.o : %.cpp $(INCLUDE)/%.h
   @echo Creando $@ a partir de: $^
   g++ -c $(CXXFLAGS) $< -o $@
   @echo
```

Ahora, la ejecución de `make` *especificando el directorio de destino funciona exactamente igual que con* `makefil8.v3` *ya que, por ejemplo, si* `DESTDIR` *se establece con el valor* `bin2`*, intenta crear el directorio* `.bin2` *(subdirectorio* `bin2` *del directorio actual) y construye el ejecutable con la orden:*

```
g++ ordena1.o vec_ES.o ordena.o -o .bin2/ordena1
```

mientras que si se ejecuta sin especificar el valor de `DESTDIR`*, intenta crear el directorio* `.` *(actual) y construye el ejecutable con la orden:*

```
g++ ordena1.o vec_ES.o ordena.o -o ./ordena1
```

7. Uso y construcción de bibliotecas con el programa *ar*

7.1. Introducción

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 una de estas 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** es un fichero que 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, claro está). Si además nuestros proyectos se matienen 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 la función y no su definición.

En las secciones 1 y 2 ya hablamos de forma muy general de las bibliotecas, justificando la necesidad de usar bibliotecas construidas por el programador. En esta sección mostraremos con detalle la forma en que se crean, eliminan y actualizan, utilizando el programa `ar`.

7.2. Estructura de una biblioteca

Una biblioteca se estructura internamente como un conjunto de módulos objeto (extensión `.o`). Cada uno de estos módulos puede ser el resultado de la compilación de un fichero de código fuente (extensión `.cpp`) que puede contener, a su vez, 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`.Este prefijo es necesario si queremos enlazar usando la opción -l del compilador, así (g++ ... -lm) se enlazarían los módulos indicados con la biblioteca matemática libm.a

Veamos, con un ejemplo, cómo se estructura una biblioteca. La biblioteca `libejemplo.a` contiene un conjunto de 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 `funcs.cpp`. Este caso se ilustra gráficamente en la figura 9.

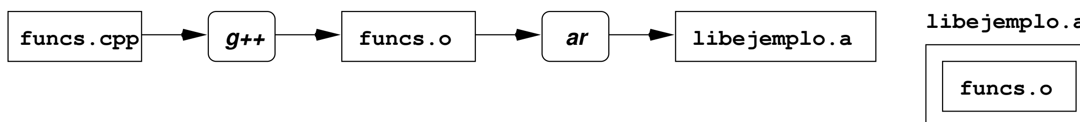


Figura 9: Construcción de una biblioteca a partir de un único módulo objeto (caso 1)

```
/** @file funcs.cpp
@brief 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`, `fun02.o`, ..., `fun10.o` resultado de la compilación de 10 ficheros fuente, por ejemplo, `fun01.cpp`, `fun02.cpp`, ..., `fun10.cpp` que contienen, cada uno, la definición de una única función. Este caso se ilustra gráficamente en la figura 10.

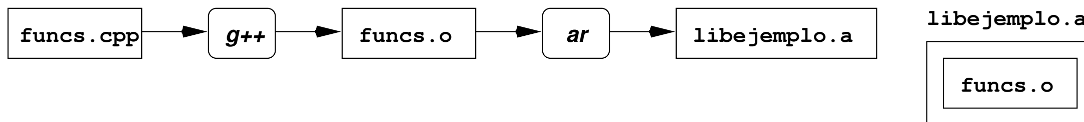


Figura 10: Construcción de una biblioteca a partir de varios módulos objeto (caso 2)

```
/** @file fun01.cpp
@brief Contiene la definicion de "funcion_1()"
*/

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

/** @file fun02.cpp
@brief Contiene la definicion de "funcion_2()"
*/
char *funcion_2 (char *s, char *t)
{
   ...........
}
   ...........

/** @file fun10.cpp
@brief Contiene la definicion de "funcion_10()"
*/

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

Una vez construida la biblioteca, y para generar el fichero ejecutable que utiliza una función de ésta, el enlazador actúa de la siguiente manera: **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 programa 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" // prototipos de las funciones de "libejemplo.a"

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

***Cada biblioteca llevará asociado un fichero de cabecera** que contendrá los prototipos de las funciones que se ofrecen (**funciones públicas**) que actúa de interface entre las funciones de la biblioteca y los programas que usan 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 considerar que la parte de la biblioteca que se enlaza al código objeto del programa principal es el módulo objeto **completo** en el que se encuentra la función de biblioteca usada. En la figura 11 mostramos cómo se construye un ejecutable a partir de un módulo objeto (`ppal.o`) que contiene la función `main()` y una biblioteca (`libejemplo.a`).

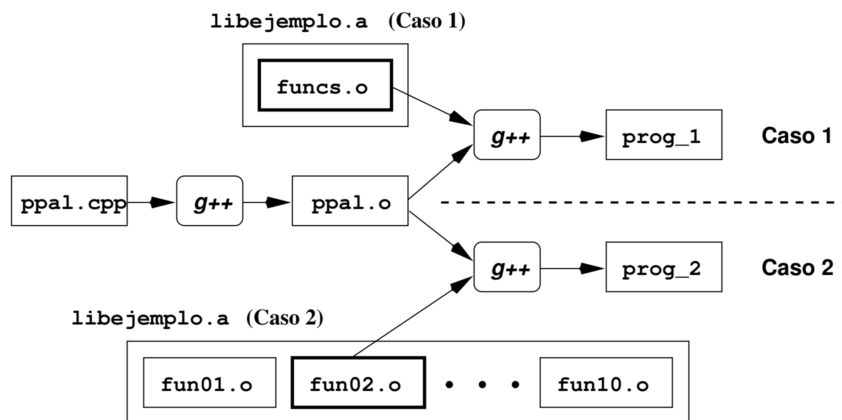


Figura 11: Construcción de un ejecutable que usa una función de biblioteca. Caso 1: biblioteca formada por un módulo objeto. Caso 2: biblioteca formada por 10 módulos objeto

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` 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` 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.

En cualquier caso, como **usuario** de la biblioteca se actúa de la misma manera en ambas situaciones. La secuencia de tareas para generar un ejecutable (suponiendo que la biblioteca ya existe) será:

1. Generación del objeto `ppal.o`: `g++ -c ppal.cpp -o ppal.o`
2. Generación del ejecutable `prog_x`: `g++ ppal.o libejemplo.a -o prog_x`

Como **diseñador** de la biblioteca sí debemos tener en cuenta la estructura que vamos a adoptar para ésta, considerando que 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[4]. En la sección 7.4.2 ilustramos con ejemplos el efecto del número de módulos objeto de una biblioteca en el tamaño del programa ejecutable.

7.3. Gestión de bibliotecas con 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:

- *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.
- *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.
  - `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 anlazador sepa cómo enlazarla*. Este modificador puede emplearse acompañando a una operación o por si solo.
  - `v` **Verbose**. Muestra información sobre la operación realizada.

7.4. Ejemplos

En esta sección mostramos cuatro ejemplos básicos (sección 7.4.1) sobre el uso de bibliotecas y dos ejemplos avanzados (sección 7.4.2) sobre el efecto del número de funciones en los módulos objeto que forman la biblioteca. Finalmente, en la sección 7.4.3 hacemos una recopilación de la materia expuesta en este capítulo y el anterior, a través de complejos ficheros makefile en los que mostramos el uso de macros en la llamada a `make`, sustituciones en macros de ficheros makefile, reglas implícitas y métodos avanzados de gestión de bibliotecas.

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

7.4.1. Ejemplos básicos

Ejemplo 21 *Este ejemplo muestra cómo crear una biblioteca (*`libadic.a`*) que consta de un único módulo objeto (*`adicion.o`*). Éste se construye a partir de* `adicion.cpp` *que contiene únicamente dos funciones.*

```
/*******************************************************/
// Fichero: adicion.cpp
// Contienes las funciones "suma()" y "resta()"
/*******************************************************/

#include "adicion.h"

int suma  (int a, int b) {return (a+b);}
int resta (int a, int b) {return (a-b);}
```

Las funciones de esta biblioteca se usan en el programa `demo1.cpp`*.*

```
/*******************************************************/
// Fichero: demo1.cpp
// Usa las funciones "suma()" y "resta()" de "libadic.a"
/*******************************************************/

#include <iostream>
#include "adicion.h"
using namespace std;

int main () {
   cout << suma (2,5) << endl;
   cout << resta(2,5) << endl;
   return (0);
}
```

La biblioteca lleva asociado su fichero cabececera correspondiente, en este caso, `adicion.h` *que contiene los prototipos de las funciones públicas de la biblioteca (en este caso, todas).*

```
/*******************************************************/
// Fichero: adicion.h
// Fichero de cabecera asociado a la biblioteca "libadic.a"
/*******************************************************/

#ifndef ADICION
#define ADICION

int suma  (int, int);
int resta (int, int);

#endif
```

La creación de la biblioteca, así como la generación del ejecutable se especifica detalladamente en el fichero `makefil1.mak`*. En la figura 12 mostramos el diagrama de dependencias entre módulos. Finalmente, comentaremos los aspectos más interesantes de este fichero makefile.*

```
# Fichero: makefil1.mak
# Ejemplo de fichero makefile que crea una biblioteca con un modulo
# objeto y lo enlaza con otro modulo objeto para formar un ejecutable.
# Ilustra como usar la opcion -L en g++ y como especificar los ficheros
```

```
# de biblioteca que se uasr para el enlace.

INCLUDE = .
LIB     = .

demo1: demo1.o libadic.a
        g++ -L$(LIB) -o demo1 demo1.o -ladic

demo1.o: demo1.cpp adicion.h
        g++ -c -o demo1.o -I$(INCLUDE) demo1.cpp

libadic.a: adicion.o
        ar rvs $(LIB)/libadic.a adicion.o

adicion.o: adicion.cpp adicion.h
        g++ -c -o adicion.o -I$(INCLUDE) adicion.cpp
```

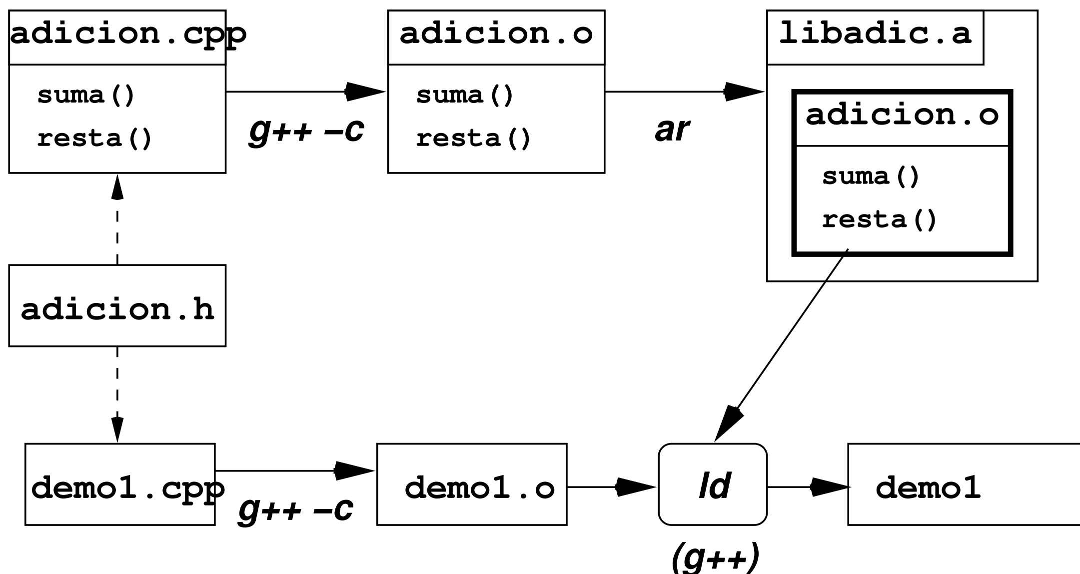


Figura 12: Construcción del ejecutable *demo1*

Observar con detenimiento la primera regla:

```
demo1: demo1.o libadic.a
        g++ -L$(LIB) -o demo1 demo1.o -ladic
```

que indica cómo generar el ejecutable `demo1`*.*

- *Aunque se llame a* `g++` *para construir el ejecutable, éste invoca al enlazador* `ld` *ya que el destino es un fichero ejecutable.*
- *En esta llamada se especifica que se incluya al directorio actual en la lista de búsqueda de los directorios de bibliotecas (opción* `-L`*).* Es muy recomendable especificar la opción `-L` cuando se van a enlazar bibliotecas propias.
- *En la lista de dependencias aparecen los ficheros* `demo1.o` *y* `libadic.a`*, mientras que en la llamada a* `g++` *no aparece explícitamente* `libadic.a`*. Los ficheros de biblioteca que se usan en el enlace se especifican a* `g++` *con la opción* `-l`fichero *(ver sección 4.3 del apéndice 4), en*

este caso, `-ladic`*. El enlazador buscará en los directorios de bibliotecas (entre los que están los especificados con* `-L`*) un fichero de biblioteca llamado* `lib`fichero`.a` *(en este caso,* `libadic.a`*) y lo usa para enlazarlo.*

Comentaremos ahora la regla que crea la biblioteca:

```
libadic.a: adicion.o
        ar rvs $(LIB)/libadic.a adicion.o
```

La biblioteca `libadic.a` *depende del módulo objeto* `adicion.o`*. La creación de la biblioteca se hace con* `ar` *mediante la operación* `r` *(adición o reemplazo) de forma que si no existe la biblioteca, la crea añdiendo el módulo* `adicion.o`*. Si existe, reemplaza ese módulo por la nueva versión. El modificador* `s` *hace que se actualice el índice de los módulos de la biblioteca. El modificador* `v` *hace que cuando se procese la orden:*

```
ar rvs ./libadic.a adicion.o
```

por primera vez, se muestre en la consola la línea informativa:

```
a - adicion.o
```

indicando que se ha añadido *el módulo* `adicion.o`*. Si se modificara* `adicion.cpp` *y/o* `adicion.h` *y se ejecutara de nuevo* `make`*, al procesar esa orden mostraría la línea informativa:*

```
r - adicion.o
```

indicando que se ha reemplazado *el módulo* `adicion.o` *(Hay que tener en cuenta que la modificación de* `adicion.cpp` *y/o* `adicion.h` *implica la reconstrucción de* `adicion.o` *y la consiguiente reconstrucción de la biblioteca y del ejecutable).*

La ejecución de la orden:

```
% make -f makefil1.mak
```

desencadena las siguientes tareas (por este orden):

```
g++ -c -o demo1.o -I. demo1.cpp
g++ -c -o adicion.o -I. adicion.cpp
ar rvs ./libadic.a adicion.o
a - adicion.o
g++ -L. -o demo1 demo1.o -ladic
```

y cuando se modifica `adicion.cpp` *y/o* `adicion.h`*, y se ejecuta de nuevo* `make`*:*

```
g++ -c -o demo1.o -I. demo1.cpp
g++ -c -o adicion.o -I. adicion.cpp
ar rvs ./libadic.a adicion.o
r - adicion.o
g++ -L. -o demo1 demo1.o -ladic
```

Ejemplo 22 *Este ejemplo muestra cómo crear dos bibliotecas:* `libadic.a` *y* `libprod.a`*. Ambas constan de un único módulo objeto (*`adicion.o` *y* `producto.o`*, respectivamente). El primer módulo objeto se construye a partir del fuente* `adicion.cpp` *(ver ejemplo anterior) y el segundo módulo objeto se construye a partir del fuente* `producto.cpp`*. Ambos ficheros contienen dos funciones.*

```
/*******************************************************/
// Fichero: producto.cpp
// Contienes las funciones "multiplica()" y "divide()"
/*******************************************************/

#include "producto.h"

int multiplica (int a, int b) {return (a*b);}
int divide (int a, int b) {return (a/b);}
```

Las dos bibliotecas se usan en el programa `demo2.cpp`. *Este programa usa las dos funciones de* `libadic.a` *y una de* `libprod.a`.

```
/*********************************************************/
// Fichero: demo2.cpp
// Usa las funciones "suma()" y "resta()" de "libadic.a"
// y la funcion "multiplica()" de "libprod.a"
/*********************************************************/

#include <iostream>
#include "adicion.h"
#include "producto.h"
using namespace std;

int main ()
{
   cout << suma (2,5) << endl;
   cout << resta(2,5) << endl;
   cout << multiplica(2,5) << endl;
   return (0);
}
```

De nuevo, cada biblioteca lleva asociado su fichero cabececera correspondiente, en este caso, `adicion.h` *(ver ejemplo anterior) y* `producto.h`.

```
/*********************************************************/
// Fichero: producto.h
// Fichero de cabecera asociado a la biblioteca "libprod.a"
/*********************************************************/

#ifndef PRODUCTO
#define PRODUCTO

int multiplica  (int, int);
int divide      (int, int);

#endif
```

La creación de las bibliotecas, así como la generación del ejecutable se especifica en el fichero `makefil2.mak`. *En la figura 13 mostramos el diagrama de dependencias entre módulos y cómo se construye el ejecutable* `demo2`.

```
# Fichero: makefil2.mak
# Ejemplo de fichero makefile que crea dos bibliotecas con un modulo objeto
# cada una y las enlaza con otro modulo objeto para formar un ejecutable.

INCLUDE = .
LIB     = .

demo2: demo2.o libadic.a libprod.a
        g++ -L$(LIB) -o demo2 demo2.o -ladic -lprod

demo2.o: demo2.cpp adicion.h producto.h
        g++ -c -o demo2.o -I$(INCLUDE) demo2.cpp
```

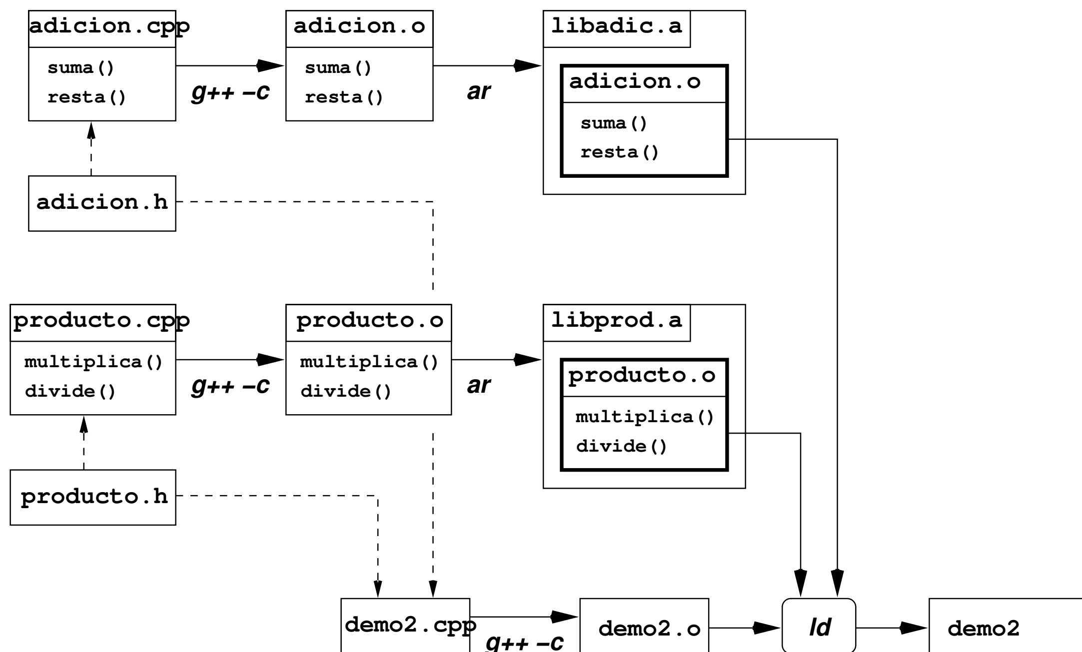


Figura 13: Construcción del ejecutable *demo2*

```
adicion.o: adicion.cpp adicion.h
        g++ -c -o adicion.o -I$(INCLUDE) adicion.cpp

producto.o: producto.o producto.h
        g++ -c -o producto.o -I$(INCLUDE) producto.cpp

libadic.a: adicion.o
        ar rvs $(LIB)/libadic.a adicion.o

libprod.a: producto.o
        ar rvs $(LIB)/libprod.a producto.o
```

Sobre este ejemplo pueden hacerla las misma consideraciones que sobre el ejemplo anterior sólo que ahora se utilizan dos bibliotecas en la fase de enlace. Como ocurre en ese ejemplo, ambas bibliotecas constan de un único módulo objeto por lo que el ejecutable se construye, realmente, enlazando los módulos objeto `demo2.o`, `adicion.o` *y* `producto.o`.

Ejemplo 23 *Este ejemplo muestra cómo crear una biblioteca llamada* `libtodos.a` *que consta de un único módulo objeto llamado* `todos.o`. *Este módulo objeto se construye a partir del fuente* `todos.cpp` *que contiene cuatro funciones, las dos que constituyen* `adicion.cpp` *y las dos que constituyen* `producto.cpp`.

```
/**************************************************************************/
// Fichero: todos.cpp
// Contienes las funciones "suma()", "resta()", "multiplica()" y "divide()"
/**************************************************************************/

#include "todos.h"

int suma  (int a, int b) {return (a+b);}
int resta (int a, int b) {return (a-b);}
```

```
int multiplica (int a, int b) {return (a*b);}
int divide (int a, int b) {return (a/b);}
```

El fichero cabececera asociado a la biblioteca será `todos.h`*.*

```
/*********************************************************/
// Fichero: todos.h
// Fichero de cabecera asociado a la biblioteca "libtodos.a"
/*********************************************************/

#ifndef TODOS
#define TODOS

int suma        (int, int);
int resta       (int, int);
int multiplica  (int, int);
int divide      (int, int);

#endif
```

Esta biblioteca se usa en el programa `demo3.cpp` *que usa las cuatro funciones de la biblioteca.*

```
/**************************************************************************/
// Fichero: demo3.cpp
// Usa: "suma()", "resta()", "multiplica()" y "divide()" de "libtodos.a"
/**************************************************************************/

#include <iostream>
#include "todos.h"
using namespace std;

int main ()
{
   cout << suma (2,5) << endl;
   cout << resta(2,5) << endl;
   cout << multiplica(2,5) << endl;
   cout << divide (2,5) << endl;

   return (0);
}
```

La creación de la biblioteca, así como la generación del ejecutable se especifica detalladamente en el fichero `makefil3.mak`*. En la figura 14 mostramos el diagrama de dependencias entre módulos y cómo se construye el ejecutable* `demo3`*.*

```
# Fichero: makefil3.mak
# Ejemplo de fichero makefile que crea una biblioteca con un modulo
# objeto y lo enlaza con otro modulo objeto para formar un ejecutable.

INCLUDE = .
LIB     = .

demo3: demo3.o libtodos.a
        g++ -L$(LIB) -o demo3 demo3.o -ltodos
```

```
demo3.o: demo3.cpp todos.h
        g++ -c -o demo3.o -I$(INCLUDE) demo3.cpp

libtodos.a: todos.o
        ar rvs $(LIB)/libtodos.a todos.o

todos.o: todos.cpp todos.h
        g++ -c -o todos.o -I$(INCLUDE) todos.cpp
```

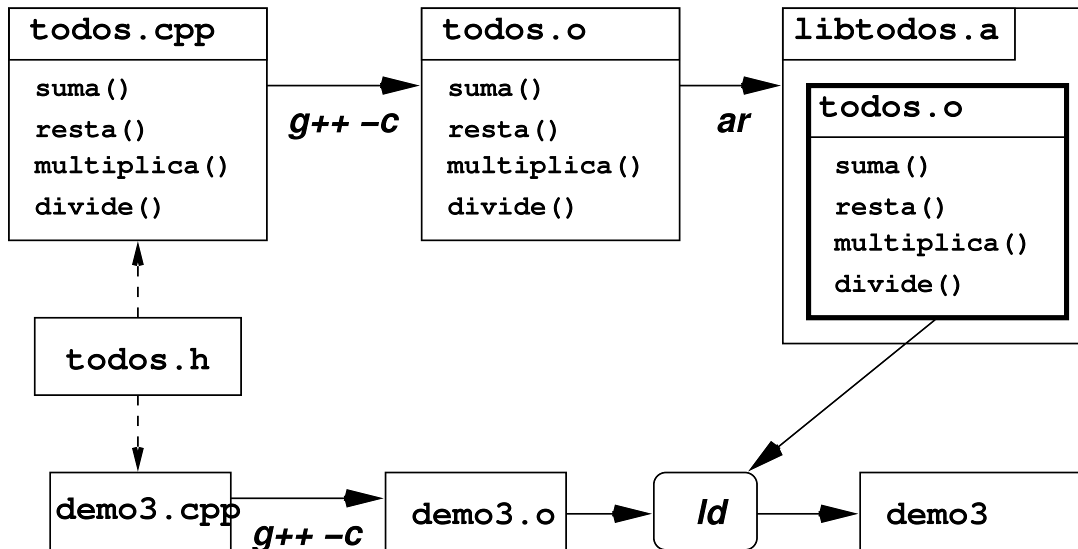


Figura 14: Construcción del ejecutable *demo3*

La estructura y funcionamiento de este fichero makefile es similar al de `makefil1.mak` *del ejemplo 21 de este apéndice.*

Ejemplo 24 *Este ejemplo muestra cómo crear una biblioteca llamada* `libmod2.a` *que consta de* **dos** *módulos objeto (los conocidos* `adicion.o` *y* `producto.o`*). Estos módulos objetos se construyen a partir de los fuentes, ya conocidos,* `adicion.cpp` *y* `producto.cpp`*. Para minimizar el númro de cambios, el fichero de cabecera asociado a la biblioteca será* `todos2.h` *cuyo contenido es el siguiente:*

```
/**********************************************************/
// Fichero: todos2.h
// Fichero de cabecera asociado a la biblioteca "libmod2.a"
/**********************************************************/

#ifndef TODOS
#define TODOS

#include "adicion.h"
#include "producto.h"

#endif
```

El programa `demo4.cpp` *usa las cuatro funciones de la biblioteca* `libmod2.a`*.*

```
/*************************************************************************/
// Fichero: demo4.cpp
```

```
// Usa: "suma()", "resta()", "multiplica()" y "divide()" de "libmod2.a"
/***************************************************************************/

#include <iostream>
#include "todos2.h"
using namespace std;

int main ()
{
   cout << suma (2,5) << endl;
   cout << resta(2,5) << endl;
   cout << multiplica(2,5) << endl;
   cout << divide (2,5) << endl;

   return (0);
}
```

El fichero makefile correspondiente es `makefil4.mak`*. En la figura 15 mostramos el diagrama de dependencias entre módulos y cómo se construye el ejecutable* `demo4`*.*

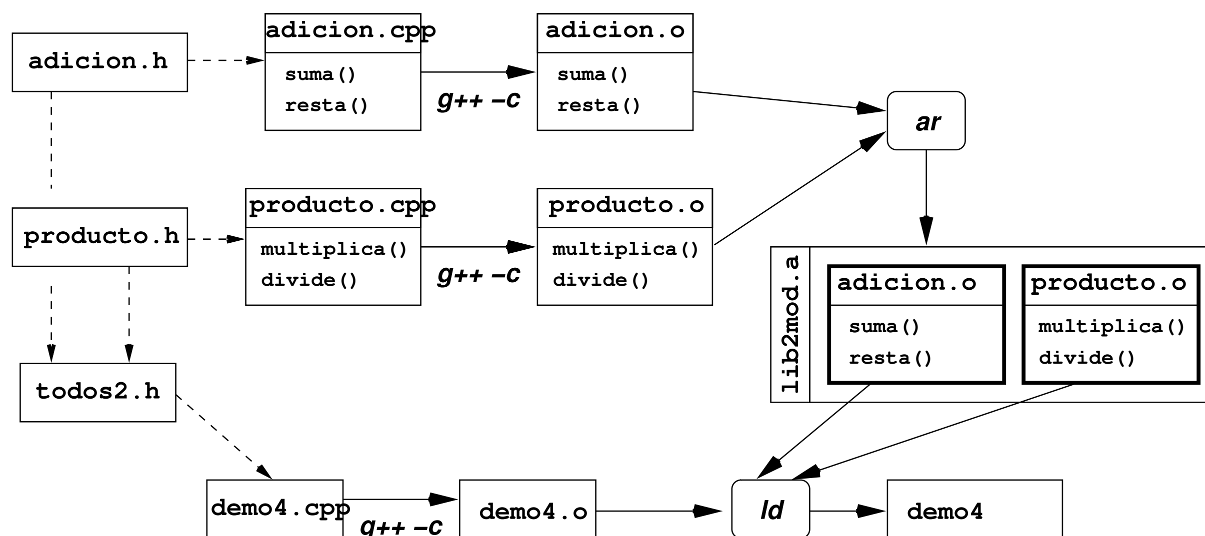


Figura 15: Construcción del ejecutable *demo4*

```
# Fichero: makefil4.mak
# Ejemplo de fichero makefile que crea una bibliotecas con dos modulos
# objeto y la enlaza con otro modulo objeto para formar un ejecutable.

INCLUDE = .
LIB     = .

demo4: demo4.o lib2mod.a
        g++ -L$(LIB) -o demo4 demo4.o -l2mod

demo4.o: demo4.cpp todos2.h
        g++ -c -o demo4.o -I$(INCLUDE) demo4.cpp

adicion.o: adicion.cpp adicion.h
        g++ -c -o adicion.o -I$(INCLUDE) adicion.cpp

producto.o: producto.cpp producto.h
        g++ -c -o producto.o -I$(INCLUDE) producto.cpp

lib2mod.a: adicion.o producto.o
        ar rvs lib2mod.a adicion.o producto.o
```

Si se se ejecuta la orden `make -f makefil4.mak` *se desencadenan las siguientes tareas:*

```
g++ -c -o demo4.o -I. demo4.cpp
g++ -c -o adicion.o -I. adicion.cpp
g++ -c -o producto.o -I. producto.cpp
ar rvs ./lib2mod.a adicion.o producto.o
a - adicion.o
a - producto.o
g++ -L. -o demo4 demo4.o -l2mod
```

Observar cómo la ejecución de `ar` *produce dos líneas de mensajes indicando que se han añadido los módulos* `adicion.o` *y* `producto.o` *a la biblioteca* `lib2mod.a`*.*

Ejercicio

1. A partir de la descomposición del programa de la sección 8.3 sobre la que se ha venido trabajando hasta ahora, construir bibliotecas con los módulos en los que se haya descompuesto y diseñar un makefile que incluya la gestión de estas bibliotecas, tal y como muestran los ejemplos anteriores.

7.4.2. Ejemplos avanzados

Como se comentó anteriormente la estructura de una biblioteca determina el tamaño del programa ejecutable que se construye usando esa biblioteca. Como venimos indicando, una vez construida la biblioteca, y para generar un módulo ejecutable, la biblioteca debe enlazarse obligatoriamente con el módulo objeto que contiene la función `main()`.

Ejemplo 25 *Sobre un mismo programa fuente generaremos tres ejecutables. Se construirán de forma diferente, enlazando tres bibliotecas distintas, las conocidas* `libadic.a`*,* `libtodos.a` *y* `lib2mod.a`*. Los tres ficheros fuente sólo se diferenciarán en la línea* `#include`*, de forma que incluirán el fichero de cabecera asociado a la biblioteca que usan. La estructura será la siguiente:*

```
/**********************************************************************/
// Fichero: demo{5,6,7}.cpp
// Usa: "suma()" de {"libadic.a", "libtodos.a", "lib2mod.a"}
/**********************************************************************/

#include <iostream>
#include {"adicion.h", "todos.h", "todos2.h"}
using namespace std;

int main ()
{
   cout << suma (2,5) << endl;
   return (0);
}
```

y se interpreta como sigue:

1. `demo5.cpp` *usa la función* `suma()` *de* `libadic.a`*, por lo que incluye* `adicion.h`*.*
2. `demo6.cpp` *usa la función* `suma()` *de* `libtodos.a`*, por lo que incluye* `todos.h`*.*
3. `demo7.cpp` *usa la función* `suma()` *de* `lib2mod.a`*, por lo que incluye* `todos2.h`*.*

A continuación estudiaremos cada caso por separado.

1. `demo5` *se crea usando la biblioteca* `libadic.a` *que consta de un único módulo objeto (*`adicion.o`*) con dos funciones. El ejecutable es el resultado de enlazar el módulo* `demo5.o` *con el módulo objeto* `adicion.o` *(en este caso, el único) de la biblioteca* `libadic.a`*. El fichero makefile utilizado para construir el ejecutable es* `makefil5.mak`*, que como puede observarse es muy similar a* `makefil1.mak`*.*

   ```
   # Fichero: makefil5.mak
   # Fichero makefile que crea una biblioteca ("libadic.a") con un modulo
   # objeto ("adicion.o") y lo enlaza con otro modulo objeto ("demo5.o")
   # para formar un ejecutable ("demo5").

   INCLUDE = .
   LIB     = .

   demo5: demo5.o libadic.a
           g++ -L$(LIB) -o demo5 demo5.o -ladic

   demo5.o: demo5.cpp adicion.h
           g++ -c -o demo5.o -I$(INCLUDE) demo5.cpp

   adicion.o: adicion.cpp adicion.h
           g++ -c -o adicion.o -I$(INCLUDE) adicion.cpp

   libadic.a: adicion.o
           ar rvs $(LIB)/libadic.a adicion.o
   ```

   El tamaño del ejecutable `demo5` *es* ***4310 bytes****.*

2. `demo6` *se crea usando la biblioteca* `libtodos.a` *que consta de un único módulo objeto (*`todos.o`*) con cuatro funciones. El ejecutable es el resultado de enlazar el módulo* `demo6.o` *con el módulo*

objeto `todos.o` *(en este caso, el único) de la biblioteca* `todos.a`*. El fichero makefile utilizado para construir el ejecutable es* `makefil6.mak`*, muy similar a* `makefil3.mak`*.*

```
# Fichero: makefil6.mak
# Fichero makefile que crea una biblioteca ("libtodos.a") con un modulo
# objeto ("todos.o") y lo enlaza con otro modulo objeto ("demo6.o")
# para formar un ejecutable ("demo6").

INCLUDE = .
LIB     = .

demo6: demo6.o libtodos.a
        g++ -L$(LIB) -o demo6 demo6.o -ltodos

demo6.o: demo6.cpp todos.h
        g++ -c -o demo6.o -I$(INCLUDE) demo6.cpp

todos.o: todos.cpp todos.h
        g++ -c -o todos.o -I$(INCLUDE) todos.cpp

libtodos.a: todos.o
        ar rvs $(LIB)/libtodos.a todos.o
```

El tamaño del ejecutable `demo6` *es* **4422 bytes***.*

3. `demo7` *se crea usando la biblioteca* `lib2mod.a` *que consta de dos módulos objeto (*`adicion.o` *y* `producto.o`*) con dos funciones cada uno. El ejecutable es el resultado de enlazar el módulo* `demo7.o` *con el módulo objeto* `adicion.o` *de la biblioteca* `lib2mod.a`*. El fichero makefile utilizado para construir el ejecutable es* `makefil7.mak`*, muy similar a* `makefil4.mak`*.*

```
# Fichero: makefil7.mak
# Fichero makefile que crea una biblioteca ("lib2mod.a") con dos modulos
# objetos ("adicion.o" y "producto.o") y la enlaza con otro modulo objeto
# ("demo7.o") para formar un ejecutable ("demo7").

INCLUDE = .
LIB     = .

demo7: demo7.o lib2mod.a
        g++ -L$(LIB) -o demo7 demo7.o -l2mod

demo7.o: demo7.cpp todos2.h
        g++ -c -o demo7.o -I$(INCLUDE) demo7.cpp

adicion.o: adicion.cpp adicion.h
        g++ -c -o adicion.o -I$(INCLUDE) adicion.cpp

producto.o: producto.cpp producto.h
        g++ -c -o producto.o -I$(INCLUDE) producto.cpp

lib2mod.a: adicion.o producto.o
        ar rvs $(LIB)/lib2mod.a adicion.o producto.o
```

El tamaño del ejecutable `demo7` *es* ***4310 bytes****.*

Los experimentos anteriores los podemos resumir en el cuadro 2.

Ejecutable	Biblioteca	Objetos	Funciones	Tamaño
`demo5`	`libadic.a`	• `adicion.o`	•`suma()` `resta()`	***4310 bytes***
`demo6`	`libtodos.a`	• `todos.o`	•`suma()` `resta()`	***4422 bytes***
`demo7`	`lib2mod.a`	•`adicion.o`	•`suma()` `resta()`	***4310 bytes***
		`producto.o`	`multiplica()` `divide()`	

Cuadro 2: Comparativa del tamaño de los ejecutables resultantes al enlazar con una biblioteca

Para cada ejecutable (columna **Ejecutable***) indicamos la biblioteca usada en la fase de enlazado (columna* **Biblioteca***). Además, indicamos los módulos objeto que componen cada biblioteca, resaltando los que se usan realmente en el enlace (columna* **Objetos***). También indicamos las funciones que forman parte de cada módulo objeto (columna* **Funciones***), resaltando aquellas que se usan por el programa. Finalmente, indicamos el tamaño del módulo ejecutable (columna* **Tamaño***).*

De estos experimentos anteriores podemos extraer algunas conclusiones interesantes.

1. `demo5` *y* `demo7` *tienen el mismo tamaño. Esto se debe a que los módulos objeto* `demo5.o` *y* `demo7.o` *se enlazan con el módulo objeto* `adicion.o` *extraido de* `libadic.a` *en el primer caso y de* `lib2mod.a` *en el segundo caso.*

2. `demo6` *es el ejecutable de mayor tamaño. El módulo objeto* `demo6.o` *se enlaza con* `todos.o` *(extraído de* `libtodos.a`*). Este módulo contiene cuatro funciones y el programa usa solamente una: como el enlace se hace con* **todo** *el módulo objeto donde se encuentra la función que usa el programa. Así, se añade código inútil que incrementa el tamaño del ejecutable respecto a* `demo5` *y* `demo7`*.*

Ejemplo 26 *Para concluir mostraremos un nuevo experimento que redundará en lo expuesto anteriormente. Utilizaremos el mismo fichero fuente (*`demo_suma.cpp`*) para los dos casos:*

```
/***********************************************/
// Fichero: demo_suma.cpp
/***********************************************/

#include <iostream>
#include "suma.h"
using namespace std;

int main ()
{
   cout << suma (2,5) << endl;
   return (0);
}
```

`demo_suma.cpp` *usa una única función externa,* `suma()`*, cuyo prototipo está declarado en* `suma.h` *y su definición se encuentra en el fichero* `suma.cpp`*:*

```
/*********************************/
// Fichero: suma.h
/*********************************/

int suma  (int, int);
```

```
/*********************************/
// Fichero: suma.cpp
/*********************************/

int suma (int a, int b) {return (a+b);}
```

Los dos experimentos consisten en generar dos ejecutables (`demo8` *y* `demo9`*) de diferente manera y comparar sus tamaños. Ambos se crean con el fichero* `makefil8.mak`*.*

```
# Fichero: makefil8.mak
# Fichero makefile que crea dos ejecutables:
#   1) "demo8": enlazando directamente dos modulos objeto: "demo_suma.o"
#      (contiene la funcion main()) y "suma.o".
#   2) "demo9": enlazando el modulo objeto "demo_suma.o" (contiene la funcion
#      main()) con la biblioteca "libsuma.a" que tiene un modulo objeto ("suma.o").

INCLUDE = .
LIB = .

destinos: demo8 demo9

demo8: demo_suma.o suma.o
        g++ -o demo8 demo_suma.o suma.o

demo9: demo_suma.o libsuma.a
        g++ -L$(LIB) -o demo9 demo_suma.o -lsuma

demo_suma.o: demo_suma.cpp suma.h
        g++ -c -o demo_suma.o -I$(INCLUDE) demo_suma.cpp

suma.o: suma.cpp suma.h
        g++ -c -o suma.o -I$(INCLUDE) suma.cpp

libsuma.a: suma.o
        ar rvs libsuma.a suma.o
```

La ejecución de `make` *sobre este fichero makefile provoca las siguientes tareas:*

```
g++ -c -o demo_suma.o -I. demo_suma.cpp
g++ -c -o suma.o -I. suma.cpp
g++ -o demo8 demo_suma.o suma.o
ar rvs libsuma.a suma.o
a - suma.o
g++ -L.  -o demo9 demo_suma.o -lsuma
```

1. `demo_8` *se crea sin utilizar bibliotecas. Es el resultado de enlazar el módulo* `demo_suma.o` *(obtenido de* `demo_suma.cpp`*) con el módulo objeto* `suma.o` *(obtenido de* `suma.cpp`*).*
2. `demo9` *se crea enlazando* `demo_suma.o` *(el mismo del caso anterior) con la biblioteca* `libsuma.a`*, que consta del único módulo objeto* `suma.o` *(el mismo del caso anterior).*

En ambos casos, los ejecutables obtenidos tienen **4273 bytes***. Este último experimento confirma el hecho de que durante el enlace con una biblioteca se utiliza* **únicamente** *el módulo objeto en el que se encuentra la función utilizada en el programa.*

7.4.3. Ejemplos para expertos

Para realizar estos ejemplos supondremos una estructura de directorios como la siguiente:

```
..../programs/source
             /include
             /lib
             /bin
```

de forma que los ficheros makefile y los fuentes (`.cpp`) se encuentran en el directorio `source` y los de cabecera (`.h`) en `include`. Deseamos que las bibliotecas se guarden en `lib` y los ejecutables en `bin`. El directorio por defecto es `source`, desde donde vamos a ejecutar `make` sobre los ficheros makefile apropiados. Los listados de los ficheros utilizados para estos ejemplos son los siguientes:

```
/**************************************/
// Fichero: demo.cpp
/**************************************/

#include <iostream>
#include "incr.h"
using namespace std;
int main ()
{
   int i=0, doble=0;

   for(; i<10; i=inc1(i), doble=inc2(doble))
      cout << i << " -> " << doble << endl;

   return (0);
}
```

```
/**************************
  // Fichero: incr.h
/**************************

int inc1 (int);
int inc2 (int);
int inc3 (int);
int inc4 (int);
```

```
/*********************************
// Fichero: inc1.cpp
/*********************************

int inc1 (int x) {return (x+1);}
```

```
/*********************************
// Fichero: inc2.cpp
/*********************************

int inc2 (int x) {return (x+2);}
```

```
/*********************************
  // Fichero: inc3.cpp
/*********************************

int inc3 (int x) {return (x+3);}
```

```
/*********************************
  // Fichero: inc4.cpp
/*********************************

int inc4 (int x) {return (x+4);}
```

Ejemplo 27 *Se trata se construir el ejecutable* `demo1` *enlazando* `demo.o` *(construido a partir de* `demo.cpp`*) con la biblioteca* `libincr.a` *que consta de tres módulos objeto:* `inc1.o`, `inc2.o` *y* `inc3.o` *(construidos a partir de* `inc1.cpp`, `inc2.cpp` *y* `inc3.cpp`, *respectivamente). El fichero makefile es* `makef_e1.mak`.

```
# Fichero: makef_e1.mak

INCLUDE = ../include
LIB     = ../lib
BIN     = ../bin

$(BIN)/demo1: demo.o $(LIB)/libincr.a
        g++ -o $(BIN)/demo1 -L$(LIB) demo.o  -lincr

demo.o:  demo.cpp $(INCLUDE)/incr.h
        g++ -c -I$(INCLUDE) -o demo.o demo.cpp

$(LIB)/libincr.a: inc1.o inc2.o inc3.o
        ar rvs $(LIB)/libincr.a inc1.o inc2.o inc3.o

inc1.o: inc1.cpp $(INCLUDE)/incr.h
        g++ -c -o inc1.o -I$(INCLUDE) inc1.cpp

inc2.o: inc2.cpp $(INCLUDE)/incr.h
        g++ -c -o inc2.o -I$(INCLUDE) inc2.cpp

inc3.o: inc3.cpp $(INCLUDE)/incr.h
        g++ -c -o inc3.o -I$(INCLUDE) inc3.cpp
```

La ejecución de la orden `make -f makef_e1.mak` *provoca las siguientes tareas:*

```
g++ -c -I../include -o demo.o demo.cpp
g++ -c -o inc1.o -I../include inc1.cpp
g++ -c -o inc2.o -I../include inc2.cpp
g++ -c -o inc3.o -I../include inc3.cpp
ar rvs ../lib/libincr.a inc1.o inc2.o inc3.o
a - inc1.o
a - inc2.o
a - inc3.o
g++ -o ../bin/demo1 -L../lib demo.o  -lincr
```

construye la biblioteca `libincr.a` *y el ejecutable* `demo1` *y los guarda en los directorios apropiados: la biblioteca en* `../lib` *y el ejecutable en* `../bin`

Ejemplo 28 *Este ejemplo es una versión mejorada del ejemplo anterior, en el que el fichero makefile resulta más compacto al utilizar macros, reglas impícitas patrón y las macros predefinidas* `$<` *y* `$@`.

```
# Fichero: makef_e2.mak

INCLUDE = ../include
LIB     = ../lib
BIN     = ../bin

OBJS = inc1.o inc2.o inc3.o

$(BIN)/demo2: demo.o $(LIB)/libincr.a
        g++ -o $(BIN)/demo2 -L$(LIB) demo.o  -lincr
```

```
demo.o:  demo.cpp $(INCLUDE)/incr.h
        g++ -c -I$(INCLUDE) -o demo.o demo.cpp

$(LIB)/libincr.a: $(OBJS)
        ar rvs $(LIB)/libincr.a $(OBJS)

%.o: %.cpp $(INCLUDE)/incr.h
        g++ -c -o $@ -I$(INCLUDE) $<
```

Si ejecutamos la orden `make -f makef_e2.mak`*, las tareas que realizará* `make` *son las mismas que en el ejemplo anterior:*

```
g++ -c -I../include -o demo.o demo.cpp
g++ -c -o inc1.o -I../include inc1.cpp
g++ -c -o inc2.o -I../include inc2.cpp
g++ -c -o inc3.o -I../include inc3.cpp
ar rvs ../lib/libincr.a inc1.o inc2.o inc3.o
a - inc1.o
a - inc2.o
a - inc3.o
g++ -o ../bin/demo2 -L../lib demo.o  -lincr
```

Ejemplo 29 *Una vez construida la biblioteca* `libincr.a` *(con cualquiera los ejemplos anteriores) con el siguiente fichero makefile podemos añadir nuevos módulos a la biblioteca especificando en la llamada a* `make` *el nombre del módulo fuente, que una vez compilado, se incorporará a la biblioteca. Este nombre se especifica mediante una macro llamada* `MODULO`*.*

```
# Fichero: makef_e3.mak

ifdef MODULO

INCLUDE = ../include
LIB     = ../lib
BIN     = ../bin

$(BIN)/demo3: demo.o $(LIB)/libincr.a
        g++ -o $(BIN)/demo3 -L$(LIB) demo.o  -lincr

demo.o:  demo.cpp $(INCLUDE)/incr.h
        g++ -c -I$(INCLUDE) -o demo.o demo.cpp

$(LIB)/libincr.a: $(MODULO:.cpp=.o)
        ar rvs $(LIB)/libincr.a $(MODULO:.cpp=.o)

$(MODULO:.cpp=.o): $(MODULO)
        g++ -c -o $(MODULO:.cpp=.o) $(MODULO)

else
error:
        @echo ERROR: Falta especificar MODULO=nombre.cpp

# error es un destino "ficticio"

endif
```

Antes de nada, Si ejecutamos la orden `ar t ../lib/libincr.a` *se muestran los módulos que componen la biblioteca* `libincr.a`*. Después del ejemplo anterior, el resultado debe ser:*

```
inc1.o
inc2.o
inc3.o
```

Ahora queremos añadir el módulo `inc4.o` *a la biblioteca y que se reconstruya el ejecutable* `demo3`*. Para esto ejecutaremos la orden:*

```
make -f makef_e3.mak MODULO=inc4.cpp
```

que provoca que `make` *desencadene las siguientes tareas:*

```
g++ -c -o inc4.o inc4.cpp
ar rvs ../lib/libincr.a inc4.o
a - inc4.o
g++ -o ../bin/demo3 -L../lib demo.o  -lincr
```

En cambio, al ejecutar `make` *sin especificar la macro* `MODULO`*, aborta su ejecución (antes de realizar ninguna tarea) y muestra el siguiente mensaje de error:*

```
ERROR: Falta especificar MODULO=nombre.cpp
```

El funcionamiento de este ejecutable es el de los anteriores ya que se construye a partir del mismo fuente.

Ejemplo 30 *Para este ejemplo crearemos la biblioteca* `libincr2.a`*. El fichero makefile utilizado es* `makef_e4.mak`*. En él usamos macros para referenciar a los fuentes que han de compilarse para la creación de la biblioteca (*`FUENTES`*) y sustituciones en macros para calcular los objetos que formarán ésta (*`OBJETOS`*).*

```
# Fichero: makef_e4.mak

INCLUDE = ../include
LIB     = ../lib
BIN     = ../bin

FUENTES  = inc1.cpp inc2.cpp inc3.cpp
OBJETOS  = $(FUENTES:.cpp=.o)


$(BIN)/demo4: demo.o $(LIB)/libincr2.a
        g++ -o $(BIN)/demo4 -L$(LIB) demo.o  -lincr2

demo.o:  demo.cpp $(INCLUDE)/incr.h
        g++ -c -I$(INCLUDE) -o demo.o demo.cpp

$(LIB)/libincr2.a: $(OBJETOS)
        ar rvs $(LIB)/libincr2.a $^

%.o: %.cpp $(INCLUDE)/incr.h
        g++ -c -o $@ -I$(INCLUDE) $<
```

Ahora, si ejecutamos la orden `make -f makef_e4.mak`*, las tareas que realizará* `make` *para construir la biblioteca y el ejecutable son las siguientes (en este orden):*

```
g++ -c -I../include -o demo.o demo.cpp
g++ -c -o inc1.o -I../include inc1.cpp
g++ -c -o inc2.o -I../include inc2.cpp
g++ -c -o inc3.o -I../include inc3.cpp
ar rvs ../lib/libincr2.a inc1.o inc2.o inc3.o
a - inc1.o
a - inc2.o
a - inc3.o
g++ -o ../bin/demo4 -L../lib demo.o  -lincr2
```

Obsérvese la similitud con los ejemplos 27 y 28.

Si en un momento posterior interesase incluir el módulo `inc4.o` *a la biblioteca, bastará con modificar mínimamente este makefile, introduciendo en la definición de la macro* `FUENTES` *el nombre del fuente a incorporar. Esto es, esa línea quedaría como sigue:*

```
FUENTES  = inc1.cpp inc2.cpp inc3.cpp inc4.cpp
```

Después, al ejecutar otra vez la orden `make -f makef_e4.mak` *se realizarán las siguientes tareas:*

```
g++ -c -o inc4.o -I../include inc4.cpp
ar rvs ../lib/libincr2.a inc1.o inc2.o inc3.o inc4.o
r - inc1.o
r - inc2.o
r - inc3.o
a - inc4.o
g++ -o ../bin/demo4 -L../lib demo.o  -lincr2
```

Se compila `inc4.cpp` *para generar* `inc4.o`*, se reemplazan los módulos* `inc1.o`*,* `inc2.o` *y* `inc3.o` *de la biblioteca y se añade* `inc4.o` *a ésta. Finalmente, se reconstruye el ejecutable.*

Este ejemplo supone una alternativa al ejemplo 29, ya con este procedimiento añadimos nuevos módulos objetos a la biblioteca y reconstruimos el ejecutable.

8. Código

8.1. Código de ejemplo 8.1

8.1.1. Un programa para depurar muy simple

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

const int max=25;

void Cambia(int x, int y);

int main()  {
   int a;
   int b;

   cout << "Programa que intercambia el contenido dos variables " << endl;
   cout << "Introduce la variable a (entero)" << endl;
   cin >> a;

   cout << "Introduce la variable b (entero)" << endl;
   cin >> b;

   cout << "Antes del intercambio las variables son" << endl;
   cout << "a=" << a << "   y b=" << b << endl;
   Cambia(a,b);
   cout << "Después del intercambio las variables son" << endl;
```

```
   cout << "a=" << a << "   y b=" << b << endl;
   return 0;
}

void Cambia(int x, int y)  {
   int a;

   a = x;
   x = y;
   y = x;
}
```

8.2. Código de ejemplo 8.2

8.2.1. Un makefile muy simple

```
# Fichero makefile.ej1
# Muestra la filosofía de las reglas y el orden en que se interpretan
# Si se invoca sin argumentos la regla principal será la de whooper (RP1)
#   make -f makefile.ej1
# Se puede invocar con el argumento bigking y entonces la regla principal será
# la de Big King (RP2). La regla RP1 se obvia
#   make -f makefile.ej1 bigking
#


# RP1: Esta será la regla principal por defecto
whooper: cocinar
   @echo
   @echo "<Whooper listo para comer!"
   @echo

# RP2: Esta podrá ser otra regla principal, pero sólo si se llama desde la línea de órdenes
bigking: cocinar queso
   @echo
   @echo "<Big King listo para comer!"
   @echo

# El resto de las reglas se llamarán o no dependiendo de cuál haya sido la regla principal
cocinar: bollo hamburguesa lechuga
   @echo Poniendo la hamburguesa y la lechuga en el bollo

bollo:
   @echo Preparando el bollo de pan

hamburguesa: ham1 ham2
   @echo Poniendo la hamburguesa en el bollo


ham1:
   @echo "   Cogiendo la hamburguesa del frigo"

ham2:
   @echo "   Asando la hamburguesa"
   @echo Pulse return para continuar
   @read

lechuga:
   @echo Añadiendo lechuga a la hamburguesa

queso:
   @echo Añadiendo queso a la hamburguesa
```

8.3. Código de ejemplo 8.3

8.3.1. Fichero calculos.cpp

```
/**
@file mi-ej.cpp
@author mp2
@brief Ejemplo de programa en C++ que será extendido en sucesivas sesiones de prácticas. Se encarga
de realizar algunos cálculos sencillos. Contiene errores sintácticos y semánticos.
1. Faltan los prototipos
2. Falta código por hacer.
3. Transcripción del combinatorio semánticamente mal
4. Falta algún ;
5. Asignación del indicador de salida del bucle mal (fin del switch)
*/

#include <iostream>
using namespace std;

int main()  {
   int a, b;                     /* Variables de entrada */
   float c;                   /* Variables de entrada */
   char opc;                  /* Teclas del menú      */
   bool salir=false;             /* Controla la salida del programa */

   while (!salir)  {
      opc = Menu();

      /* Opciones del menú */
      switch (opc)  {
         /* Factorial */
         case 'F':
         case 'f':
            a = FiltroPos();
            c = Factorial(a);
            cout << "Resultado = " << c;
            break;

         /* Potencia */
         case 'P':
         case 'p':
            /* NO IMPLEMENTADO */
            break

         /* Combinatorio */
         case 'c':
         case 'C':
            a = FiltroPos();
            b = FiltroPos();
            /* NO IMPLEMENTADO */
            break;

         /* Salir del programa */
         case 'S':
         case 's':
            salir = false;
            break;
      } /* del switch */
   } /* del while */
   return 0;
}


/**
@brief Calcula la potencia
@param b La base
@param e El exponente
@pre  \a e > 0
@return \a b elevado a \a e
```

```
*/
float Potencia(float b, int e)  {

/*NO IMPLEMENTADA */

}

/**
@brief Calcula el factorial
@param x El número al que se le va a calcular el factorial
@pre \a x >= 0
@return \a x!
*/
float Factorial(int x)  {
   float res = 1;
   int i;

   for (i=2; i<x; i++)
      res = res * i;
   return res;
}

/**
@brief Calcula un número combinatorio
@param n
@param m
@pre \a n <= \a m
@return  n! / [(n-m)! m!]
*/
float Combina(int n, int m)  {

   return factorial(n)/factorial(n-m) * factorial(m);
}

/**
@brief Lee un número entero desde el teclado asegurándose de que éste es mayor o igual a cero.
@return Un entero >= cero
*/
int FiltroPos()  {
   int x;

   do  {
      cout << "Introduce un número entero >=0  ";
      cin >> x;
   } while (x < 0);
   return x;
}


/**
@brief Presenta un menú en la pantalla y lee la opción correspondiente. No filtra esta entrada de caracteres.
@return Un carácter que representa a la tecla que se ha pulsado
*/
char Menu()  {
   char t;

   cout << "\n\n   Menu\n=======\n\n";
   cout << "[F]actorial\n[P]otencia\n[C]ombinatorio\n[S]alir\n\n";
   cout << "Introduce la opción: ";
   cin >> t;
   return t;
}
```

8.4. Código de ejemplo 8.4

8.4.1. Fichero ordena1.cpp

```
/**
@file ordena1.cpp
@author mp2
@brief Este es un fichero que se va autilizar en sucesivas sesiones de trabajo durante las prácticas.
*/

#include <iostream>
#include <iomanip>
#include <cstdlib>     // Para manejar nums. aleatorios
#include <ctime>       // Para fijar la semilla del generador de nums. aleat.
using namespace std;

/* Prototipos y constantes */

const int MAX=25;          // Tamanio del vector
const int MAX_LINE=10;     // Máximo por línea
const int MY_MAX_RAND=100; // Nums. aleatorios entre 0 y 99


/**
@brief Rellena un vector de enteros con numeros aleatorios.
@param p Un vector a rellenar
@param tope  El numero de elementos del vector \a p
*/
void llena_vector  (int *p, int tope);

/**
@brief Muestra un vector de enteros tabulado en pantalla
@param p Un vector a mostrar
@param tope El numero de elementos del vector \a p
*/
void pinta_vector  (int *p, int tope);

/**
@brief Ordena un vector de forma ascendente (metodo de la burbuja)
@param p Un vector a ordenar
@param tope El numero de elementos del vector
*/
void ordena_vector (int *p, int tope);

/**
@brief Intercambia los valores de dos variables enteras
@param a Primera variable entera
@param b Segunda variable entera
*/
void swap (int &a, int &b);


int main ()
{
   int m [MAX];       // vector de trabajo

   /* Rellena completamente el vector m */
   llena_vector (m, MAX);

   /* Muestra el vector m antes de ordenarlo */
   cout << endl << "Vector original (Antes de ordenar): " << endl;
   pinta_vector (m, MAX);

   /* Ordena el vector m */
   ordena_vector (m, MAX);

   /* Muestra el vector m despues de ordenarlo */
   cout << endl << "Vector final (Despues de ordenar): " << endl;
   pinta_vector (m, MAX);
```

```
   cout << endl;

   return (0);
}


void llena_vector  (int *p, int tope)
{
   time_t t;         // tipo definido en time.h
   int i;

   srand ((int) time(&t)); // Fija la semilla del generador: Inicializa el
                           // generador de nums.aleat. con el reloj del sistema
   for (i=0; i<tope; i++, p++)
      *p = rand () % MY_MAX_RAND;  // Numero aleatorio entre 0 y MY_MAX_RAND-1
}

void pinta_vector  (int *p, int tope)
{
   int i;

   for (i=0; i<tope; i++, p++) {
     if (i%MAX_LINE == 0)
       cout << endl;
     cout << setw(5) << *p;
   }

}

void ordena_vector (int *p, int tope)
{
   int i, j;

   for (i=0; i<tope-1; i++)
      for (j=0; j<tope-1; j++)
         if (p[j] > p[j+1])
            swap (p[j], p[j+1]);

}


void swap (int &a, int &b)
{
   int tmp;

   tmp = a;
   a = b;
   b = tmp;
}
```

8.4.2. makefile

```
# Fichero: makefile.v1
# Generacion del ejecutable a partir de un unico fuente: "ordena1.cpp

destinos: ordena1

ordena1 : ordena1.cpp
   @echo Compilando ordena1.cpp ...
   g++ ordena1.cpp -o ordena1
```

8.5. Código de ejemplo 8.5

8.5.1. Fichero ppal.cpp

```
/**
@file ppal.cpp
@author mp2
```

```
@brief Este es el fichero principal del proyecto. En el se encuentra únicamente la función main.
*/

#include <iostream>
#include "funcsvec.h"
using namespace std;

int main ()
{
   int m [MAX];       // vector de trabajo

   /* Rellena completamente el vector m */
   llena_vector (m, MAX);

   /* Muestra el vector m antes de ordenarlo */
   cout << endl << "Vector original (Antes de ordenar): " << endl;
   pinta_vector (m, MAX);

   /* Ordena el vector m */
   ordena_vector (m, MAX);

   /* Muestra el vector m despues de ordenarlo */
   cout << endl << "Vector final (Despues de ordenar): " << endl;
   pinta_vector (m, MAX);
   cout << endl;

   return (0);
}
```

8.5.2. Fichero funcsvec.cpp

```
/**
@file funcsvec.cpp
@author mp2
@brief En este fichero se han dejado todas las funciones que necesita el programa
distintas de la función main y las constantes MAX_LINE y MY_MAX_RAND
*/

#include <iostream>
#include <iomanip>
#include <cstdlib>     // Para manejar nums. aleatorios
#include <ctime>       // Para fijar la semilla del generador de nums. aleat.
#include "funcsvec.h"
using namespace std;

const int MAX_LINE=10;     // Máximo por línea
const int MY_MAX_RAND=100; // Nums. aleatorios entre 0 y 99

/**
@brief Intercambia los valores de dos variables enteras
@param a Primera variable entera
@param b Segunda variable entera
*/
void swap (int &a, int &b);


void llena_vector  (int *p, int tope)
{
   time_t t;         // tipo definido en time.h
   int i;

   srand ((int) time(&t)); // Fija la semilla del generador: Inicializa el
                           // generador de nums.aleat. con el reloj del sistema
   for (i=0; i<tope; i++, p++)
      *p = rand () % MY_MAX_RAND;  // Numero aleatorio entre 0 y MY_MAX_RAND-1
}
```

```
void pinta_vector  (int *p, int tope)
{
   int i;

   for (i=0; i<tope; i++, p++) {
     if (i%MAX_LINE == 0)
       cout << endl;
     cout << setw(5) << *p;
   }

}

void ordena_vector (int *p, int tope)
{
   int i, j;

   for (i=0; i<tope-1; i++)
      for (j=0; j<tope-1; j++)
         if (p[j] > p[j+1])
            swap (p[j], p[j+1]);

}


void swap (int &a, int &b)
{
   int tmp;

   tmp = a;
   a = b;
   b = tmp;
}
```

8.5.3. Fichero funcsvec.h

```
/**
@file funcsvec.h
@author mp2
@brief Fichero de encabezamiento
*/

#ifndef FUNCSVEC
#define FUNCSVEC

/* Prototipos y constantes */

const int MAX=25;          // Tamanio del vector



/**
@brief Rellena un vector de enteros con numeros aleatorios.
@param p Un vector a rellenar
@param tope  El numero de elementos del vector \a p
*/
void llena_vector  (int *p, int tope);

/**
@brief Muestra un vector de enteros tabulado en pantalla
@param p Un vector a mostrar
@param tope El numero de elementos del vector \a p
*/
void pinta_vector  (int *p, int tope);

/**
@brief Ordena un vector de forma ascendente (metodo de la burbuja)
@param p Un vector a ordenar
```

```
@param tope El numero de elementos del vector
*/
void ordena_vector (int *p, int tope);

#endif
```

8.5.4. makefile

```
# Fichero: makefile.v2
# Ejemplo de makefile que genera un ejecutable a partir de dos ficheros objeto.
#   1) "ppal.o":codigo objeto del programa principal (de "ppal.cpp.
#   2) "funcsvec.o": codigo objeto de las funciones auxiliares (de "funcsvec.cpp.

destinos: ordena1 clean

ordena1 : ppal.o funcsvec.o
   g++ -o ordena1 ppal.o funcsvec.o

ppal.o : ppal.cpp funcsvec.h
   g++ -c -o ppal.o -I. ppal.cpp

funcsvec.o : funcsvec.cpp funcsvec.h
   g++ -c -o funcsvec.o -I. funcsvec.cpp

clean :
   rm ppal.o funcsvec.o
```

8.6. Código de ejemplo 8.6

8.6.1. Fichero ppal.cpp

```
/**
@file ppal.cpp
@author mp2
@brief Este es el fichero principal del proyecto. En el se encuentra únicamente la función main.
*/

#include <iostream>
#include "vec_ES.h"
#include "ordena.h"
using namespace std;

int main ()
{
   int m [MAX];       // vector de trabajo

   /* Rellena completamente el vector m */
   llena_vector (m, MAX);

   /* Muestra el vector m antes de ordenarlo */
   cout << endl << "Vector original (Antes de ordenar): " << endl;
   pinta_vector (m, MAX);

   /* Ordena el vector m */
   ordena_vector (m, MAX);

   /* Muestra el vector m despues de ordenarlo */
   cout << endl << "Vector final (Despues de ordenar): " << endl;
   pinta_vector (m, MAX);
   cout << endl;

   return (0);
}
```

8.6.2. Fichero vec_ES.cpp

```
/**
@file vec_ES.cpp
@author mp2
@brief Implementación de algunas funciones de E/S sobre vectores
*/

#include <iostream>
#include <iomanip>
#include <cstdlib>     // Para manejar nums. aleatorios
#include <ctime>       // Para fijar la semilla del generador de nums. aleat.
#include "vec_ES.h"
using namespace std;

const int MAX_LINE=10;     // Máximo por línea
const int MY_MAX_RAND=100; // Nums. aleatorios entre 0 y 99

void llena_vector  (int *p, int tope)
{
   time_t t;         // tipo definido en time.h
   int i;

   srand ((int) time(&t)); // Fija la semilla del generador: Inicializa el
                           // generador de nums.aleat. con el reloj del sistema
   for (i=0; i<tope; i++, p++)
      *p = rand () % MY_MAX_RAND;  // Numero aleatorio entre 0 y MY_MAX_RAND-1
}

void pinta_vector  (int *p, int tope)
{
   int i;

   for (i=0; i<tope; i++, p++) {
     if (i%MAX_LINE == 0)
       cout << endl;
     cout << setw(5) << *p;
   }

}
```

8.6.3. Fichero vec_ES.h

```
/**
@file vec_ES.h
@author mp2
@brief Operaciones de E/S sobre vectores genéricos
*/

#ifndef VEC_ES
#define VEC_ES

/* Prototipos y constantes */

const int MAX=25;          // Tamanio del vector


/**
@brief Rellena un vector de enteros con numeros aleatorios.
@param p Un vector a rellenar
@param tope  El numero de elementos del vector \a p
*/
void llena_vector  (int *p, int tope);

/**
@brief Muestra un vector de enteros tabulado en pantalla
@param p Un vector a mostrar
@param tope El numero de elementos del vector \a p
```

```
*/
void pinta_vector  (int *p, int tope);
#endif
```

8.6.4. Fichero ordena.cpp

```
/**
@file ordena.cpp
@author mp2
@brief Implementación de funciones de ordenación sobre vectores.
Ampliar con nuevas funciones de ordenación más eficientes
*/

/**
@brief Intercambia los valores de dos variables enteras
@param a Primera variable entera
@param b Segunda variable entera
*/
void swap (int &a, int &b);

void ordena_vector (int *p, int tope)
{
   int i, j;

   for (i=0; i<tope-1; i++)
      for (j=0; j<tope-1; j++)
         if (p[j] > p[j+1])
            swap (p[j], p[j+1]);

}


void swap (int &a, int &b)
{
   int tmp;

   tmp = a;
   a = b;
   b = tmp;
}
```

8.6.5. Fichero ordena.h

```
/**
@file ordena.h
@author mp2
@brief Funciones de ordenación de vectores genéricos
*/

#ifndef ORDENA
#define ORDENA

/**
@brief Ordena un vector de forma ascendente (metodo de la burbuja)
@param p Un vector a ordenar
@param tope El numero de elementos del vector
*/
void ordena_vector (int *p, int tope);

#endif
```

8.6.6. makefile

```
# Fichero: makefile.v3
# Ejemplo de makefile que genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal (de "ppal.cpp").
```

```
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector (de "vec_ES.cpp").
#   3) "ordena.o": codigo objeto de la funcion de ordenacion (de "ordena.cpp").

destinos: ordena1 clean

ordena1 : ppal.o vec_ES.o ordena.o
   g++ -o ordena1 ppal.o vec_ES.o ordena.o

ppal.o : ppal.cpp vec_ES.h ordena.h
   g++ -c -o ppal.o -I. ppal.cpp

vec_ES.o : vec_ES.cpp vec_ES.h
   g++ -c -o vec_ES.o -I. vec_ES.cpp

ordena.o : ordena.cpp ordena.h
   g++ -c -o ordena.o -I. ordena.cpp

clean :
   rm ppal.o vec_ES.o ordena.o
```

9. Más información

9.1. Documentación dentro del SO

1. Documentación sobre `gcc/g++`

   `man g++`

2. Documentación sobre `make`

   `info make` En modo de texto

   `Alt+F2 info:/make` En kdehelp, modo gráfico (navegable)

3. Documentación y software sobre `ddd` y `gbd`.

   `info ddd info gdb` En modo de texto

   `Alt+F2 info:/ddd` En kdehelp, modo gráfico (navegable)

   `Alt+F2 info:/gdb`

   Desde el menú de `ddd`: Opción Help

9.2. Documentación en Internet

1. Documentación sobre `gcc/g++` en GNU

   `http://www.gnu.org/software/gcc/onlinedocs/`

   `http://www.gnu.org/software/gcc/gcc.html`

2. Documentación sobre `make`

   `http://www.gnu.org/manual/make-3.79.1/make.html`

3. Documentación y software sobre `ddd` y `gdb`.

   `http://www.gnu.org/software/ddd/ddd.html`

   `http://www.gnu.org/software/gdb`