



DECSAI

Departamento de Ciencias de la Computación e I.A.

Universidad de Granada

Metodología de la Programación Grado en Ingeniería Informática

Prácticas

Curso 2012/2013

Francisco J. Cortijo Bon

Departamento de Ciencias de la Computación
e Inteligencia Artificial
ETS de Ingenierías Informática y de Telecomunicación
Universidad de Granada
cb@decsai.ugr.es

Práctica 1. El modelo de compilación

Francisco J. Cortijo Bon

Curso 2012-2013

Objetivos

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

Índice

1. El modelo de compilación en C++	2
1.1. El preprocesador	3
1.2. El compilador	4
1.3. El enlazador	4
1.4. Bibliotecas. El gestor de bibliotecas	4
2. g++ : el compilador de GNU para C++	5
2.1. Un poco de historia	5
2.2. Sintaxis	6
2.3. Opciones más importantes	7
A. Introducción al depurador DDD	10
A.1. Conceptos básicos	10
A.2. Pantalla principal	10
A.3. Ejecución de un programa paso a paso	11
A.4. Inspección y modificación de datos	11
A.5. Inspección de la pila	12
A.6. Mantenimiento de sesiones de depuración	13
A.7. Reparación del código	13
B. El preprocesador de C++	13
B.1. Constantes simbólicas y macros funcionales	14
B.1.1. Constantes simbólicas	14
B.1.2. Macros funcionales (con argumentos)	15
B.1.3. Eliminación de constantes simbólicas	15
B.2. Inclusión de ficheros	16
B.2.1. Inclusión condicional de código	16

1. El modelo de compilación en C++

En la figura 1 mostramos el esquema básico del proceso de compilación de programas y creación de bibliotecas en C++. En este gráfico, indicamos mediante un *rectángulo con esquinas redondeadas* los diferentes programas involucrados en estas tareas, mientras que los *cilindros* indican los tipos de ficheros (con su extensión habitual) que intervienen.

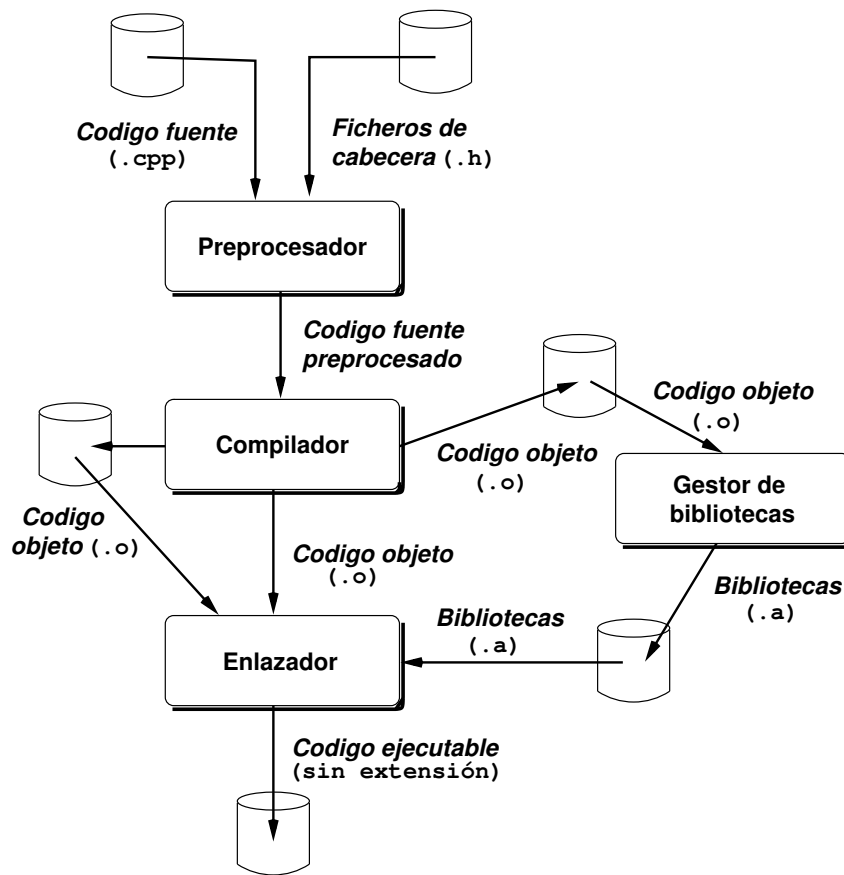


Figura 1: Esquema del proceso de compilación (generación de programas ejecutables) en C++

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. Éste puede construirse de diversas formas:

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

Las dos últimas requieren que previamente se hayan construido los ficheros objeto (opción 2) y los ficheros de biblioteca (opción 3). Como se puede comprobar en el esquema anterior, la creación de éstos también está contemplada en el esquema. Así, es posible generar únicamente ficheros objeto para:

1. Enlazarlos con otros y generar un ejecutable.

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

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

2. Incorporarlos a una biblioteca.

Una biblioteca, en la terminología de C++, será es una *colección de módulos objeto*. Entre ellos existirá alguna *relación*, que debe entenderse en un sentido amplio: si dos módulos objeto están en la misma biblioteca, contendrán funciones que trabajen sobre un mismo *tema* (por ejemplo, funciones de procesamiento de cadenas de caracteres).

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

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

Ejercicio

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

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

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

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

1.1. El preprocesador

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

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

Dos de las directivas más comúnmente empleadas en C++ son `#include` y `#define`. En la sección **B** 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 pueden 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 detallado de lo que se conoce comúnmente por compilación.

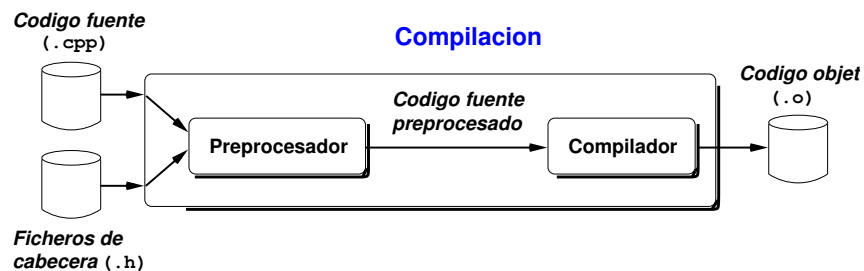


Figura 2: Fase de compilación

1.2. El compilador

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

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

1.3. El enlazador

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

1.4. Bibliotecas. El gestor de bibliotecas

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

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

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.

2. g++ : el compilador de GNU para C++

2.1. Un poco de historia

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

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

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

GNU es un acrónimo recursivo que significa **GNU No es Unix** (GNU is **Not** Unix). Puesto que en inglés “gnu” (en español “ñu”) se pronuncia igual que “new”, Richard Stallman recomienda pronunciarlo “guh-noo”. En español, se recomienda pronunciarlo ñu como el antílope africano o fonéticamente; por ello, el término mayoritariamente se deletrea (G-N-U) para su mejor comprensión. En sus charlas Richard Stallman finalmente dice siempre «Se puede pronunciar de cualquier forma, la única pronunciación errónea es decirle ‘linux’».

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

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

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

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

En 1991, **Linus Torvalds** empezó a escribir el núcleo Linux y decidió distribuirlo bajo la licencia

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

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

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

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

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

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

Otros programas del Proyecto GNU relacionados con nuestra materia son:

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

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

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

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

2.2. Sintaxis

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

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

donde:

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

En el caso de ir acompañadas de algún argumento, se especifican a continuación de la opción. Por ejemplo, la opción `-o saludo.o` indica que el nombre del fichero resultado es `saludo.o`, la opción `-I /usr/include` indica que se busquen los ficheros de cabecera en el directorio `/usr/include`, etc. Las opciones mas importantes se describen con detalle en la sección 2.3.

- *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), `.cpp` (código fuente C++).

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

2.3. Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

- ansi considera únicamente código fuente escrito en C/C++ estándar y rechaza cualquier extensión que pudiese tener conflictos con ese estándar.
- c realizar solamente el preprocesamiento y la compilación de los ficheros fuentes. No se lleva a cabo la etapa de enlazado.

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

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

Ejercicio

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

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

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



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

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

Ejercicio

Ejecutar la siguiente orden y observar e interpretar el resultado. El diagrama de dependencias se muestra en la figura 3.

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



Figura 3: Diagrama de dependencias para *saludo*

Ejercicio

Ejecutar las siguientes órdenes y observar e interpretar el resultado. El diagrama de dependencias se muestra en la figura 4.

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

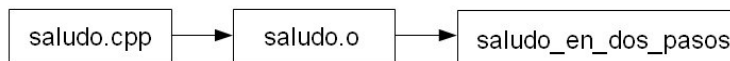


Figura 4: Diagrama de dependencias para *saludo_en_dos_pasos*

- W *all* Muestra todos los mensajes de advertencia del compilador.
- g Incluye en el ejecutable la información necesaria para poder trazarlo empleando un depurador.
- v Muestra con detalle en las órdenes ejecutadas por g++.

Ejercicio

1. Ejecutar la siguiente orden y observar e interpretar el resultado
`g++ -v -o bin/saludo src/saludo.cpp`
2. Ejecutar la siguiente orden y observar e interpretar el resultado
`g++ -Wall -v -o bin/saludo src/saludo.cpp`
3. Ejecutar la siguiente orden y observar e interpretar el resultado, comparando el tamaño del fichero obtenido con el de *saludo*
`g++ -g -o bin/saludo_con_g src/saludo.cpp`

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

Ejercicio

Ejecutar la siguiente orden:

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

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

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

- L *path* indica al enlazador el directorio donde se encuentran los ficheros de biblioteca. Como ocurre con la opción `-I`, se puede utilizar la opción `-L` varias veces para especificar distintos directorios de biblioteca.
 1. Los ficheros de biblioteca que deben usarse se proporcionan a `g++` con la opción `-l fichero`.
 2. Esta opción hace que el enlazador busque en los directorios de bibliotecas (entre los que están los especificados con `-L`) un fichero de biblioteca llamado `libfichero.a` y lo usa para enlazarlo.

Ejercicio

Ejecutar la siguiente orden:

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

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

Copiar el fichero `potencias.cpp` en el directorio `src` y ejecutar las siguiente orden. Observar e interpretar los resultado

```
g++ -o bin/potencias src/potencias.cpp -lm
```

Ejecutar ahora la misma orden pero sin enlazar con la biblioteca matemática (`libm.a`). Observará que funciona correctamente ¿porqué? Ejecute la siguiente orden para interpretarlo:

```
g++ -v -o bin/potencias src/potencias.cpp -lm
```

- 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 línea `#define` al principio del fichero fuente, salvo que si se usa `-D`, el ámbito de la macrodefinición incluye todos los ficheros especificados en la llamada al compilador.
- O Optimiza el tamaño y la velocidad del código compilado. Existen varios niveles de optimización cada uno de los cuales proporciona un código menor y más rápido a costa de emplear más tiempo de compilación y memoria principal. Ordenadas de menor a mayor intensidad son: `-O`, `-O1`, `-O2` y `-O3`. Existe una opción adicional `-Os` orientada a optimizar exclusivamente el tamaño del código compilado (esta opción no lleva a cabo ninguna optimización de velocidad que implique un aumento de código).

A. Introducción al depurador DDD

A.1. Conceptos básicos

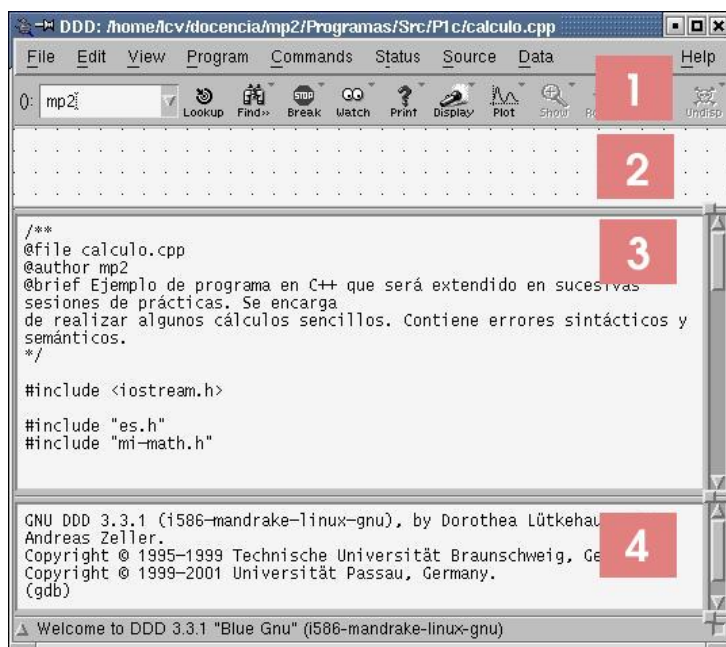
El programa `ddd` es, básicamente, una interfaz (*front-end*) separada que se puede utilizar con un depurador en línea de órdenes. En el caso que concierne a este documento, `ddd` será la interfaz de alto nivel del depurador `gdb`.

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.2. Pantalla principal

La pantalla principal del depurador se muestra en la figura 5.a).



(a)



(b)

Figura 5: Pantalla principal de `ddd`

En ella se pueden apreciar las siguientes partes.

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

Sobre la ventana principal aparece una ventana flotante de herramientas que se muestra en la figura 5.b) desde la que se pueden hacer, de forma simplificada, las mayoría de las operaciones de depuración.

A.3. Ejecución de un programa paso a paso


Una vez cargado un programa binario, se puede comenzar la ejecución siguiendo cualquiera de los métodos mostrados en el cuadro 1. Hay que tener en cuenta que esta orden inicia la ejecución del programa de la misma forma que si se hubiese llamado desde la línea de argumentos, de forma que, de no haber operaciones de entrada/salida desde el teclado, el programa comenzará a ejecutarse sin control directo desde el depurador hasta que termine, momento en el que devuelve el control al depurador mostrando el siguiente mensaje

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

En cualquier momento se puede terminar la ejecución de un programa mediante distintas formas, la más rápida es mediante la orden `kill` (ver cuadro 1). También se pueden pasar argumentos a la función `main` desde la ventana que aparece en la figura 6.




Figura 6: Ventana para pasar argumentos a `main`

Para comenzar a ejecutar un programa bajo control del depurador es conveniente colocar un punto de ruptura¹ 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`). Si esta ventana no estuviese visible, entonces puede mostrarse pulsando en menú `Program - Run in execution window`.

A.4. 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

¹Un 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 .

valores durante un tiempo limitado, o permanentemente en la ventana de datos (*watch*, de forma que sus valores se visualicen durante toda la ejecución (ver cuadro 1). Es necesario aclarar que sólo se puede visualizar el valor de una variable cuando la línea de ejecución activa se encuentre en un ámbito en el que sea visible esta variable. Asimismo, *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.

A.5. 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 momento dado de la ejecución de un programa (ver cuadro 1).

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

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

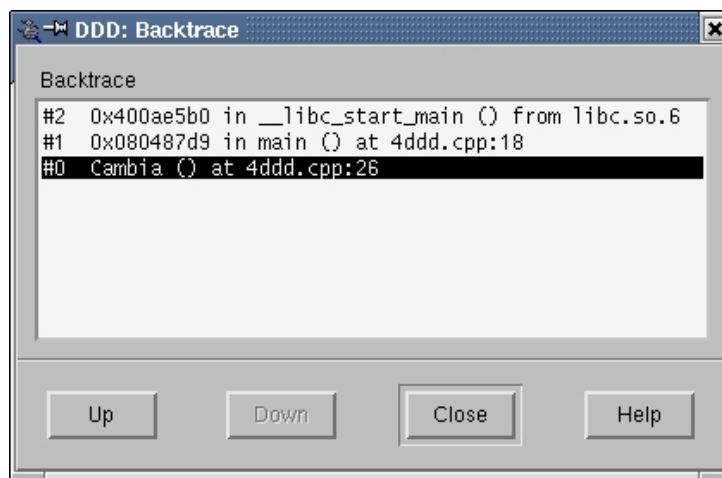


Figura 7: Ventana que muestra el estado de la pilla de llamadas a módulos

A.6. 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.

A.7. Reparación del código

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

B. El preprocesador de C++

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

1. Eliminar los comentarios.
2. Interpretar y procesar las directivas de preprocesamiento. El preprocesador proporciona un conjunto de directivas que resultan una herramienta sumamente útil al programador. Todas las directivas comienzan *siempre* por el símbolo `#`.
 - `#include`: Sustituye la línea por el contenido del fichero especificado.
Por ejemplo, `#include <iostream>` incluye el fichero `iostream.h`, que contiene declaraciones de tipos y funciones de entrada/salida de la biblioteca estándar de C++. La inclusión implica que *todo* el contenido del fichero incluido sustituye a la línea `#include`.
Los nombres de los ficheros de cabecera heredados de C comienzan por la letra `c`, y se incluyen usando la misma sintaxis. Por ejemplo: `#include <cstring>`, `#include <cstdlib>`,...

- **#define:** Define una constante (identificador) simbólico.
Sustituye las apariciones del identificador por el valor especificado, salvo si el identificador se encuentra dentro de una constante de cadena de caracteres (entre comillas).
Por ejemplo, `#define MAX_SIZE 100` establece el valor de la constante simbólica `MAX_SIZE` a 100. En el programa se utilizará la constante simbólica y el preprocesador sustituye cada aparición de `MAX_SIZE` por el literal 100 (de tipo `int`).

El uso de las directivas de preprocesamiento proporciona varias ventajas:

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

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

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

`#undef`: Eliminación de constantes simbólicas.

`#include`: Inclusión de ficheros.

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

B.1. Constantes simbólicas y macros funcionales

B.1.1. Constantes simbólicas

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

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

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

Veamos algunos ejemplos sencillos.

```
#define TAMMAX 256
```

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

```
#define UTIL_VEC
```

simplemente define la constante simbólica `UTIL_VEC`, aunque sin asignarle ningún valor de sustitución: se puede interpretar como una “bandera” (existe/no existe). Se suele emplear para la inclusión condicional de código (ver sección [B.2.1](#) de este apéndice).

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

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

B.1.2. 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.
2. En general, las macros recursivas no funcionan.
3. En las macros funcionales el tipo de los argumentos es indiferente. Suponen una gran ventaja cuando queremos hacer el mismo tratamiento a diferentes tipos de datos.

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

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

y tenemos la sentencia

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

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

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

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

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

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

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

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

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

Calcula la diferencia absoluta entre dos operandos.

B.1.3. 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`.

B.2. 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 2.3), mientras que `"fichero"` indica que se encuentra en el directorio donde se está realizando la compilación. Así,

```
#include <iostream>
```

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

B.2.1. 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 2.3).

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 cláusula `#define`). Este predicado se suele usar en su forma resumida (columna derecha):

```
#if defined(identificador)      #ifdef (identificador)
#if !defined(identificador)      #ifndef (identificador)
```

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

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

sí puede ocurrir lo siguiente:

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

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

```
#ifndef (HDR)
#define HDR
```

Resto del contenido del fichero de cabecera

```
#endif
```

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

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

Práctica 2. El programa `make` y los ficheros *makefile*

Francisco J. Cortijo Bon

Curso 2012-2013

Objetivos

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

Índice

1. El programa <i>make</i>	20
1.1. Sintaxis	21
1.2. Opciones más importantes	21
1.3. Funcionamiento de <i>make</i>	21
2. Ficheros <i>makefile</i>	22
2.1. Comentarios	22
2.2. Reglas. Reglas explícitas	22
2.3. Órdenes. Prefijos de órdenes	24
2.4. Destinos Simbólicos	25
2.5. Destinos .PHONY	25
2.6. Macros en ficheros <i>makefile</i>	26
2.7. Macros predefinidas	27
A. Sustituciones en macros	28
B. Macros como parámetros en la llamada a <i>make</i>	28
C. Reglas implícitas	29
C.1. Reglas implícitas patrón	30
C.2. Reglas patrón estáticas	30
D. Directivas condicionales en ficheros <i>makefile</i>	31

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

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

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

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

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

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

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

1. El programa *make*

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

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

1.1. 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 `/`. En la sección 1.2 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 2.3). Una explicación detallada de los destinos puede encontrarse en las secciones 2.2 y 2.4.

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 1.3.

1.2. Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

- `-h` ó `--help` Proporciona ayuda acerca de `make`.
- `-f fichero`. Utilizaremos esta opción si se proporciona a `make` un nombre de fichero distinto del de `makefile` o `Makefile`. Se toma el fichero llamado `fichero` como el fichero `makefile`.
- `-n`, `--just-print`, `--dry-run` ó `--recon`: Muestra las instrucciones que *ejecutaría* la utilidad `make`, pero **no** los ejecuta. Sirve para verificar la corrección de un fichero `makefile`.
- `-p`, `--print-data-base`: Muestra las reglas y macros asociadas al fichero `makefile`, incluidas las *predefinidas*.

1.3. 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 si el destino especificado depende de otros que no están contruidos. Para saber qué destinos debe construir comprueba las listas de dependencias. Esta reacción en cadena se llama **dependencia encadenada**.
3. Si en cualquier paso falla al construir algún destino, se detiene la ejecución, muestra un mensaje de error y borra el destino que estaba construyendo.

2. Ficheros *makefile*

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

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

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

2.1. 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.

Ejercicio

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

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

Se incluyen dos líneas de comentario al principio del fichero *makefile* que indican las tareas que realizará la utilidad *make*.

Si el fichero *makefile* se encuentra en el directorio actual, para realizar las acciones en él indicadas tan solo habrá que ejecutar la orden:

```
% make
```

ya que el fichero *makefile* se llama *makefile*. El mismo efecto hubiéramos obtenido ejecutando la orden:

```
% make -f makefile
```

2.2. Reglas. Reglas explícitas

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

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

El formato habitual de una regla explícita es el siguiente:

objetivo: lista de dependencia
orden(es)

donde:

- El **objetivo** identifica la regla e indica el fichero a crear.

- La **lista de dependencia** especifica los ficheros de los que depende **objetivo**. Esta lista contiene los nombres de los ficheros separados por espacios en blanco.

Si alguno de los ficheros especificados en esta lista se ha modificado, se busca una regla que contenga a ese fichero como destino y se construye. Una vez se han construido las últimas versiones de los ficheros especificados en **lista de dependencia** se construye **objetivo**.

- Las **órden(es)** son órdenes válidas para el sistema operativo en el que se ejecute la utilidad `make`. Pueden incluirse varias instrucciones en una regla, cada uno en una línea distinta. Usualmente estas instrucciones sirven para construir el **objetivo** (en esta asignatura, habitualmente son llamadas al compilador `g++`), aunque no tiene porque ser así.

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

Ejercicio

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

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

que indica que para construir el objetivo `saludo` se requiere la existencia de `saludo.cpp` (`saludo` *depende de* `saludo.cpp`). Esta dependencia se esquematiza en el diagrama de dependencias mostrado en la figura 1. Finalmente, el destino se construye ejecutando la orden:

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

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

1. Ejecutar las siguientes órdenes e interpretar el resultado:

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

2. Antes de volver a ejecutar `make` con las cuatro variantes enumeradas anteriormente, modificar el fichero `saludo.cpp`. Interpretar el resultado.
3. Probar la orden `touch` sobre `saludo.cpp` y volver a ejecutar `make`. Interpretar el resultado.



Figura 1: Diagrama de dependencias para `saludo`

Ejercicio

A partir del diagrama de dependencias mostrado en la figura 2 construir el fichero makefile llamado `makefile2.mak`.

Ejecutar las siguientes órdenes e interpretar el resultado:

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

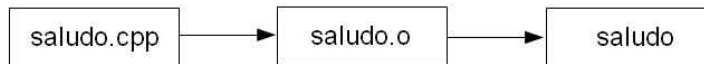


Figura 2: Diagrama de dependencias para `makefile2.mak`

2.3. Órdenes. Prefijos de órdenes

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

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

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

Ejercicio

Copiar el fichero `makefile3.mak` en vuestro directorio de trabajo.

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

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

Ejercicio

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

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

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

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

2.4. Destinos Simbólicos

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

destino simbólico: *lista de destinos*

donde:

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

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

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

Ejercicio

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

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

Ejercicio

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

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

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

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

```
move saludo unico resultado
```

2.5. Destinos .PHONY

Si en un fichero makefile apareciera una regla:

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

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

```
make clean
```

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

```
.PHONY : clean
```

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

```
make clean
```

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

2.6. Macros en ficheros *makefile*

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

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

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

NombreMacro = texto a expandir

donde:

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

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

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

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

La expansión de la macro se hace *recursivamente*. O sea, que si la macro contiene referencias a otras macros, estas referencias serán expandidas también. Veamos un ejemplo. Podemos definir una macro para cada uno de los directorios de trabajo:

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

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

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

se sustituye por:

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

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

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

y se modifican las anteriores por:

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

Ahora, la regla anterior se sustituye por:

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

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

```
HOMEDIR = ~/MP
```

Ejercicio

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

2.7. Macros predefinidas

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

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

Ejercicio

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

A. Sustituciones en macros

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

`$(NombreMacro:TextoOriginal = TextoNuevo)`

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

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

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

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

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

B. Macros como parámetros en la llamada a make

Además de las opciones básicas indicadas en la sección 1.2, hay una opción que permite especificar el valor de una constante simbólica que se emplea en un fichero `makefile` en la llamada a `make` en lugar de especificar su valor en el fichero `makefile`.

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

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

NombreMacro[=*cadena*] define la constante simbólica **NombreMacro** con el valor especificado (si lo hubiera) después del signo `=`. Si *cadena* contiene espacios, será necesario encerrar *cadena* entre comillas.

Si **NombreMacro** también está definida dentro del fichero `makefile`, se ignorará la definición del fichero.

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

Ejercicio

Modificar el fichero `makefile_ppal_2` para que el resultado (el ejecutable `ppal_2`) lo guarde en un directorio cuyo nombre **completo** (camino absoluto, desde la raíz `/`) se indica como parámetro al fichero `makefile` con una macro llamada `DESTDIR`.

Importante: Como el directorio puede no existir, crearlo en el propio fichero `makefile`.

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

Ejercicio

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

C. Reglas implícitas

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

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

Existe un catálogo de reglas implícitas predefinidas que pueden usarse para distintos lenguajes de programación (ver <http://www.gnu.org/software/make/manual/make.html#Implicit-Rules>). El que `make` elija una u otra dependerá del nombre y extensión de los ficheros. Por ejemplo, para el caso que nos interesa, compilación de programas en C++, existe una regla implícita que dice cómo obtener el fichero objeto (`.o`) a partir del fichero fuente (`.cpp`). Esa regla se usa cuando no existe una regla explícita que diga como construir ese módulo objeto. En tal caso se ejecutará la siguiente orden para construir el módulo objeto cuando el fichero fuente sea modificado:

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

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

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

El valor de estas macros pueden ser definidas:

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

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

Ejercicio

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

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

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

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

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

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

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

Ejercicio

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

C.1. 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 carácter `%` 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 carácter `%`. Por ejemplo:

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

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

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

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

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

IMPORTANTE: Una regla 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 puedan encontrarse las dependencias. Cuando se puede aplicar más de una regla implícita, sólo se aplicará una de ellas: la elección depende del orden de las reglas.

C.2. 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*.

D. Directivas condicionales en ficheros *makefile*

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

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

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

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

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

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

Las directivas condicionales para ficheros `makefile` son las siguientes:

`ifndef macro`: Actúa como la directiva `#ifndef` 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 `ifndef`, `ifndef` `ifeq` ó `ifneq`.

Práctica 3. Compilación separada.

Gestión y uso de bibliotecas.

El programa `ar`

Francisco J. Cortijo Bon

Curso 2012-2013

Objetivos

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

Índice

1. La modularización del software en C++	33
1.1. Introducción	33
1.2. Ventajas de la modularización del software	33
1.3. Cómo dividir un programa en varios ficheros	34
1.4. Organización de los ficheros fuente	34
2. Bibliotecas	38
2.1. Tipos de bibliotecas	39
2.2. Estructura de una biblioteca	39
3. El programa <code>ar</code>	42
4. <code>g++</code>, <code>make</code> y <code>ar</code> trabajando conjuntamente	43
4.1. Creación de la biblioteca	43
4.2. Enlazar con una biblioteca	44
5. Ejercicios	45

1. La modularización del software en C++

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

1.2. 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. Pueden reusarse en otros programas, reduciendo el tiempo y coste del desarrollo del software.
4. La compilación de los módulos se puede realizar por separado. Cuando un módulo está validado y compilado no será preciso recompilarlo. Además, cuando haya que modificar el programa, sólo tendremos que recompilar los ficheros fuente alterados por la modificación. La utilidad `make` será muy útil para esta tarea.
5. El ocultamiento de información puede conseguirse con la modularización. El usuario de un módulo objeto o de una biblioteca de módulos desconoce los detalles de implementación de las funciones y objetos definidos en éstos. Mediante el uso de ficheros de cabecera proporcionaremos la interface necesaria para poder usar estas funciones y objetos.

1.3. 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 subtarear que resulten más manejables. Cada una de estas tareas se implementará como una o más funciones. Normalmente, todas las funciones de una subtarea residen en un fichero fuente.

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

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

Cuando las funciones de un módulo invocan objetos o funciones que se encuentran definidos en otros módulos, necesitan alguna información acerca de cómo realizar estas llamadas. El compilador requiere que se proporcionen declaraciones de funciones y/o objetos definidos en otros módulos. La mejor forma de hacerlo (y, probablemente, la única razonable) es mediante la creación de ficheros de

cabecera (.h), que contienen las declaraciones de las funciones definidas en el fichero .cpp correspondiente. De esta forma, cuando un módulo requiera invocar funciones definidas en otros módulos, bastará con que se inserte una línea `#include` del fichero de cabecera apropiado.

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

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

1. **Que la definición de la función se encuentre en el mismo fichero en el que se realiza la llamada.** En este caso,
 - a) se sitúa la definición de la función antes de la llamada (ésta es una solución raramente empleada por los programadores),
 - b) se puede incluir una línea 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 ésta.
2. **Que la definición de la función se encuentre en otro fichero diferente al que contiene la llamada.** En este caso es preciso incluir al principio del mismo una línea de declaración (prototipo) de la función en el fichero que contiene las llamadas a ésta. Normalmente se realiza incluyendo (mediante la directiva de preprocesamiento `#include`) el fichero de cabecera asociado al módulo que contiene la definición de la función.

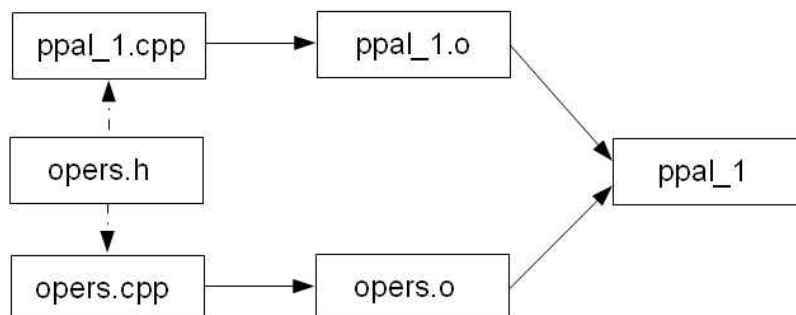
Cuando modularizamos nuestros programas, hemos de hacer un uso adecuado de los ficheros de cabecera asociados a los módulos que estamos desarrollando. Además, los ficheros de cabecera también son útiles para contener las declaraciones de tipos, funciones y otros objetos que necesitemos compartir entre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

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

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

Ejercicio

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

Figura 1: Diagrama de dependencias para construir `ppal_1`**Ejercicio**

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

Los ficheros involucrados y sus dependencias se muestran en la figura 1. Con más detalle:

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

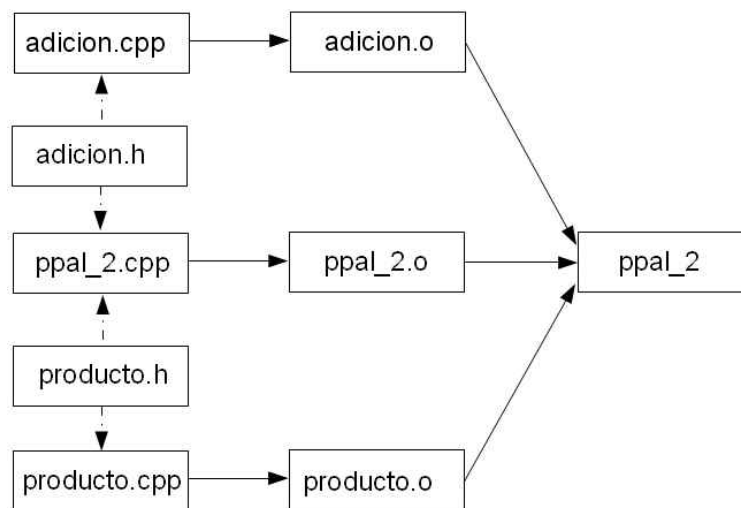
Realizar las siguientes tareas:

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

Ejercicio

Escribir un fichero llamado `makefile_ppal_1` para generar el ejecutable `ppal_1`, siguiendo el diagrama de dependencia mostrado en la figura 1.

Usad macros para especificar los directorios de trabajo.

Figura 2: Diagrama de dependencias para construir `ppal_2`

Ejercicio

Continuaremos distribuyendo el código fuente en distintos ficheros, siguiendo el esquema indicado en la figura 2.

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

Realizar las siguientes tareas:

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

Ejercicio

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

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

Ejercicio

Escribir un fichero llamado `makefile_ppal_2` para generar el ejecutable `ppal_2`, siguiendo el diagrama de dependencia mostrado en la figura 2.

Usad macros para especificar los directorios de trabajo.

2. Bibliotecas

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

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

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

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

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

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

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

2.1. Tipos de bibliotecas

Bibliotecas estáticas

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

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

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

Bibliotecas dinámicas

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

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

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

2.2. Estructura de una biblioteca

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

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

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

1. Está formada por *un único fichero objeto* (por ejemplo, `funcs.o`) que es el resultado de la compilación de un fichero fuente (`funcs.cpp`). Este caso se ilustra en la figura 3.

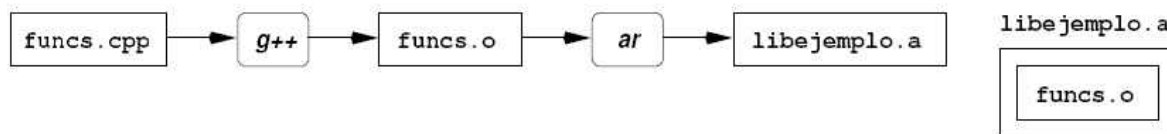


Figura 3: Construcción de una biblioteca a partir de un único módulo objeto (caso 1)

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

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

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

.....

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

2. Está formada por 10 ficheros objeto (por ejemplo, fun01.o, ..., fun10.o) resultado de la compilación de 10 ficheros fuente (por ejemplo, fun01.cpp, ..., fun10.cpp) que contienen, cada uno, la definición de una única función. Este caso se ilustra en las figuras 4 y 5.

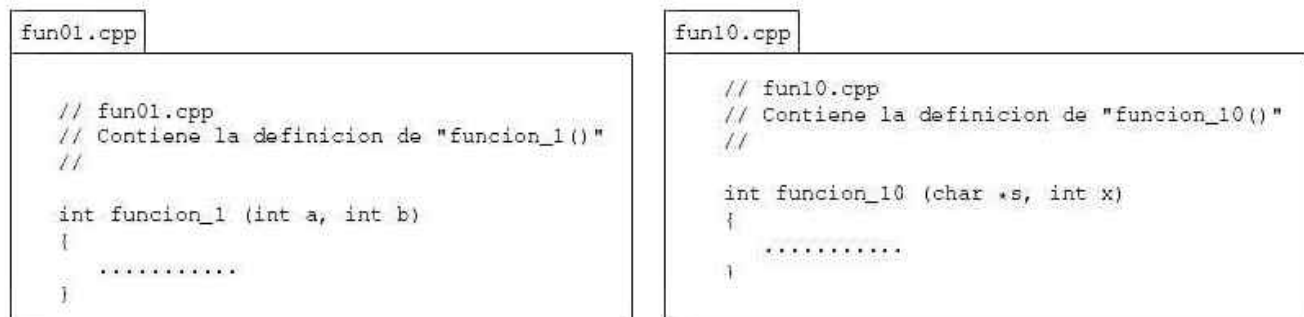


Figura 4: Varios módulos fuente (caso 2)

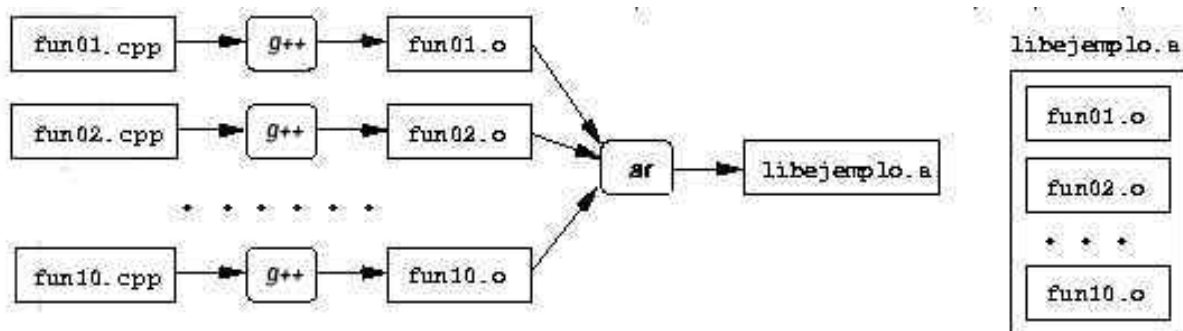


Figura 5: Construcción de una biblioteca a partir de varios módulos objeto (caso 2)

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

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

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

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

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

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

Para la elección de la estructura óptima de la biblioteca hay que recordar que la parte de la biblioteca que se enlaza al código objeto de la función `main()` es el módulo objeto **completo** en el que se encuentra la función de biblioteca usada. En la figura 6 mostramos cómo se construye un ejecutable a partir de un módulo objeto (`ppal.o`) que contiene la función `main()` y una biblioteca (`libejemplo.a`).

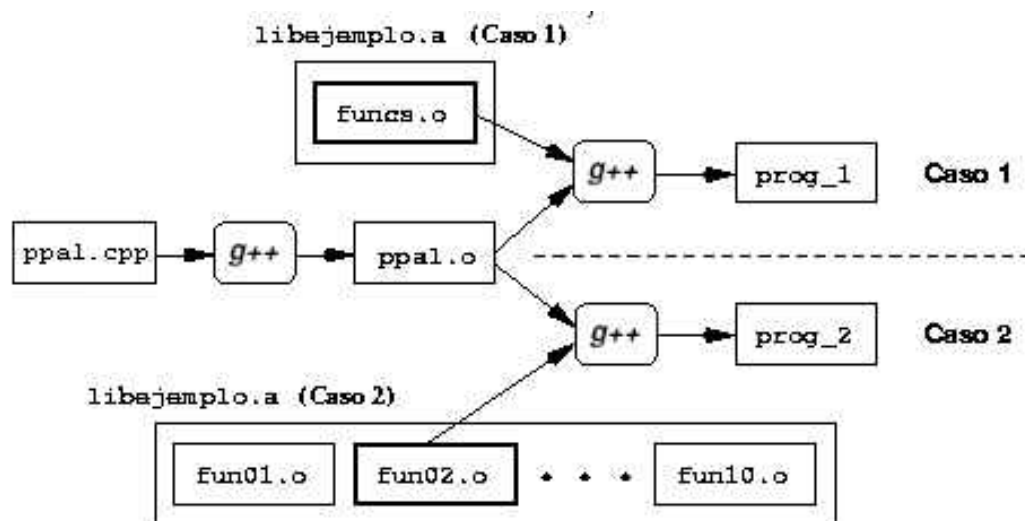


Figura 6: Construcción de un ejecutable que usa una función de biblioteca. Caso 1: biblioteca formada por un módulo objeto. Caso 2: biblioteca formada por 10 módulos objeto

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 construido la biblioteca.

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

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

Como **diseñador** de la biblioteca sí debemos tener en cuenta la estructura que vamos a adoptar para ésta. La idea básica es conocida: si las bibliotecas están formadas por módulos objeto con muchas funciones cada una, los tamaños de los ejecutables serán muy grandes. En cambio, si las bibliotecas están formadas por módulos objeto con pocas funciones cada una, los tamaños de los ejecutables serán más pequeños¹.

3. El programa `ar`

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

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

donde:

- *operación* indica la tarea que se desea realizar sobre la biblioteca. Éstas pueden ser:
 - `r` **Adición o reemplazo.** Reemplaza el módulo objeto de la biblioteca por la nueva versión. Si el módulo no se encuentra en la biblioteca, se añade a la misma. Si se emplea, además, el modificador `v`, `ar` imprimirá una línea por cada módulo añadido o reemplazado, especificando el nombre del módulo y las letras `a` o `r`, respectivamente.
 - `d` **Borrado.** Elimina un módulo de la biblioteca.
 - `x` **Extracción.** Crea el fichero objeto cuyo nombre se especifica y copia su contenido de la biblioteca. La biblioteca queda inalterada.
Si no se especifica ningún nombre, se extraen todos los ficheros de la biblioteca.
 - `t` **Listado.** Proporciona una lista especificando los módulos que componen la biblioteca.
- *modificadores:* Se pueden añadir a las operaciones, modificando de alguna manera el comportamiento por defecto de la operación. Los más importantes (y casi siempre se utilizan) son:
 - `s` **Indexación.** Actualiza o crea (si no existía previamente) el índice de los módulos que componen la biblioteca. *Es necesario que la biblioteca tenga un índice para que el enlazador sepa cómo enlazarla.* Este modificador puede emplearse acompañando a una operación o por sí solo.
 - `v` **Verbose.** Muestra información sobre la operación realizada.
- *biblioteca* es el nombre de la biblioteca a crear o modificar.
- *módulos objeto* es la lista de ficheros objeto que se van a añadir, eliminar, actualizar, etc. en la biblioteca.

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

¹Estas afirmaciones suponen que las funciones son equiparables en tamaño, obviamente

Ejercicio

Repetir los siguientes ejemplos:

1. Crear la biblioteca `libprueba.a` a partir del módulo objeto `opers.o`.

```
ar -rvs lib/libprueba.a obj/opers.o
```

2. Mostrar los módulos que la componen.

```
ar -tv lib/libprueba.a
```

Ejercicio

1. Añadir los módulos `adicion.o` y `producto.o` a la biblioteca `libprueba.a`.

```
ar -rvs lib/libprueba.a obj/adicion.o obj/producto.o
```

2. Mostrar los módulos que la componen.

```
ar -tv lib/libprueba.a
```

Ejercicio

1. Extraer el módulo `adicion.o` de la biblioteca `libprueba.a`.

```
ar -xvs lib/libprueba.a adicion.o
```

2. Mostrar los módulos que la componen.

```
ar -tv lib/libprueba.a
```

3. Mostrar el directorio actual.

Ejercicio

1. Borrar el módulo `producto.o` de la biblioteca `libprueba.a`.

```
ar -dvs lib/libprueba.a producto.o
```

2. Mostrar los módulos que la componen.

```
ar -tv lib/libprueba.a
```

3. Mostrar el directorio actual.

4. g++, make y ar trabajando conjuntamente

Como hemos señalado, `ar` es un programa general que empaqueta/desempaqueta ficheros en/desde archivos, de manera similar a como hacen otros programas como `zip`, `rar`, `tar`, etc. (una lista amplia puede encontrarse en http://es.wikipedia.org/wiki/Anexo:Archivadores_de_ficheros).

Nuestro interés es aprender cómo puede emplearse una biblioteca para la generación de un ejecutable, y cómo escribir las dependencias en ficheros `makefile` para poder automatizar todo el proceso de creación/actualización con la orden `make`.

4.1. Creación de la biblioteca

Emplearemos una regla para la construcción de la biblioteca.

1. El *objetivo* será la biblioteca a construir.
2. La *lista de dependencias* estará formada por los módulos objeto que constituirán la biblioteca.

Para los dos casos estudiados en la sección 2.2 escribiríamos:

1. Caso 1.

```
lib/libejemplo.a : obj/funcs.o
ar -rvs /libejemplo.a obj/funcs.o
```

2. Caso 2.

```
lib/libejemplo.a : obj/fun01.o obj/fun02.o obj/fun03.o obj/fun04.o\
obj/fun05.o obj/fun06.o obj/fun07.o obj/fun08.o\
obj/fun09.o obj/fun10.o
ar -rvs lib/libejemplo.a obj/fun01.o obj/fun02.o obj/fun03.o\
obj/fun04.o obj/fun05.o obj/fun06.o obj/fun07.o\
obj/fun08.o obj/fun09.o obj/fun10.o
```

(La barra simple invertida (\) es muy útil para dividir en varias líneas órdenes muy largas ya que permite continuar escribiendo en una nueva línea la misma orden)

Evidentemente, resulta aconsejable escribir de manera más compacta y generalizable esta regla:

```
MODS_EJEMPLO = obj/fun01.o obj/fun02.o obj/fun03.o obj/fun04.o\
obj/fun05.o obj/fun06.o obj/fun07.o obj/fun08.o\
obj/fun09.o obj/fun10.o
.....
lib/libejemplo.a : $(MODS_EJEMPLO)
ar -rvs lib/libejemplo.a $(MODS_EJEMPLO)
```

4.2. Enlazar con una biblioteca

El enlace lo realiza `g++`, llamando al enlazador `ld`. Extenderemos la regla que genera el ejecutable:

1. El *objetivo* es el mismo: generar el ejecutable.
2. La *lista de dependencias* incluirá ahora a la biblioteca que se va a enlazar.
3. La *orden* debe especificar:
 - a) El directorio dónde buscar la biblioteca (opción `-L`)
Recordemos que la opción `-Lpath` indica al enlazador el directorio donde se encuentran los ficheros de biblioteca. Se puede utilizar la opción `-L` varias veces para especificar distintos directorios de biblioteca.
 - b) El nombre (resumido) de la biblioteca (opción `-l`).
Los ficheros de biblioteca se proporcionan a `g++` de manera resumida, escribiendo `-lnombre` para referirnos al fichero de biblioteca `libnombre.a`. El enlazador busca en los directorios de bibliotecas (entre los que están los especificados con `-L`) un fichero de biblioteca llamado `libnombre.a` y lo usa para enlazarlo.

Para los dos casos estudiados en la sección 2.2 escribiríamos:

1. Caso 1.

```
bin/prog_1 : obj/ppal.o lib/libejemplo.a
g++ -o bin/prog_1 obj/ppal.o -L./lib -lejemplo
```

2. Caso 2.

```
bin/prog_2 : obj/ppal.o lib/libejemplo.a
g++ -o bin/prog_2 obj/ppal.o -L./lib -lejemplo
```

5. Ejercicios

Para poder realizar los ejercicios propuestos es necesario disponer de los ficheros:

- `ppal_1.cpp`, `ppal_2.cpp`, `opers.cpp`, `adicion.cpp` y `producto.cpp`

Usaremos un nuevo fichero fuente, `ppal.cpp`, que será una copia de `ppal_1.cpp`. Este fichero se usará en todos los ejercicios, y únicamente cambiará(n) la(s) línea(s) `#include`

- `opers.h`, `adicion.h` y `producto.h`
- `makefile_ppal_1` y `makefile_ppal_2`

En todos los ejercicios debe poder diferenciar claramente los dos actores que pueden intervenir:

1. El creador de la biblioteca
2. El usuario de la biblioteca

aunque sea la misma persona (en este caso, usted) quien realice las dos tareas. Esta distinción es fundamental cuando se maneje los ficheros de cabecera: puede llegar a manejar dos ficheros exactamente iguales aunque con nombre diferentes. Uno de ellos será empleado por el creador de la biblioteca y una vez construida la biblioteca, proporcionará otro fichero de cabecera que sirva de interface de la biblioteca a los usuarios de la misma.

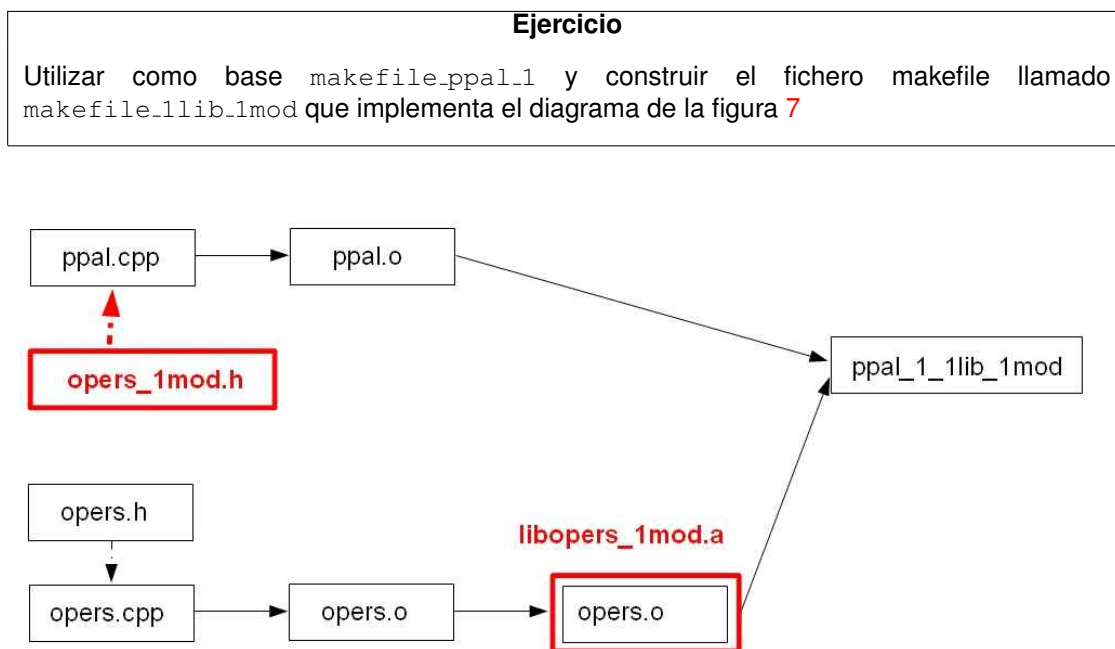


Figura 7: Construcción de un ejecutable que usa funciones de biblioteca (1). Una biblioteca formada por un módulo objeto que implementa cuatro funciones.

En este caso tenemos una biblioteca formada por un único módulo objeto. Todas las funciones están definidas en ese módulo objeto. Observe que el ejecutable será exactamente igual que el que llamamos anteriormente `ppal_1` (ver figura 3 de la práctica 2). Explique por qué.

El fichero de cabecera `opers_1mod.h` asociado a la biblioteca `libopers_1mod.a` podría tener el mismo contenido que el que se emplea para construir la biblioteca.

Ejercicio

Utilizar como base `makefile_ppal_2` y construir el fichero `makefile` llamado `makefile_l1lib_2mod` que implementa el diagrama de la figura 8

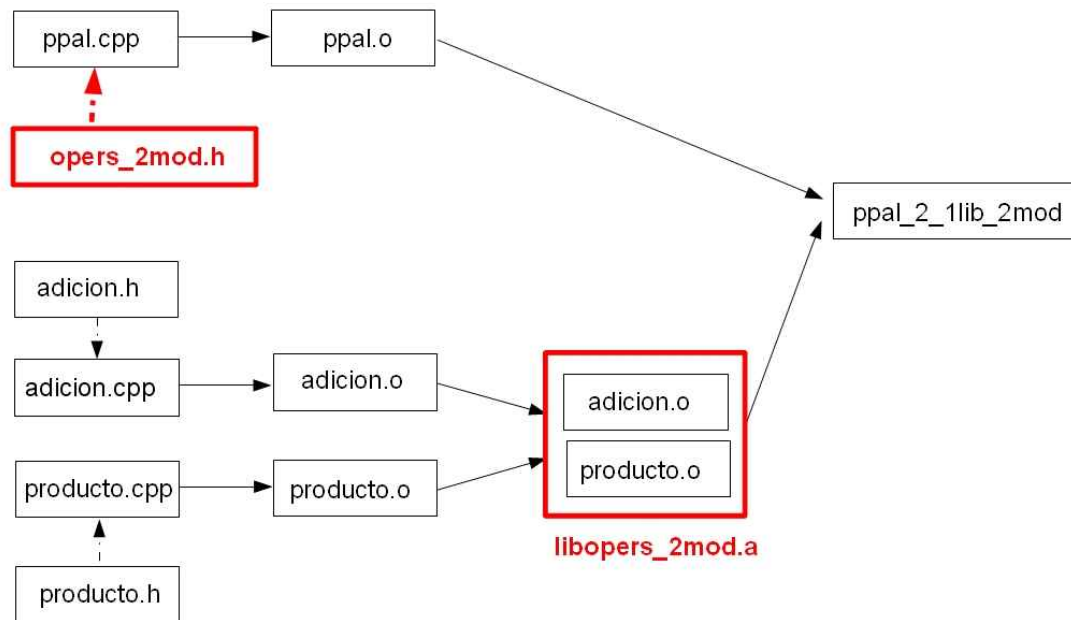


Figura 8: Construcción de un ejecutable que usa funciones de biblioteca (2). Una biblioteca formada por dos módulos objeto que implementan dos funciones cada uno de ellos.

En este caso tenemos una biblioteca formada por dos módulos objeto. Cada uno define dos funciones. Observe que el ejecutable será exactamente igual que el que llamamos anteriormente `ppal_2` (ver figura 4 de la práctica 2). Explique por qué.

El fichero de cabecera `opers_2mod.h` asociado a la biblioteca `libopers_2mod.a` podría tener el mismo contenido que `opers_1mod.h` (ejercicio anterior) si se pretende ofrecer la misma funcionalidad.

Ejercicio

Construir el fichero makefile llamado `makefile_1lib_4mod`. El fichero makefile implementa el diagrama de dependencias mostrado en la figura 9. En este caso tenemos una biblioteca formada por cuatro módulos objeto, y cada uno define una única función.

Como trabajo previo deberá crear los ficheros fuente `suma.cpp`, `resta.cpp`, `multiplica.cpp` y `divide.cpp`. Observe que no se han considerado ficheros de cabecera para cada uno de éstos ¿por qué?

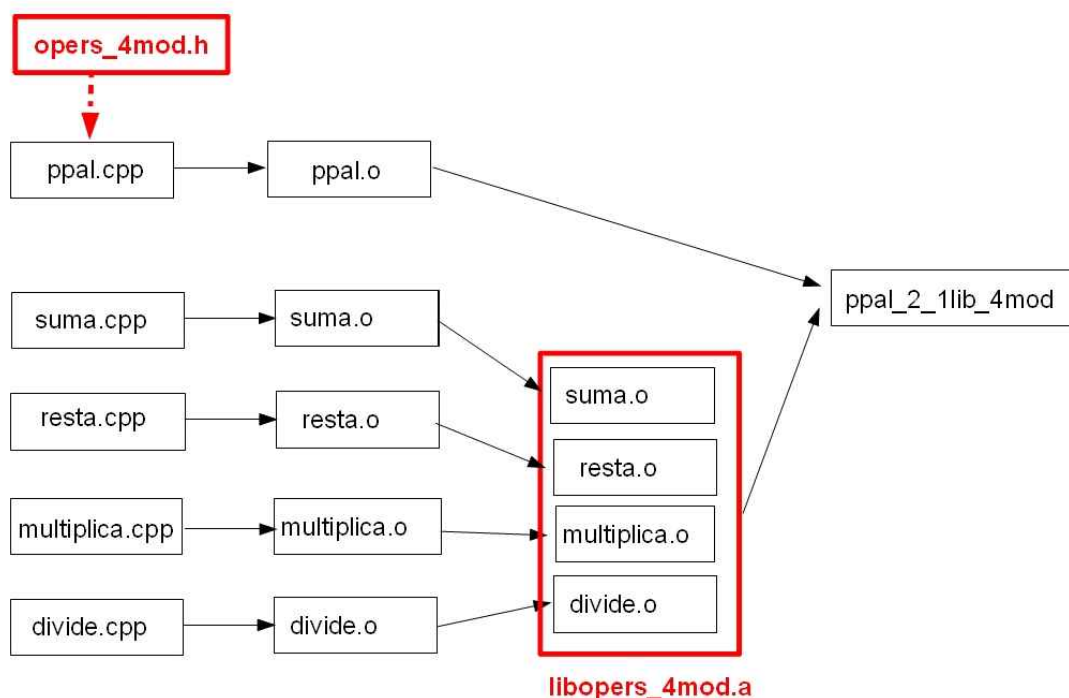


Figura 9: Construcción de un ejecutable que usa funciones de biblioteca (3). Una biblioteca formada por cuatro módulos objeto, y cada uno define una única función.

El fichero de cabecera `opers_4mod.h` asociado a la biblioteca `libopers_4mod.a` podría tener el mismo contenido que `opers_2mod.h` y `opers_1mod.h` (ejercicios anteriores) si se pretende ofrecer la misma funcionalidad.

Ejercicio

Construir el fichero makefile llamado `makefile_2lib_2mod`. El fichero makefile implementa el diagrama de dependencias mostrado en la figura 10

Observad que el ejecutable resultante debe ser exactamente igual que el anterior ¿por qué?

En este caso tenemos dos bibliotecas (`libadic_2mod.a` y `libproducto_2mod.a`) con sus ficheros de cabecera asociados (`adic_2mod.h` y `producto_2mod.h`). Cada una de las bibliotecas está compuesta de dos módulos objeto, y cada uno define dos funciones.

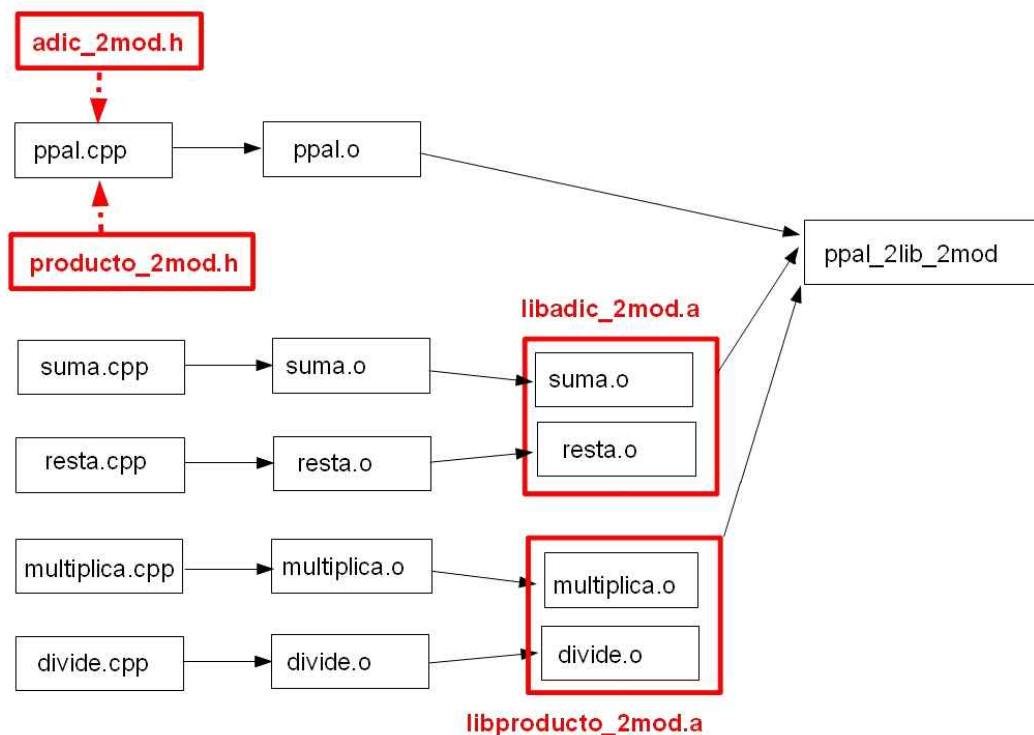


Figura 10: Construcción de un ejecutable que usa funciones de biblioteca (4). Dos bibliotecas formadas cada una por dos módulos objeto, y cada uno define dos funciones.

Ejercicio

1. Tomar nota de los tamaños de los ejecutables `ppal_1lib_1mod`, `ppal_1lib_2mod`, `ppal_1lib_4mod` y `ppal_2lib_2mod`.
2. Editar el fichero `ppal.cpp` y comentar la línea que llama a la función `suma()`.
3. Volver a construir los ejecutables `ppal_1lib_1mod`, `ppal_1lib_2mod`, `ppal_1lib_4mod` y `ppal_2lib_2mod`.
4. Anotar los tamaños de los ejecutables, compararlos con los anteriores y discutir.

Ejercicio

En la *Relación de problemas I (punteros)* propusimos escribir unas funciones para la gestión de cadenas clásicas de caracteres.

Los prototipos de las funciones son:

```
int longitud_cadena (const char * cadena);  
bool es_palindromo (const char * cad);  
int comparar_cadenas (const char * cad1, const char * cad2);  
char * copiar_cadena (char * cad1, const char * cad2);  
char * encadenar_cadena (char * cad1, const char * cad2);
```

1. Encapsular estas funciones en una biblioteca llamada `libmiscadenas.a` y escribir un módulo llamado `demo_cadenas.cpp` que contenga únicamente una función `main()` y que use las cuatro funciones.

No olvide escribir el fichero de cabecera asociado a la biblioteca, y llamarle `miscadenas.h`

2. Escribir un fichero *makefile* para generar la biblioteca y el ejecutable.
3. Añadir a la biblioteca la función propuesta para extraer una subcadena, con prototipo

```
char * subcadena (char * cad1, const char * cad2, int p, int l);
```

4. Editar `demo_cadenas.cpp` para que use la nueva función.
5. Volver a generar el ejecutable.
6. ¿Qué puede decirse del fichero de cabecera asociado a la biblioteca?

Práctica 4. Gestión de memoria dinámica (I). Resolución de la Relación de Problemas II.

Francisco J. Cortijo Bon

Marzo de 2013

Objetivos

1. Reservar memoria en el heap y liberarla.
2. Gestionar estructuras tipo *lista*.
3. Gestionar vectores dinámicos
4. Seguir profundizando en la compilación separada de programas.

Se trabajará sobre la **Relación de Problemas II: Memoria dinámica**. Los problemas propuestos en esta relación se han organizado en orden creciente de dificultad y tiempo de resolución. Proponemos organizar la resolución de los problemas en dos bloques:

1. En el primero (ejercicios 1 a 7) se trabaja sobre estructuras dinámicas lineales simples (listas y vectores dinámicos). Este será el trabajo a desarrollar en esta práctica.
2. En el segundo (ejercicios 8 a 10) se trabaja con estructuras más complicadas (matrices bidimensionales y listas con información compleja)

Trabajo a realizar

* Ejercicios obligatorios:

Se entregarán en la fecha, hora, lugar y siguiendo el procedimiento que se anunciará en la página web de la asignatura y/o correo electrónico.

Los ejercicios obligatorios son: 1, 2, 3, 4 y 5.

* Ejercicios opcionales:

Opcionalmente podrá entregarse el ejercicio 6.

* Trabajo a realizar en el aula de prácticas:

Mientras los compañeros defienden los ejercicios entregados, los demás deben trabajar en la resolución del ejercicio 7.

Recomendamos pensar en la solución a los ejercicios usando clases.

Práctica 5. Gestión de memoria dinámica (II). Resolución de la Relación de Problemas II.

Francisco J. Cortijo Bon

Abril de 2013

Objetivos

1. Seguir practicando con la reserva y liberación de memoria en el heap.
2. Gestionar estructuras dinámicas bidimensionales (tipo *matriz*).
3. Gestionar estructuras de datos con información compleja.
4. Seguir profundizando en la compilación separada de programas.

Se trabajará sobre la **Relación de Problemas II: Memoria dinámica**. Los problemas propuestos en esta relación se han organizado en orden creciente de dificultad y tiempo de resolución. Hemos organizado la resolución de los problemas en dos bloques:

1. En el primero (ejercicios 1 a 7) se trabaja sobre estructuras dinámicas lineales simples (listas y vectores dinámicos). Estos ejercicios se resolvieron en la práctica 4.
2. En el segundo (ejercicios 8 a 10) se trabaja con estructuras más complicadas (matrices bidimensionales y listas con información compleja). Este será el trabajo a desarrollar en esta práctica.

Trabajo a realizar

* Ejercicios obligatorios:

Se entregarán en la fecha, hora, lugar y siguiendo el procedimiento que se anunciará en la página web de la asignatura y/o correo electrónico.

Los ejercicios obligatorios son: 8 (completo) y 9 (apartado a).

* Ejercicios opcionales:

Opcionalmente podrán entregarse los apartados b y c del ejercicio 9.

* Trabajo a realizar en el aula de prácticas:

Mientras los compañeros defienden los ejercicios entregados, los demás deben trabajar en la resolución del ejercicio 10.

Recomendamos pensar en la solución a los ejercicios usando clases.

Práctica 6. Gestión de memoria dinámica (III). Resolución de la Relación de Problemas II.

Francisco J. Cortijo Bon

Abril de 2013

Objetivos

1. Seguir practicando con la reserva y liberación de memoria en el heap.
2. Gestionar estructuras dinámicas bidimensionales (tipo *matriz*).
3. Gestionar estructuras de datos con información compleja.
4. Seguir profundizando en la compilación separada de programas.

Se trabajará sobre la **Relación de Problemas II: Memoria dinámica**. Los problemas propuestos en esta relación se han organizado en orden creciente de dificultad y tiempo de resolución. Hemos organizado la resolución de los problemas en dos bloques:

1. En el primero (ejercicios 1 a 7) se trabaja sobre estructuras dinámicas lineales simples (listas y vectores dinámicos). Estos ejercicios se resolvieron en la práctica 4.
2. En el segundo (ejercicios 8 a 10) se trabaja con estructuras más complicadas (matrices bidimensionales y listas con información compleja). Este es el trabajo que se empezó a realizar en la práctica 5 y que se completa en esta práctica.

Trabajo a realizar

* Ejercicios obligatorios:

Se entregarán en la fecha, hora, lugar y siguiendo el procedimiento que se anunciará en la página web de la asignatura y/o correo electrónico.

Los ejercicios obligatorios son:

- Apartados b y c del ejercicio 9.
- Ejercicio 10 (completo). Se valorará la modularidad y generalidad de la solución. Escribid una función `main()` que permita comprobar el funcionamiento de los módulos.

Recomendamos pensar en la solución a los ejercicios usando clases.

Práctica 7. Clases (I).

Resolución de la Relación de Problemas III.

Francisco J. Cortijo Bon

Abril de 2013

Objetivos

1. Seguir profundizando en la compilación separada de programas.
2. Trabajar en el encapsulamiento de datos y métodos en clases.
3. Practicar con constructores y el destructor.
4. Practicar los mecanismos de ocultación de información.
5. Escribir métodos sencillos de acceso y manipulación de la clase.
6. Continuar trabajando con la memoria dinámica.
7. Continuar con el trabajo sobre estructuras de datos dinámicas con nodos enlazados.
8. Continuar con el trabajo sobre vectores y matrices dinámicas

Se trabajará sobre la **Relación de Problemas III: Clases (I)**. Los problemas propuestos en esta relación se han organizado en orden creciente de dificultad y tiempo de resolución.

Observarán que los ejercicios 1, 2, 3 y 4 trabajan sobre estructuras de datos conocidas (vectores dinámicos, matrices bidimensionales dinámicas y listas enlazadas) y sobre las que ya se ha practicado con ejercicios publicados en las relaciones de problemas I y II.

Trabajo a realizar

* Ejercicios obligatorios:

Se entregarán en la fecha, hora, lugar y siguiendo el procedimiento que se anunciará en la página web de la asignatura y/o correo electrónico.

Los ejercicios obligatorios son: 1, 2, 3 y 4.

* Ejercicios opcionales:

Opcionalmente podrán entregarse los ejercicios 5 y 6.

Para la resolución de estos ejercicios puede emplearse una representación basada en un vector dinámico o en una lista enlazada (recomendable).

Cuidado: no sugerimos que se usen las clases `VectorDinamico` o `Lista`, sino que se emplee una **nueva** estructura dinámica tipo vector o basada en una serie de nodos enlazados.

* **Trabajo a realizar en el aula de prácticas:**

Si no se han entregado como ejercicios opcionales, se trabajará en la resolución de los ejercicios 5 y 6 siguiendo las recomendaciones indicadas en la propuesta de ejercicios opcionales.

Práctica 8. Clases (II).

Resolución de la Relación de Problemas IV.

Francisco J. Cortijo Bon

Abril de 2012

Objetivos

1. Seguir profundizando en la compilación separada de programas.
2. Continuar trabajando con la memoria dinámica.
3. Continuar con el trabajo sobre estructuras de datos dinámicas con nodos enlazados.
4. Continuar con el trabajo sobre vectores y matrices dinámicas
5. Continuar trabajando en el encapsulamiento de datos y métodos en clases.
6. Continuar trabajando en los mecanismos de ocultación de información.
7. Escribir métodos de acceso y manipulación de la clase.
8. Los objetivos fundamentales de esta práctica son:
 - a) Escribir el **constructor de copia** y la sobrecarga del **operador de asignación**.
 - b) Promover el uso de funciones privadas auxiliares para modularizar las tareas comunes entre los constructores, destructor y operador de asignación.
9. **Diseñar e implementar un nuevo tipo de dato, el tipo Conjunto.**

Se trabajará sobre la **Relación de Problemas IV: Clases (II)**. Los problemas propuestos en esta relación son ampliación de los ejercicios propuestos en la **Relación de Problemas III: Clases (I)**, por lo que éstos deberían estar resueltos previamente.

Trabajo a realizar

* Ejercicios obligatorios:

Se entregarán en la fecha, hora, lugar y siguiendo el procedimiento que se anunciará en la página web de la asignatura y/o correo electrónico.

Los ejercicios obligatorios son:

- 1: apartado a.
- 2: apartados a y b.
- 3: apartados a y b.

- 4: apartado a.

*** Ejercicios opcionales:**

- 5: apartado a.
- 6: apartado a.
- 7: apartados a, b, c, d, e, f, g y h.

*** Trabajo a realizar en el aula de prácticas:**

Si no se han entregado los ejercicios opcionales, se trabajará en su resolución.

Práctica 9. Clases (II).

Resolución de la Relación de Problemas IV (2).

Francisco J. Cortijo Bon

Mayo de 2013

Objetivos

1. Seguir profundizando en la compilación separada de programas.
 2. Continuar trabajando con la memoria dinámica.
 3. Continuar con el trabajo sobre estructuras de datos dinámicas con nodos enlazados.
 4. Continuar con el trabajo sobre vectores y matrices dinámicas
 5. Continuar trabajando en el encapsulamiento de datos y métodos en clases.
 6. Continuar trabajando en los mecanismos de ocultación de información.
 7. Escribir métodos de acceso y manipulación de la clase, incluyendo los métodos constructores y el destructor.
 8. Los objetivos fundamentales de esta práctica son:
 - a) Practicar con la sobrecarga de los **operadores aritméticos** y **operadores lógicos** habituales, promoviendo la independencia y el acoplamiento bajo entre los métodos desarrollados.
 - b) Practicar con la sobrecarga del **operador de inserción en flujo** y del **operador de extracción de flujo**.
 9. **Terminar el diseño e implementación del tipo Conjunto.**
-

Se finalizará el trabajo propuesto en la **Relación de Problemas IV: Clases (II)**. Los problemas propuestos en esta relación son los que quedaron pendientes de resolver en la práctica 8.

Trabajo a realizar

* Ejercicios obligatorios:

Se entregarán en la fecha, hora, lugar y siguiendo el procedimiento que se anunciará en la página web de la asignatura y/o correo electrónico.

Los ejercicios obligatorios son:

- 1: apartados b, c, d y f.
- 2: apartados c, d, e y f.
- 3: apartados c, d, e y f.
- 4: apartados b y c.

*** Ejercicios opcionales:**

- 5: apartado b.
- 6: apartado b.
- 7: apartados i, j, k, l, m y n.

*** Trabajo a realizar en el aula de prácticas:**

Si no se han entregado los ejercicios opcionales, se trabajará en su resolución.

.