



DECSAI

Departamento de Ciencias de la Computación e I.A.

Universidad de Granada

METODOLOGÍA DE LA PROGRAMACIÓN

Grado en Ingeniería Informática

GRUPO A

Profesor: Francisco José Cortijo Bon

`cb@decsai.ugr.es`



Guión de Prácticas

Curso 2017/2018

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

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



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



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



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



Índice del Guión de Prácticas

| | |
|---|----|
| SESIÓN 1 | 5 |
| Objetivos | 5 |
| El modelo de compilación en C++ | 5 |
| g++ : el compilador de GNU para C++ | 10 |
| Resumen: un ejemplo completo | 17 |
| Introducción al depurador DDD | 19 |
| El preprocesador de C++ | 24 |
| SESIÓN 2 | 31 |
| Objetivos | 31 |
| Gestión de un proyecto software | 31 |
| El programa <i>make</i> | 32 |
| Ficheros <i>makefile</i> | 34 |
| Sustituciones en macros | 44 |
| Macros como parámetros en la llamada a <i>make</i> | 44 |
| Reglas implícitas | 45 |
| Directivas condicionales en ficheros <i>makefile</i> | 49 |
| SESIÓN 3 | 51 |
| Objetivos | 51 |
| La modularización del software en C++ | 51 |
| Bibliotecas | 57 |
| El programa <i>ar</i> | 63 |
| g++, <i>make</i> y <i>ar</i> trabajando conjuntamente | 65 |
| Ejercicios | 67 |
| SESIÓN 4 | 73 |
| Punteros | 73 |
| SESIÓN 5 | 77 |
| Punteros y funciones | 77 |
| SESIÓN 6 | 81 |

| | |
|--|----------|
| Gestión de memoria dinámica (1) | 81 |
| SESIÓN 7 | 83 |
| Gestión de memoria dinámica (2) | 83 |
| SESIÓN 8 | 87 |
| Gestión de memoria dinámica (3) | 87 |
| SESIÓN 9 | 91 |
| Clases (I): El constructor de copia y el destructor | 91 |
| SESIÓN 10 | 95 |
| Clases (II): Sobrecarga de operadores | 95 |
| SESIÓN 11 | 99 |
| Gestión de E/S | 99 |
| SESIÓN 12 | 101 |
| Flujos asociados a <code>string</code> . Ficheros de texto | 101 |
| SESIÓN 13 | 103 |
| Ficheros binarios | 103 |
| RELACIÓN DE PROBLEMAS I. Punteros | RP-I.1 |
| RELACIÓN DE PROBLEMAS II. Memoria dinámica | RP-II.1 |
| RELACIÓN DE PROBLEMAS III. Clases (I) | RP-III.1 |
| RELACIÓN DE PROBLEMAS IV. Clases (II) | RP-IV.1 |
| RELACIÓN DE PROBLEMAS V. Gestión de E/S. | RP-V.1 |
| RELACIÓN DE PROBLEMAS VI. Ficheros (I) | RP-VI.1 |
| RELACIÓN DE PROBLEMAS VII. Ficheros (II) | RP-VII.1 |

Sesión 1

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.

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.

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.

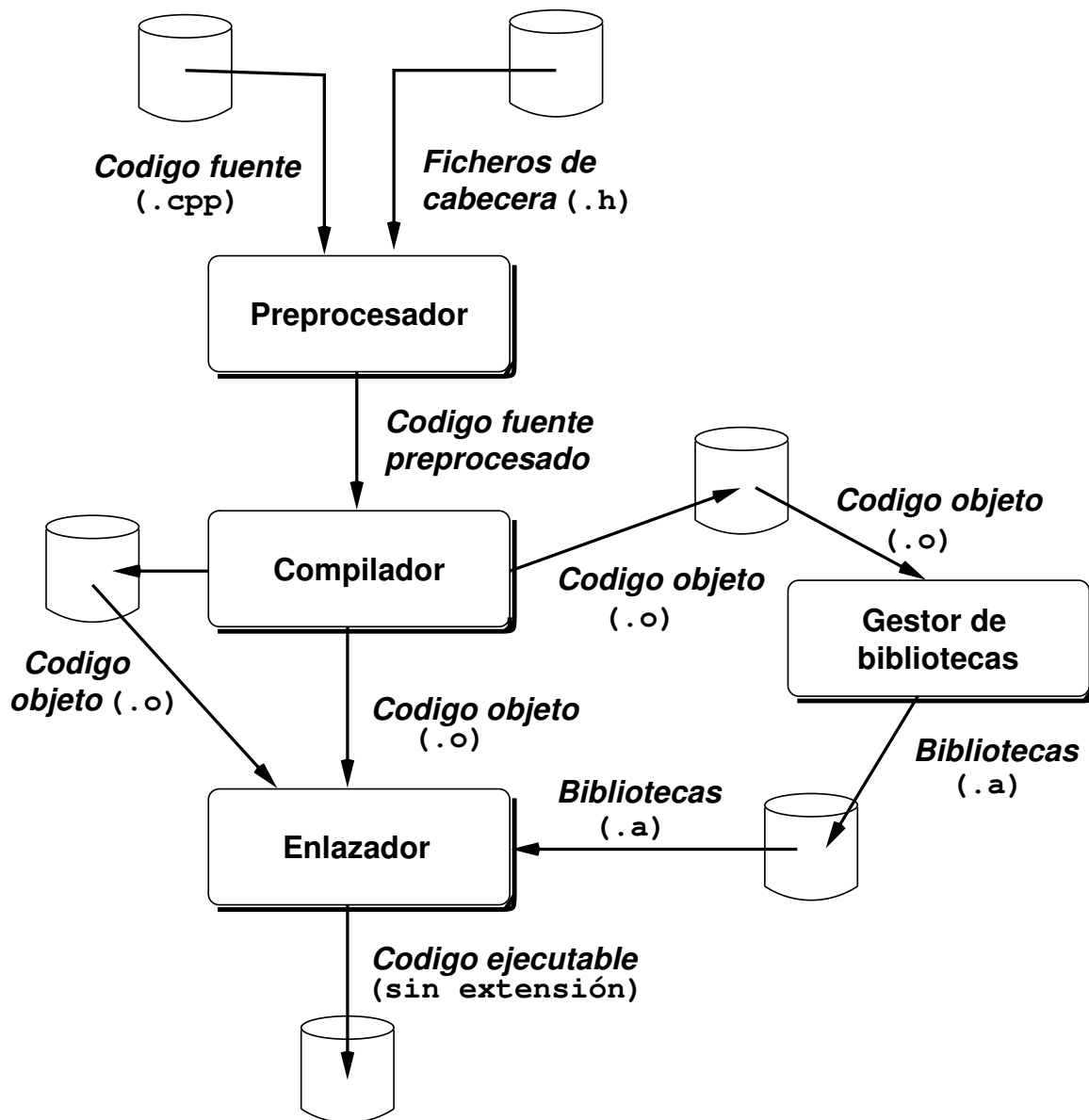


Figura 1: El proceso de compilación (generación de programas ejecutables) en C++

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 (`~`)

El preprocesador

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

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

Dos de las directivas más comúnmente empleadas en C++ son `#include` y `#define`. En la sección tratamos con más profundidad estas directivas y algunas otras más complejas. En cualquier caso, retomando el esquema mostrado en la figura 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.

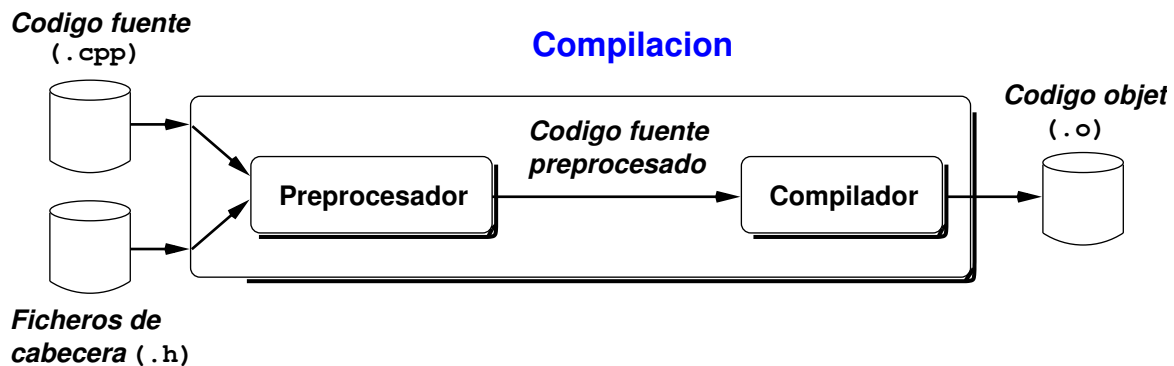


Figura 2: Fase de compilación

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.

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*).

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.

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

Un poco de historia

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

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

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

GNU es un acrónimo recursivo que significa **GNU No es Unix** (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.

Sintaxis

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

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

donde:

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

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

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

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

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

Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

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

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

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

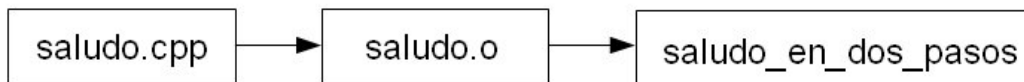


Figura 4: Diagrama de dependencias para saludo_en_dos_pasos

- 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`.

- 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).

Resumen: un ejemplo completo

Los ficheros necesarios para realizar este ejercicio puede encontrarlos en `decsai.ugr.es` (sección *material de la asignatura*). El fichero `demo.cpp` se copiará en la carpeta `src`, el fichero `utils.h` en la carpeta `include` y el fichero `libutils.a` en la carpeta `lib`.

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

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

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

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

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

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

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

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

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

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

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

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

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

Introducción al depurador DDD

Conceptos básicos

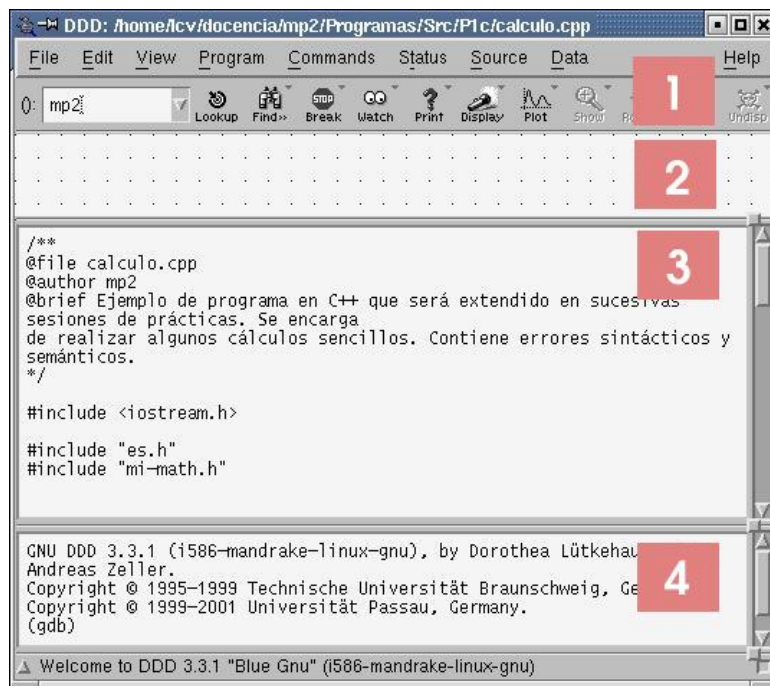
El programa ddd es, básicamente, una interfaz (*front-end*) separada que se puede utilizar con un depurador en línea de órdenes. En el caso que concierne a este documento, ddd será la interfaz de alto nivel del depurador gdb.

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

```
ddd fichero-binario
```

Pantalla principal

La pantalla principal del depurador se muestra en la figura 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 ventala 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.

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.

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

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 .

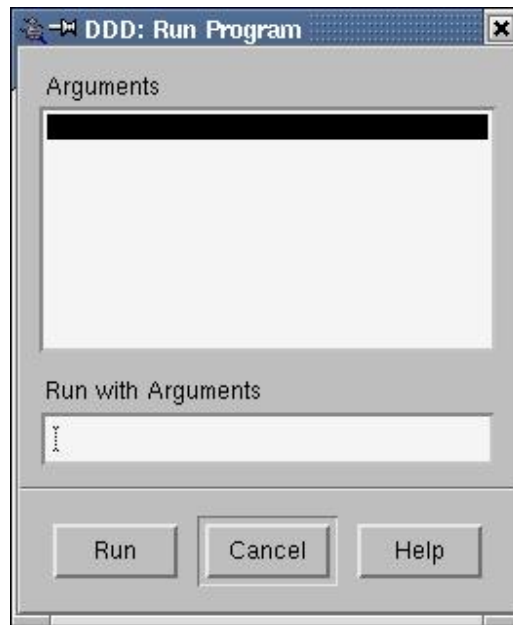


Figura 6: Ventana para pasar argumentos a `main`

estuviese visible, entonces puede mostrarse pulsando en menú Program - Run in execution window.

Inspección y modificación de datos

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

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

| 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

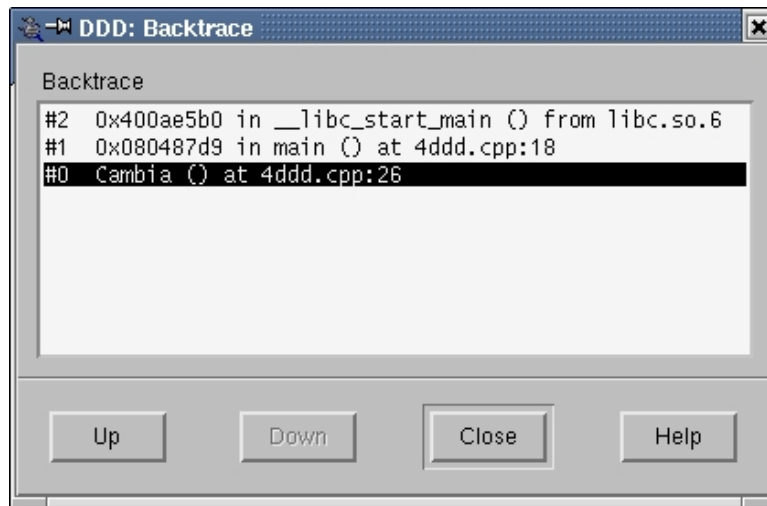


Figura 7: Ventana que muestra el estado de la pilla de llamadas a módulos

programa (ver cuadro 1).

Mantenimiento de sesiones de depuración

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

Reparación del código

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

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.

Constantes simbólicas y macros funcionales

Constantes simbólicas

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

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

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

Veamos algunos ejemplos sencillos.

```
#define TAMMAX 256
```

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

```
#define UTIL_VEC
```

simplemente define la constante simbólica UTIL_VEC, aunque sin asignarle ningún valor de sustitución: se puede interpretar como una “bandera” (existe/no existe). Se suele emplear para la inclusión condicional de código (ver página 27 de este apéndice).

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

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

Macros funcionales (con argumentos)

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

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

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

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

y tenemos la sentencia

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

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

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

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

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

con lo que se avalúa la suma antes del producto.

Veamos ahora algunos ejemplos adicionales.

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

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

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

Calcula la diferencia absoluta entre dos operandos.

Eliminación de constantes simbólicas

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

Inclusión de ficheros

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

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

La única diferencia es que `<fichero>` indica que el fichero se encuentra en alguno de los directorios de ficheros de cabecera del sistema o entre los especificados como directorios de ficheros de cabecera (opción `-I` del compilador: ver página 13), 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.

Inclusión condicional de código

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

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

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

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

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

#include CABECERA
```

De esta forma, estamos seguros de que incluiremos el fichero de cabecera apropiado al sistema en el que estemos compilando. Por supuesto, debemos especificar de alguna forma el valor de la constante `SISTEMA` (por ejemplo, usando macros en la llamada al compilador, como indicamos en la página 13).

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.

Sesión 2

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`.

Gestión de un proyecto software

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.

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.

Sintaxis

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

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

donde:

- cada **opción** va precedida por un signo - o una barra inclinada /.
- **destino** indica el destino que debe crear. Generalmente se trata del fichero que debe crear o actualizar, estando especificado en el fichero makefile el procedimiento de creación/actualización del mismo (página 38). Una explicación detallada de los destinos puede encontrarse en las páginas 35 y 39.

Obsérvese que tanto las opciones como los destinos son opcionales, por lo que podría ejecutarse make sin más.

Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

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

Funcionamiento de make

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

1. En primer lugar, busca el fichero `makefile` que debe interpretar. Si se ha especificado la opción `-f fichero`, busca ese fichero. Si no, busca en el directorio actual un fichero llamado `makefile` ó `Makefile`. En cualquier caso, si lo encuentra, lo interpreta; si no, da un mensaje de error y termina.
2. Intenta construir el(los) destino(s) especificado(s). Si no se proporciona ningún destino, intenta construir *solamente* el primer destino que aparece en el fichero `makefile`. Para construir un destino es posible que deba construir antes otros destinos si el destino especificado depende de otros que no están 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.

Ficheros makefile

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

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

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

Comentarios

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

Ejercicio

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

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

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

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

```
% make
```

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

```
% make -f makefile
```

Reglas. Reglas explícitas

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

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

El formato habitual de una regla explícita es el siguiente:

objetivo: lista de dependencia
orden(es)

donde:

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

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

- Las **orden(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 8. 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 8: Diagrama de dependencias para saludo

Ejercicio

A partir del diagrama de dependencias mostrado en la figura 9 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
```

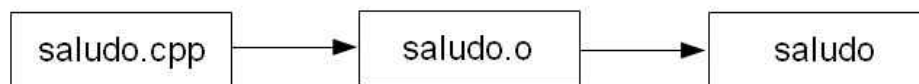


Figura 9: Diagrama de dependencias para `makefile2.mak`

Órdenes. Prefijos de órdenes

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

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

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

Ejercicio

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

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

Ejercicio

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

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

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

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


Destinos Simbólicos

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

destino simbólico: *lista de destinos*

donde:

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

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

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

Ejercicio

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

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

Ejercicio

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

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

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

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

```
move saludo unico resultado
```

Destinos .PHONY

Si en un fichero makefile apareciera una regla:

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

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

```
make clean
```

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

```
.PHONY : clean
```

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

```
make clean
```

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

Macros en ficheros makefile

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

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

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

NombreMacro = texto a expandir

donde:

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

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

OBJ = ~/MP/obj

Si esta línea se incluye en el fichero makefile, cuando `make` encuentra la construcción `$(OBJ)` en él, sustituye dicha construcción por `~/MP/obj`. Cada macro debe estar en una línea separada en un fichero makefile y se 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.

Macros predefinidas

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

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

Ejercicio

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

Sustituciones en macros

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

`$(NombreMacro:TextoOriginal = TextoNuevo)`

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

1. **No** se permiten espacios en blanco antes o 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 `FUENTE` definida como:

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

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

Macros como parámetros en la llamada a make

Además de las opciones básicas indicadas en la página 33, 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_ppa1_2` para que el resultado (el ejecutable `ppa1_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_ppa1_2` para evitar el error.

Ejercicio

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

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` 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`.

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 patrón implícita 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.

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*.

Directivas condicionales en ficheros makefile

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

Para más información ver

<http://www.gnu.org/software/make/manual/make.html#Conditionals>

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

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

La sintaxis para un condicional con parte else sería:

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

Las directivas condicionales para ficheros makefile son las siguientes:

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

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

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

Devuelve verdad si los dos argumentos expandidos son iguales.

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

Devuelve verdad si los dos argumentos expandidos son distintos

else

Actúa como un **else** de C++.

endif

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

Sesión 3

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*.

La modularización del software en C++

Introducción

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

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

Ventajas de la modularización del software

1. Los módulos contendrán de forma natural conjuntos de funciones relacionadas desde un punto de vista lógico.
2. Resulta fácil aplicar un enfoque orientado a objetos. Cada objeto (tipo de dato abstracto) se agrupa en un módulo junto con las operaciones del tipo definido.

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

Cómo dividir un programa en varios ficheros

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

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

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

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

Organización de los ficheros fuente

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

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

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

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

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

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

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

Ejercicio

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

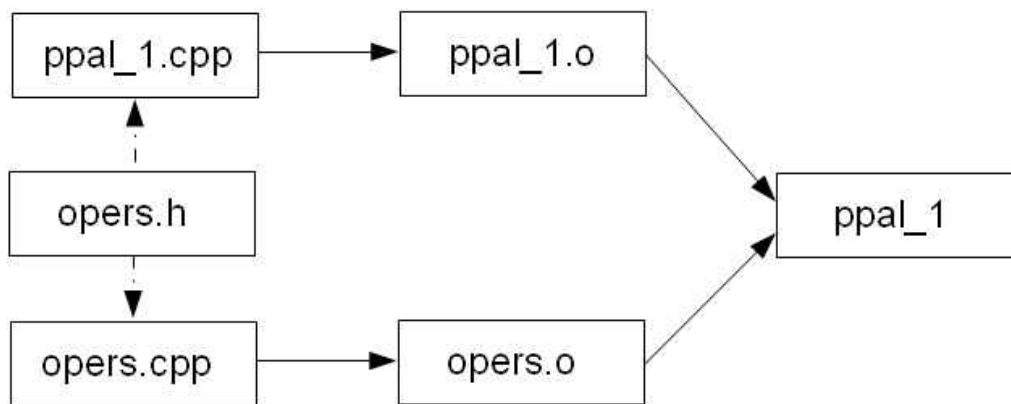


Figura 10: Diagrama de dependencias para construir ppa1_1

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 10. Con detalle:

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

Realizar las siguientes tareas:

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

Ejercicio

Escribir un fichero llamado `makefile_ppal_1` para generar el ejecutable `ppal_1`, siguiendo el diagrama de dependencia mostrado en la figura 10.

Usad macros para especificar los directorios de trabajo.

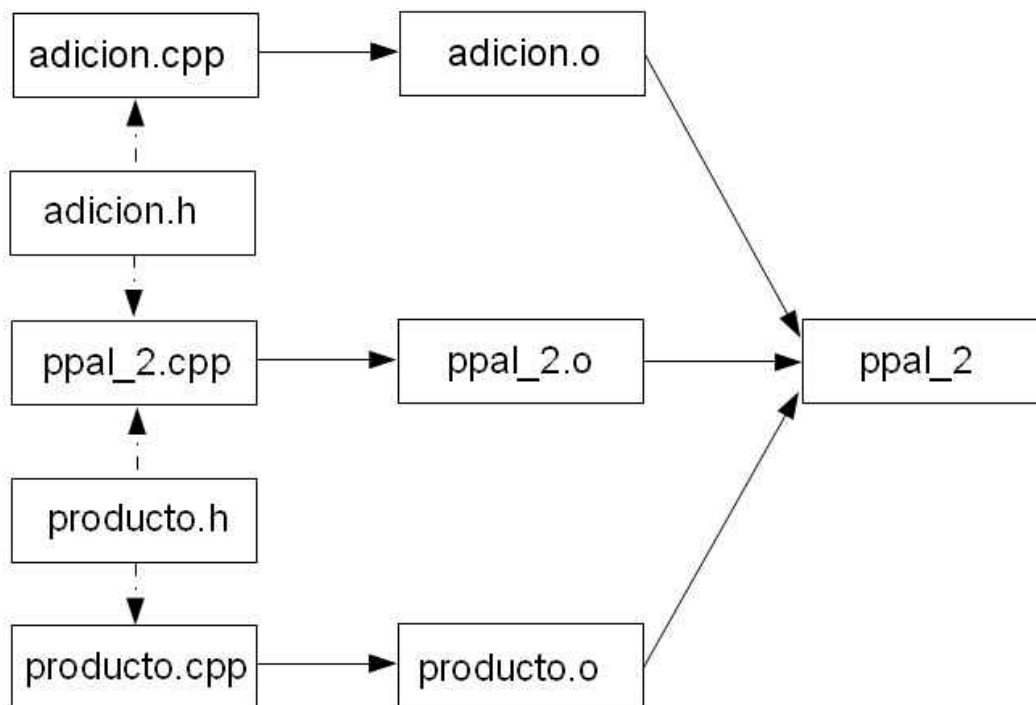


Figura 11: Diagrama de dependencias para construir ppa1_2

Ejercicio

Continuaremos distribuyendo el código fuente en distintos ficheros, siguiendo el esquema indicado en la figura 11.

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 `ppa1_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 11.

Usad macros para especificar los directorios de trabajo.

Bibliotecas

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

1. El tamaño total del código fuente de los programas se incrementa innecesariamente y es redundante.
2. Si se hace necesaria una actualización del código de una función es preciso modificar **TO-DOS** 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).

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.

Estructura de una biblioteca

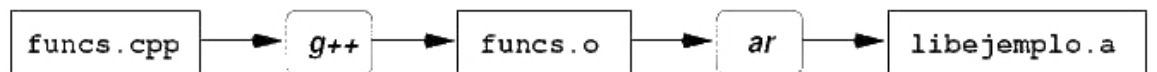
Una biblioteca se estructura internamente como un **conjunto de módulos objeto**. Cada uno de estos módulos será el resultado de la compilación de un fichero de código fuente que puede contener una o varias funciones.

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 12.

Figura 12: Construcción de una biblioteca a partir de un único módulo objeto (caso 1)



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

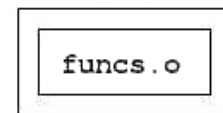
```
int funcion_1 (int a, int b)
{
    .....
}

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

.....

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

libejemplo.a



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 13 y 14.

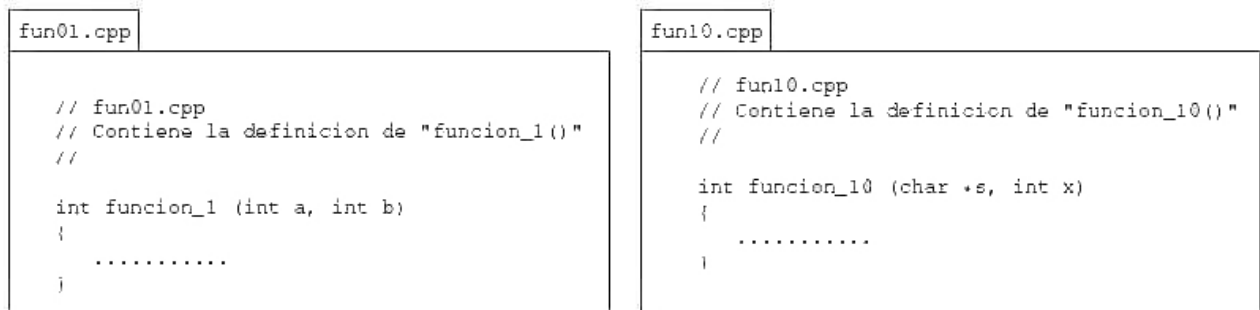


Figura 13: Varios módulos fuente (caso 2)

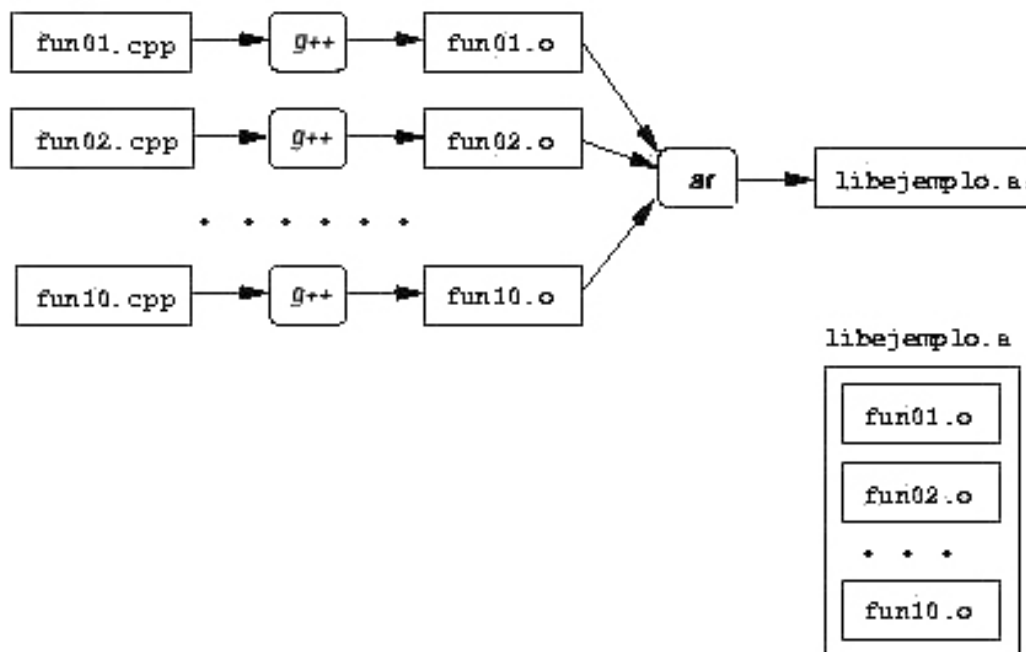


Figura 14: Construcción de una biblioteca a partir de varios módulos objeto (caso 2)

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

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

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

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

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

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

Para la elección de la estructura óptima de la biblioteca hay que recordar que la parte de la biblioteca que se enlaza al código objeto de la función `main()` es el módulo objeto **completo** en el que se encuentra la función de biblioteca usada. En la figura 15 mostramos cómo se construye un ejecutable a partir de un módulo objeto (`ppal.o`) que contiene la función `main()` y una biblioteca (`libejemplo.a`).

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

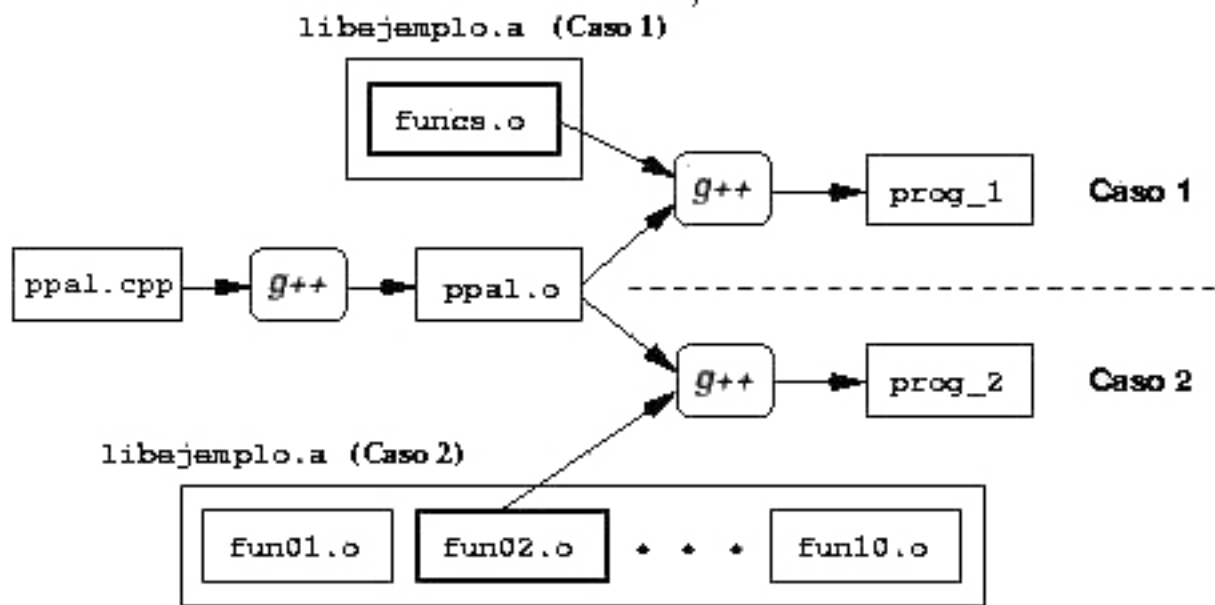


Figura 15: Construcción de un ejecutable que usa una función de biblioteca. Caso 1: biblioteca formada por un módulo objeto. Caso 2: biblioteca formada por 10 módulos objeto

En cualquier caso, como **usuario** de la biblioteca se actúa de la misma manera en ambas situaciones: enlaza el módulo objeto que 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 pequeños².

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

El programa ar

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

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

donde:

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

Ejercicio

Repetir los siguientes ejemplos:

1. Crear la biblioteca **libprueba.a** a partir del módulo objeto **opers.o**.
ar -rvs lib/libprueba.a obj/opers.o
2. Mostrar los módulos que la componen.
ar -tv lib/libprueba.a

Ejercicio

1. Añadir los módulos `adicion.o` y `producto.o` a la biblioteca `libprueba.a`.
`ar -rvs lib/libprueba.a obj/adicion.o obj/producto.o`
2. Mostrar los módulos que la componen.
`ar -tv lib/libprueba.a`

Ejercicio

1. Extraer el módulo `adicion.o` de la biblioteca `libprueba.a`.
`ar -xvs lib/libprueba.a adicion.o`
2. Mostrar los módulos que la componen.
`ar -tv lib/libprueba.a`
3. Mostrar el directorio actual.

Ejercicio

1. Borrar el módulo `producto.o` de la biblioteca `libprueba.a`.
`ar -dvs lib/libprueba.a producto.o`
2. Mostrar los módulos que la componen.
`ar -tv lib/libprueba.a`
3. Mostrar el directorio actual.

g++, make *y* ar **trabajando conjuntamente**

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

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

Creación de la biblioteca

Emplearemos una regla para la construcción de la biblioteca.

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

Para los dos casos estudiados en la página 59, escribiríamos:

1. Caso 1.

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

2. Caso 2.

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

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

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

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

Enlazar con una biblioteca

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

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

Para los dos casos estudiados en la página 59, 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
```

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.

Ejercicio

Utilizar como base `makefile_ppal_1` y construir el fichero `makefile` llamado `makefile_1lib_1mod` que implementa el diagrama de la figura 16

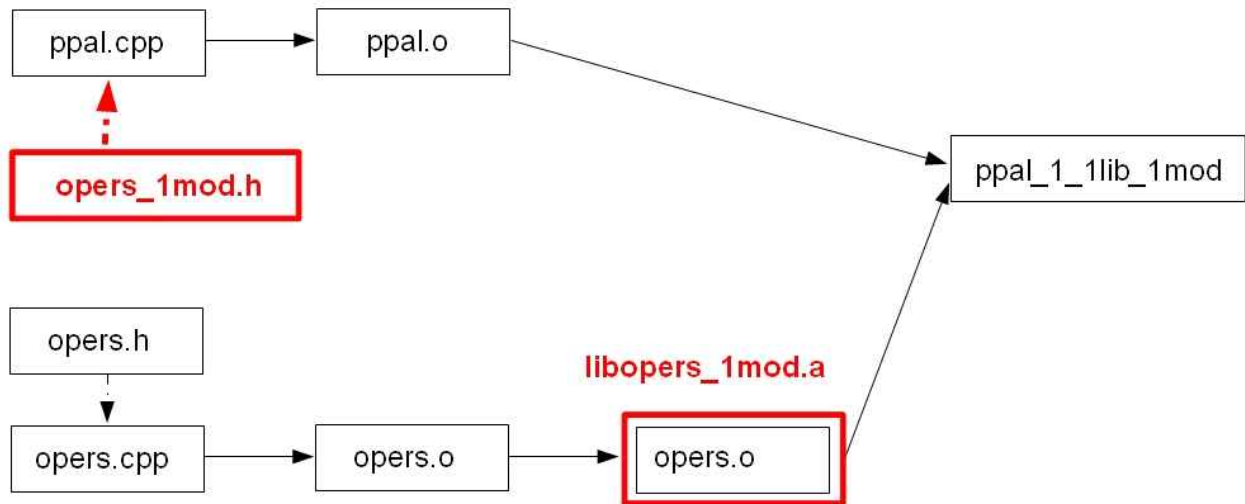


Figura 16: Construcción de un ejecutable que usa funciones de biblioteca (1). Una biblioteca formada por un módulo objeto que implementa cuatro funciones.

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 17

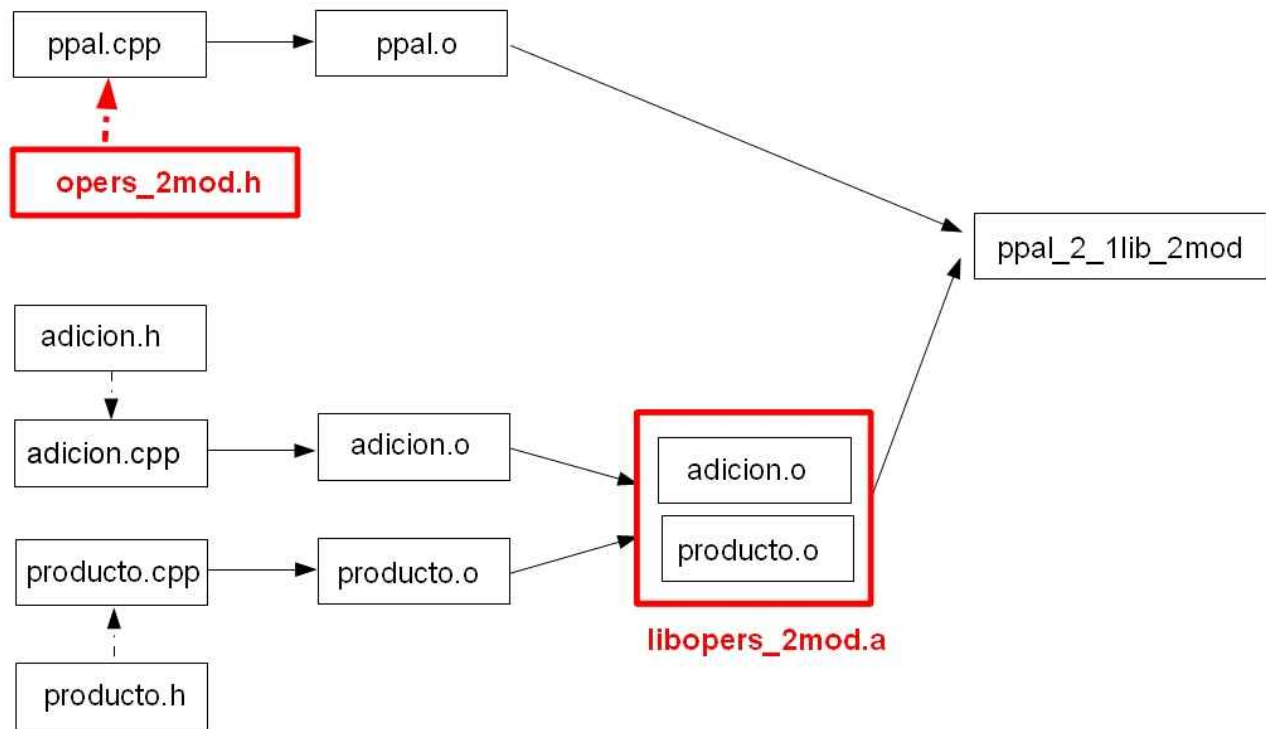


Figura 17: Construcción de un ejecutable que usa funciones de biblioteca (2). Una biblioteca formada por dos módulos objeto que implementan dos funciones cada uno de ellos.

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 18. 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?

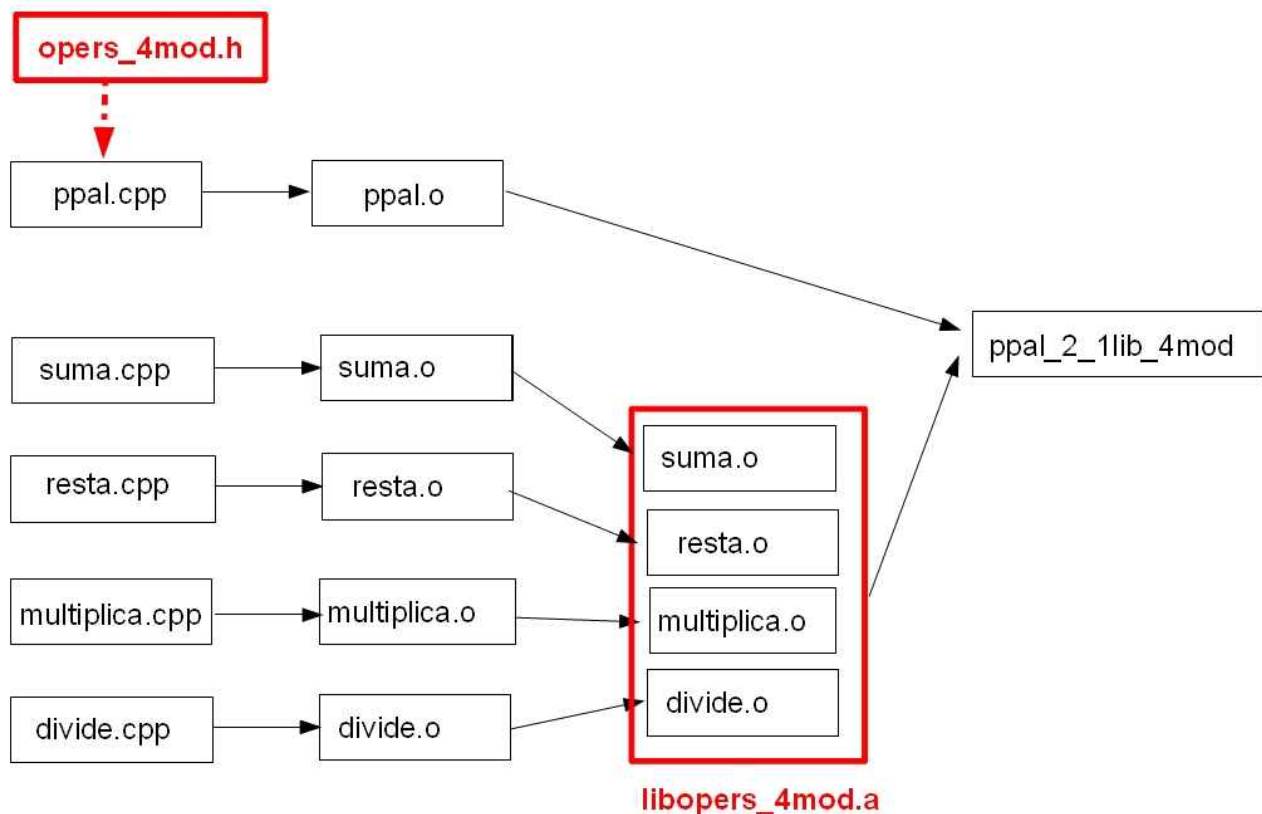


Figura 18: Construcción de un ejecutable que usa funciones de biblioteca (3). Una biblioteca formada por cuatro módulos objeto, y cada uno define una única función.

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 19

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.

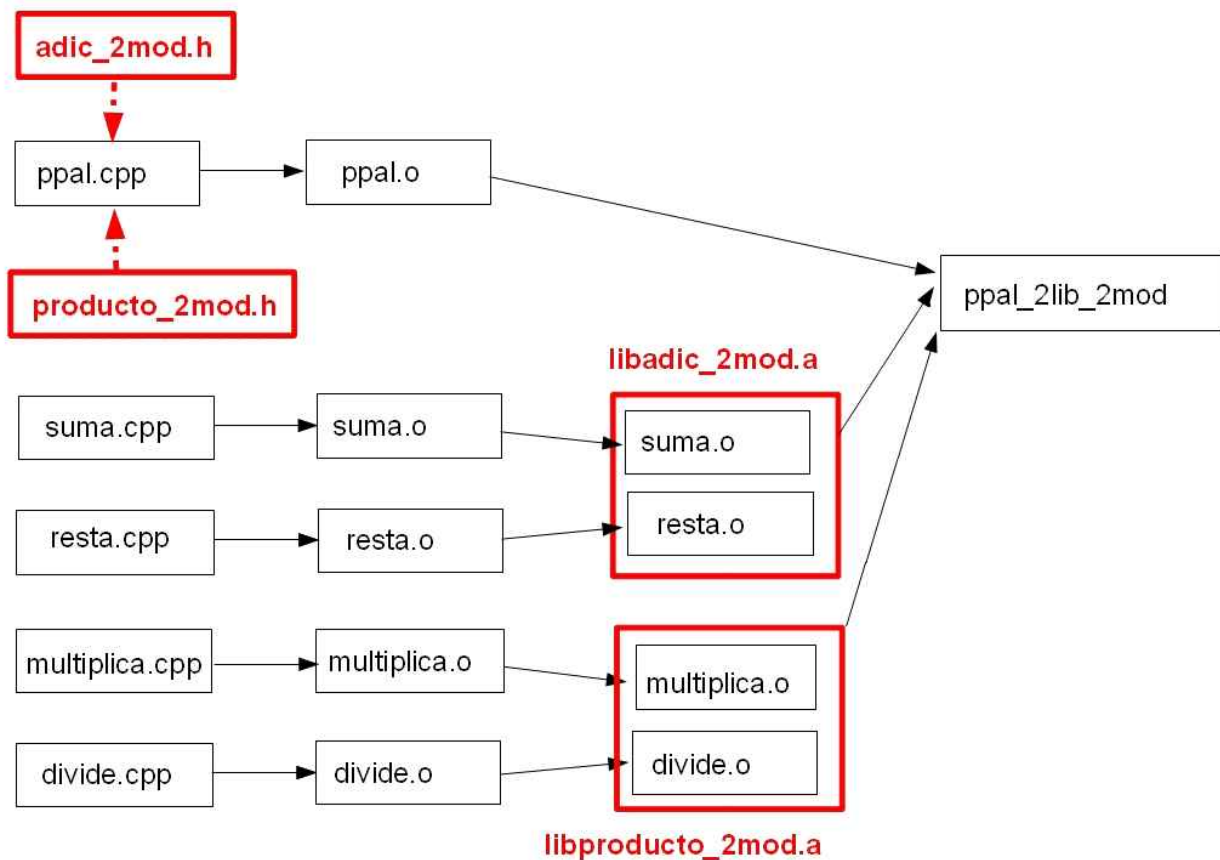


Figura 19: Construcción de un ejecutable que usa funciones de biblioteca (4). Dos bibliotecas formadas cada una por dos módulos objeto, y cada uno define dos funciones.

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?

Sesión 4

Punteros

► Objetivos

Los ejercicios a resolver en esta práctica tienen como objetivo practicar con las cadenas de caracteres *clásicas* (tipo C), modularizando las soluciones en diferentes ficheros. Concretamente:

1. Entender la manera en la que se declaran y definen cadenas clásicas y la necesidad de reservar suficiente espacio.
2. Entender la importancia de gestionar correctamente el carácter `'\0'`, que debe estar presente siempre en toda cadena clásica.
3. Escribir funciones que reciben/devuelven punteros. En particular, punteros que contienen la dirección de memoria de algún elemento de un vector.
4. Modularizar en diferentes ficheros la solución a un problema.
5. Gestionar el proyecto empleando un fichero *makefile*.

► Actividades a realizar en casa

Actividad: Resolución de problemas.

Se trabajará sobre la **Relación de Problemas I: Punteros** (página **RP-I.1**). Concretamente, se trata de resolver los ejercicios **7, 8, 9, 10, 11, 12, 13, 14 y 16**.

Todos los ejercicios indicados son **obligatorios**.

- Los ejercicios **7** y **8** se resuelven con dos programas independientes y autocontenidos:
 - 7** (Posición del primer espacio) `I_PosicionPrimerBlanco.cpp`.
 - 8** (Saltar la primera palabra) `I_SaltaPrimeraPalabra.cpp`.
- Los problemas **9** (longitud), **10** (palíndromo), **11** (comparación), **12** (copia), **13** (concatenación), **14** (subcadena) y **16** (inversión) piden escribir funciones sobre cadenas clásicas. Para probar esas funciones podríamos:
 1. escribir un programa independiente (con su correspondiente función `main`) para probar cada una de las funciones, o

2. probar todas las funciones en un único programa.

Esta será la opción que habrá que implementar.

Modularizar la solución escribiendo:

- el fichero `MiCadenaClasica.h`, que contenga las **declaraciones** (los prototipos) de las funciones,
- el fichero `MiCadenaClasica.cpp`, que contenga las **definiciones** (la implementación),
- el fichero `I_DemoCadenasClasicas.cpp`, que contenga únicamente la función `main`. En esta función se escribirá el código que permita probar todas las funciones.

Para poder generar correctamente los tres ejecutables pedidos en esta práctica (a saber, `I_PosicionPrimerBlanco`, `I_SaltaPrimeraPalabra` y `I_DemoCadenasClasicas`) será preciso escribir un fichero `makefile`, al que llamarán `sesion04.mak`.

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

Mientras el profesor va llamando a algunos alumnos para la corrección de los ejercicios de esta sesión, el resto debe realizar la siguiente tarea:

1. Añadir al módulo `MiCadenaClasica` una función que devuelva un puntero al primer espacio en blanco de una cadena (de manera similar a como se especifica en el problema 7, aunque ahora la “posición” viene dada por un dato `char *`).
2. Modificar la función `main` para poder probar la nueva función y reconstruir el proyecto usando el fichero `makefile` `sesion04.mak`.
3. Construir una biblioteca que contenga el código objeto de las funciones escritas. Reescribir convenientemente el fichero `makefile` para que `make` construya/actualice la biblioteca y para que el ejecutable `I_DemoCadenasClasicas.cpp` se construya a partir de la biblioteca. Llamar al nuevo fichero `makefile` `sesion04_lib.mak`.

► **Recomendaciones**

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

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion04.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion04.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 5

Punteros y funciones

► **Objetivos**

Los ejercicios a resolver en esta práctica tienen como objetivo practicar con el paso de vectores y matrices a funciones, y su gestión con punteros.

1. Definir cadenas clásicas y entender la necesidad de reservar suficiente espacio.
2. Escribir funciones que reciben/devuelven punteros.
3. Escribir funciones que procesan arrays y matrices.
4. Practicar la sobrecarga de funciones.
5. Practicar con referencias como parámetros formales.
6. Escribir programas que reciben argumentos desde la línea órdenes.
7. Gestionar el proyecto empleando un fichero `makefile`.

► **Actividades a realizar en casa**

Actividad: Resolución de problemas.

Se trabajará sobre la **Relación de Problemas I: Punteros** (página **RP-I.1**). Concretamente, se trata de resolver los ejercicios **21**, **22**, **23**, y **24**.

- **Obligatorios:**

- 19** (Leer entero) `I_LeeEntero.cpp`.
- 20** (Máximo y mínimo de un vector) `I_MaxMin_Array.cpp`.
- 21** (Posición del mayor) `I_PosMayor.cpp`.
- 23** (Mezcla de *arrays*) `I_MezclaArrays.cpp`.
- 24** (Sucursales) `I_Sucursales_Matriz_Clasica.cpp`.

- Opcionales:

22 (Ordenación por índices) `I_OrdenConPunteros.cpp`.

Para poder generar correctamente los ejecutables pedidos en esta práctica será preciso escribir un fichero makefile, al que llamarán `sesion05.mak`.

► Actividades a realizar en las aulas de ordenadores

El profesor irá llamando a los alumnos para la corrección de los ejercicios de esta sesión.

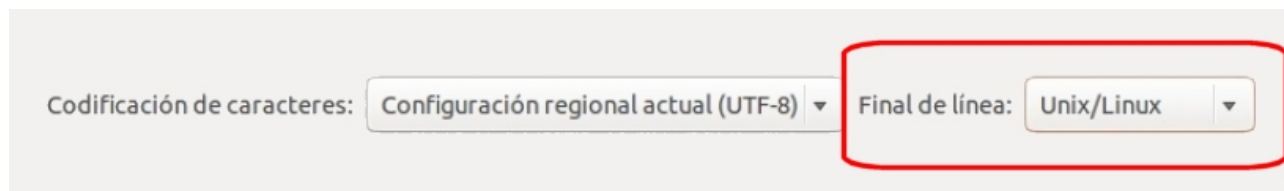
► Recomendaciones

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

Para la prueba y ejecución del ejercicio 24 recomendamos que la lectura se realice mediante la redirección de la entrada, tomando los datos de un fichero de texto.

En la página de la asignatura en PRADO disponen de ficheros de texto para ese propósito.

Nota: Si usa su propio fichero de datos, tenga mucho cuidado si el fichero se ha creado en Windows: los saltos de línea en ficheros de texto se gestionan de manera diferente en Windows y en Gnu/Linux. Puede leer el fichero creado en Windows y guardarlo convenientemente con `gedit` seleccionando **Archivo | Guardar como** y seleccionando el formato adecuado de final de línea en la parte baja de la ventana.



► Normas detalladas

Deberá entregar únicamente el fichero `sesion05.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion05.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter ~

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 6

Gestión de memoria dinámica (1)

Esta sesión de prácticas será la primera de las tres dedicadas a trabajar sobre la **Relación de Problemas II: Memoria dinámica** (página **RP-II.1**) Los problemas propuestos en esta relación se han organizado en tres bloques, de acuerdo a la manera en que se estudian en las clases de teoría:

1. En el primero (ejercicios **1, 2, 3, 4 y 5**) se trabaja sobre vectores dinámicos.

Este será el trabajo a desarrollar en esta práctica.

2. En el segundo (ejercicios **6, 7, 8, 9, 10 y 11**) se trabaja con matrices dinámicas.
3. En el tercero (ejercicios **12, 13, 14, 15 y 16**) se trabaja con listas enlazadas.

► **Objetivos**

Los problemas propuestos tienen como objetivo practicar con la gestión de la memoria dinámica: reserva, acceso y liberación. Los ejercicios tratan sobre una sencilla estructura de datos: el *vector dinámico*. Concretando, se trata de adquirir destrezas en:

1. Reservar memoria en el heap y liberarla. Escribir funciones que reservan memoria dinámica y devuelven la dirección del bloque reservado en el Heap.
2. Dimensionar y redimensionar estructuras dinámicas (vectores) de acuerdo a las necesidades de almacenamiento que marquen los datos que se están procesando.
3. Seguir practicando con cadenas de caracteres clásicas.
4. Gestionar estructuras de datos con información heterogénea (**struct**) como una herramienta sencilla para encapsular las propiedades básicas de tipos de datos complejos.
5. Compilación separada de programas, diferenciando declaración y definición de tipos y funciones, y usando el tipo de datos modularizado en un módulo que contiene la función **main**.

► **Actividades a realizar en casa**

Actividad: Resolución de problemas.

Ejercicios **obligatorios** de la Relación de Problemas I:

17 (Encontrar el inicio de palabras) `I_EncuentraInicioPalabras.cpp`.

18 (Encontrar palabras) `I_EncuentraPalabras.cpp`.

Ejercicios **obligatorios** de la Relación de Problemas II:

- 1 (Redimensionar vector dinámico) `II_RedimensionaVectorDinamico.cpp`.
- 2 (Modularización - vector dinámico) `II_Demo-VectorDinamico.cpp` para la función `main` y `VectorDinamico.h` junto a `VectorDinamico.cpp` para el tipo y las funciones que gestionan el vector dinámico.
- 3 (Encontrar palabras) `II_EncuentraPalabras_MemDin.cpp`.
- 5 (Vector dinámico de cadenas) `II_VectorDinamicoCadenas.cpp`.

Para la resolución del ejercicio 5 recomendamos que la lectura se realice mediante la redirección de la entrada, tomando los datos de un fichero de texto.

Nota: Si usa su propio fichero de datos, tenga mucho cuidado si el fichero se ha creado en Windows: los saltos de línea en ficheros de texto se gestionan de manera diferente en Windows y en Gnu/Linux. Puede leer el fichero y guardarlo convenientemente con `gedit` (ver detalles en la sesión 5.)

Ejercicio **opcional** (Relación de Problemas II):

- 4 (Descomposición en factores primos) `II_FactoresPrimos_VectorDinamicoPrimos.cpp`.

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

El profesor irá llamando a los alumnos para que expliquen al resto de compañeros los ejercicios de esta sesión.

► **Recomendaciones**

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

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion06.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion06.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 7

Gestión de memoria dinámica (2)

Esta sesión de prácticas es la segunda de las tres dedicadas a trabajar sobre la **Relación de Problemas II: Memoria dinámica** (página **RP-II.1**). Los problemas propuestos en esta relación se han organizado en tres bloques, de acuerdo a la manera en que se estudian en las clases de teoría:

1. En el primero (ejercicios **1**, **2**, **3**, **4** y **5**) se trabaja sobre vectores dinámicos.
2. En el segundo (ejercicios **6**, **7**, **8**, **9**, **10** y **11**) se trabaja con matrices dinámicas.
Este será el trabajo a desarrollar en esta práctica.
3. En el tercero (ejercicios **12**, **13**, **14**, **15** y **16**) se trabaja con listas enlazadas.

► **Objetivos**

1. Reservar memoria en el heap y liberarla. Escribir funciones que reservan memoria dinámica y devuelven la dirección del bloque reservado en el Heap.
2. Gestionar estructuras dinámicas bidimensionales (tipo *matriz*).
3. Gestionar vectores dinámicos como parte de una matriz bidimensional, y practicar con las instrucciones que copian bloques de memoria.
4. Gestionar estructuras de datos con información heterogénea (**struct**) como una herramienta sencilla para encapsular las propiedades básicas de tipos de datos complejos.
5. Compilación separada de programas, diferenciando declaración y definición de tipos y funciones, y usando el tipo de datos modularizado en un módulo que contiene la función **main**.

Uno de los objetivos más importantes de esta práctica es la de modularizar correctamente (a nivel de fichero) los tipos de datos **Matriz2D_1** y **Matriz2D_2** y su relación. Si la modularización se hace bien, los programas que usan estos tipos de datos serán similares, y la resolución del ejercicio que permite la conversión entre ambos tipos será muy sencilla.

Nota: Próximamente abordaremos el problema de modularización usando clases y resolveremos los problemas derivados de la copia superficial que realizan el constructor de copia y el operador de asignación.

► Actividades a realizar en casa

Actividad: Resolución de problemas.

- *Obligatorios:*

- 6 (Matriz dinámica - filas independientes) `II_Demo-Matriz2D_1.cpp`.
- 9 (Asignación de tareas a técnicos) `II_ProblemaAsignacion.cpp`.
- 10 (Viajante de comercio) `II_ProblemaViajanteComercio.cpp`.

- *Opcionales:*

- 7 (Matriz dinámica - datos en una fila) `II_Demo-Matriz2D_2.cpp`.
- 8 (Conversión entre matrices) `II_Demo-Conversiones-Matriz2D.cpp`.
- 11 (Sucursales - Matriz dinámica) `II_Sucursales_MatrizDinamica.cpp`.

En los ejercicios 6 y 7 `II_Demo-Matriz2D_1.cpp` y `II_Demo-Matriz2D_2.cpp` contendrán la función `main`, en la que se hace una demostración del uso de las funciones desarrolladas para cada uno de los tipos de matriz dinámica. Si fuera preciso usar alguna otra función desde `main` que no estuviera asociada a la gestión de la matriz dinámica también se escribiría en este fichero.

Los tipos `Matriz2D_1` y `Matriz2D_2` se modularizarán en los ficheros:

- `Matriz2D_1.h` y `Matriz2D_1.cpp`
- `Matriz2D_2.h` y `Matriz2D_2.cpp`

de manera que los ficheros `.h` contendrán las declaraciones de los tipos de datos y de las funciones necesarias para la gestión de los datos y los ficheros `.cpp` sus definiciones.

En los ejercicios 9 y 10 se usarán las matrices dinámicas `Matriz2D_1` ó `Matriz2D_2`. Modularice la solución para que la función `main` esté en un fichero diferente a la declaración/definición del tipo de las matrices dinámicas.

Sugerimos modularizar la solución del ejercicio 11 dejando la función `main` en el fichero `II_Sucursales_MatrizDinamica.cpp` y en los ficheros:

- `Sucursales.h`: las declaraciones de constantes, tipos de datos y funciones.
- `Sucursales.cpp`: las definiciones de las funciones.

Nota: Esta modularización sugerida se realiza para adquirir práctica más que por la utilidad del módulo. Como norma general no es recomendable este tipo de modularización.

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

El profesor irá llamando a los alumnos para defender los ejercicios entregados en esta sesión.

► **Recomendaciones**

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

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion07.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion07.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 8

Gestión de memoria dinámica (3)

Esta sesión de prácticas es la tercera (y última) de las tres dedicadas a trabajar sobre la **Relación de Problemas II: Memoria dinámica** (página **RP-II.1**). Los problemas propuestos en esta relación se han organizado en tres bloques, de acuerdo a la manera en que se estudian en las clases de teoría:

1. En el primero (ejercicios **1**, **2**, **3**, **4** y **5**) se trabaja sobre vectores dinámicos.
2. En el segundo (ejercicios **6**, **7**, **8**, **9**, **10** y **11**) se trabaja con matrices dinámicas.
3. En el tercero (ejercicios **12**, **13**, **14**, **15** y **16**) se trabaja con listas enlazadas.

Este será el trabajo a desarrollar en esta práctica.

► **Objetivos**

1. Seguir practicando con la reserva y liberación de memoria en el heap.
2. Gestionar estructuras de datos con información compleja.
3. Gestionar estructuras lineales dinámicas autorreferenciadas (tipo *lista*).
4. Gestionar estructuras de datos con información heterogénea (**struct**) como una herramienta sencilla para encapsular las propiedades básicas de tipos de datos complejos.
5. Seguir profundizando en la compilación separada de programas, diferenciando declaración y definición de tipos y funciones, y usando el tipo de datos modularizado en un módulo que contiene la función **main**.

Uno de los objetivos más importantes de esta práctica es la de modularizar correctamente (a nivel de fichero) el tipo de dato **Lista**.

Nota: Próximamente abordaremos el problema de modularización usando clases y resolveremos los problemas derivados de la copia superficial que realizan el constructor de copia y el operador de asignación.

► **Actividades a realizar en casa**

Actividad: Resolución de problemas.

- *Obligatorios:*

- 12 (Leer y mostrar lista. Recorridos sobre listas) `II_BasicosLista.cpp`

- 13 (Ordenar lista) `II_OrdenarLista.cpp`

- 14 (Insertar y eliminar en lista ordenada) `II_GestionarListaOrdenada.cpp`

- 15 (Mezclar dos listas ordenadas) `II_MezclarListasOrdenadas.cpp`

Estos ficheros `.cpp` contendrán la función `main`, en la que se hace una demostración del uso de las funciones desarrolladas para la gestión de listas. Si fuera preciso usar alguna otra función desde `main` que no estuviera asociada a la gestión de los tipos de datos sobre los que se trabaja en esta práctica también se escribiría en el mismo fichero.

El tipo `Lista` se modularizará en los ficheros:

- `Lista.h` (declaraciones de los tipos de datos y de las funciones necesarias -prototipos- para la gestión de los datos)
- `Lista.cpp` (definiciones de las funciones que gestionan los datos).

Por la manera en la que se enuncian los ejercicios (en orden creciente de dificultad) estos ficheros pueden ir escribiéndose incrementalmente, conforme se van resolviendo los ejercicios.

Una aplicación (opcional) extensa: gestión de tareas y recursos.

Deberá resolver el ejercicio 16 (Gestión de tareas y recursos).

El ejercicio debería resolverse modularizando en varios ficheros.

- El fichero que contiene la función `main` se llamará `II_GestionTareas.cpp`
- Los ficheros que contienen las funciones que gestionan la lista de tareas se llamarán `ListaTareas.h` y `ListaTareas.cpp`
- A partir de los dos anteriores construya la biblioteca `libListaTareas.a`

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

El profesor irá llamando a los alumnos para defender los ejercicios entregados en esta sesión.

► **Recomendaciones**

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

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion08.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion08.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 9

Clases (I): El constructor de copia y el destructor

En esta sesión de prácticas se trabajará sobre los problemas propuestos en la **Relación de Problemas III: Clases (I)** (página **RP-III.1**).

► **Objetivos**

1. Diseñar *clases* sencillas para tipos de datos conocidos, cuya representación incluya memoria dinámica: vectores dinámicos, matrices bidimensionales y listas.
2. Practicar los mecanismos de ocultación de información.
3. Implementar diferentes constructores (especialmente el **constructor de copia**).
4. Implementar el **destructor**.
5. Escribir métodos que permitan el acceso a los datos y la gestión de los objetos de la la clase.
6. Evitar duplicar código y modularizar las operaciones de **reserva** y **liberación** de memoria, así como la **copia** de contenido entre objetos con **métodos privados**.
7. Seguir profundizando en la compilación separada de programas.

► **Actividades a realizar en casa**

Actividad: Resolución de problemas.

- *Obligatorios:*

- 1 (Clase *vector dinámico*) III_Demo-VectorDinamico_ConstructorCopia.cpp
- 2 (Clase *matriz dinámica -filas independientes-*) III_Demo-Matriz2D_1_ConstructorCopia.cpp
- 4 (Clase *lista*) III_Demo-Lista_ConstructorCopia.cpp

- *Opcionales:*

- 3 (Clase *matriz dinámica -datos en una fila-*) III_Demo-Matriz2D_2_ConstructorCopia.cpp
- 5 (Clase *Pila*) III_Demo-Pila_ConstructorCopia.cpp
- 6 (Clase *Cola*) III_Demo-Cola_ConstructorCopia.cpp

Los ficheros `III_Demo*.cpp` contendrán la función `main`, en la que se hace una demostración del uso de las clases. Si fuera preciso usar alguna otra función desde `main` que no estuviera directamente relacionada con la gestión de los tipos de datos sobre los que se trabaja en esta práctica (p.e. mostrar el contenido de un objeto) **también** se escribiría en el mismo fichero.

Debe modularizar cada clase en dos ficheros y construir una biblioteca para cada una. Se llamarán:

- Vector dinámico:
 - `VectorDinamico_ConstructorCopia.h y`
 - `VectorDinamico_ConstructorCopia.cpp`
 - Biblioteca: `libVectorDinamico_ConstructorCopia.a`
- Matriz dinámica (1):
 - `Matriz2D_1_ConstructorCopia.h y`
 - `Matriz2D_1_ConstructorCopia.cpp`
 - Biblioteca: `libMatriz2D_1_ConstructorCopia.a`
- Matriz dinámica (2):
 - `Matriz2D_2_ConstructorCopia.h y`
 - `Matriz2D_2_ConstructorCopia.cpp`
 - Biblioteca: `libMatriz2D_2_ConstructorCopia.a`
- Lista:
 - `Lista_ConstructorCopia.h y`
 - `Lista_ConstructorCopia.cpp`
 - Biblioteca: `libLista_ConstructorCopia.a`
- Pila:
 - `Pila_ConstructorCopia.h y`
 - `Pila_ConstructorCopia.cpp`
 - Biblioteca: `libPila_ConstructorCopia.a`
- Cola:
 - `Cola_ConstructorCopia.h y`
 - `Cola_ConstructorCopia.cpp`
 - Biblioteca: `libCola_ConstructorCopia.a`

El nombre escogido para las clases pretende poner de manifiesto que **no es una versión definitiva de la la clase**. Falta, fundamentalmente, implementar la sobrecarga del operador de asignación y de los *operadores* de acceso. También falta implementar la sobrecarga de muchos operadores (aritméticos, lógicos, de inserción/extracción en/de flujo y métodos para leer/escribir de/en ficheros, etc.) que se implementarán en próximas sesiones de prácticas. Entonces, podremos asignar un nombre más simple y adecuado a las clases.

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

El profesor irá llamando a los alumnos para que expliquen al resto de compañeros los ejercicios de esta sesión.

► **Recomendaciones**

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

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion09.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion09.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 10

Clases (II): Sobrecarga de operadores

En esta sesión de prácticas se trabajará sobre los problemas propuestos en la **Relación de Problemas IV: Clases (II)** (página **RP-IV.1**).

► **Objetivos**

1. Ampliar la funcionalidad de las clases sobre la que empezamos a trabajar en la sesión de prácticas 8: vectores dinámicos, matrices bidimensionales y listas.
2. Implementar el **operador de asignación**, y alguna sobrecarga de éste.
3. Evitar duplicar código y modularizar las operaciones de reserva y liberación de memoria, así como la **copia** del contenido de objetos con métodos **privados**.
4. Implementar diferentes **operadores de acceso**, dada su posición, para los datos de las clases.
5. Implementar **operadores unarios**.
6. Implementar **operadores binarios** mediante métodos y funciones **friend**.
7. Comprender la diferencia entre un método y una función **friend**, y practicar con ambas soluciones para la sobrecarga de operadores.
8. Implementar **operadores aritméticos**.
9. Implementar **operadores combinados y relacionales**, y diferentes sobrecargas de éstos.
10. Evitar duplicar código y escribir el código de algunos operadores basándose en el código escrito para otros operadores.
11. Seguir profundizando en la compilación separada de programas.

► Actividades a realizar en casa

Actividad: Resolución de problemas.

- **Obligatorios:**

- 2 (Clase *matriz dinámica -filas independientes-* Apartados a, b, c, d, e, f y g)
IV_Demo-Matriz2D_1_Todo.cpp
- 4 (Clase *lista* - Apartados a, b, c, d, e, f y g) IV_Demo-Lista_Todo.cpp
- 9 (Clase *polilínea* - Apartados 1 y 2) IV_Demo-PoliLinea.cpp
- 11 (Red de ciudades) IV_Demo-RedCiudades.cpp

- **Opcionales:**

- 1 (Clase *vector dinámico* - Apartados a, b, c, d y e)
IV_Demo-VectorDinamico_Todo.cpp
- 3 (Clase *matriz dinámica -datos en una fila-* Apartados a, b, c, d, e, f y g)
IV_Demo-Matriz2D_2_Todo.cpp
- 5 (Clase *pila* - Apartados a, b, c y d) IV_Demo-Pila_Todo.cpp
- 6 (Clase *cola* - Apartados a, b, c y d) IV_Demo-Cola_Todo.cpp
- 7 (Clase *Conjunto* - Apartados a-ñ) IV_Demo-Conjunto_Todo.cpp

Estos ficheros .cpp contendrán la función `main`, en la que se hace una demostración del uso de las clases. Si fuera preciso usar alguna otra función desde `main` que no estuviera directamente relacionada con la gestión de los tipos de datos sobre los que se trabaja en esta práctica también se escribiría en el mismo fichero.

Debe modularizar cada clase en dos ficheros y construir una biblioteca para cada una. Las bibliotecas se llamarán como se detalla en el cuadro 2.

Nota: No se trata de las versiones *definitivas* de las clases. Falta implementar la sobrecarga de los operadores de inserción/extracción en/de flujo y métodos para leer/escribir de/en ficheros que se implementarán en próximas sesiones de prácticas.

En particular, para resolver el ejercicio 11 Escriba un constructor *ad-hoc* para inicializar un objeto de la clase con los valores que estime oportuno. Hasta que no se imparta el tema 6 no se podrá escribir un constructor que inicialice un objeto de la clase `RedCiudades` con los datos leídos de un fichero, ni métodos que lean desde un fichero o escriban en él, ni sobrecargar los operadores << ó >>.

| Clase | Ficheros | Biblioteca |
|---------------------|--|---------------------|
| Vector dinámico | VectorDinamico.h VectorDinamico.cpp | libVectorDinamico.a |
| Matriz dinámica (1) | Matriz2D_1.h Matriz2D_1.cpp | libMatriz2D_1.a |
| Matriz dinámica (2) | Matriz2D_2.h Matriz2D_2.cpp | libMatriz2D_2.a |
| Lista | Lista.h Lista.cpp | libLista.a |
| Pila | Pila.h Pila.cpp | libPila.a |
| Cola | Cola.h Cola.cpp | libCola.a |
| Conjunto | Conjunto.h Conjunto.cpp | libConjunto.a |
| Polilínea | PoliLinea.h PoliLinea.cpp | libPoliLinea.a |
| Red de ciudades | RedCiudades.h RedCiudades.cpp | libRedCiudades.a |

Cuadro 2: Nomenclatura de los ficheros

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

El profesor irá llamando a los alumnos para que expliquen al resto de compañeros los ejercicios de esta sesión.

► **Recomendaciones**

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

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion10.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion10.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 11

Gestión de E/S

En esta sesión de prácticas se trabajará sobre los problemas propuestos en la **Relación de Problemas V: Gestión de E/S. Ficheros (I)** (página **RP-V.1**).

► **Objetivos**

1. Comprender cómo se realiza la E/S en C++ mediante flujos.
2. Practicar con operadores y métodos sobre flujos de E/S.
3. Usar la redirección de E/S para leer y/o escribir de/en ficheros de texto.
4. Construir programas que reciben argumentos desde la línea de órdenes.
5. Usar algunas de las clases desarrolladas en anteriores prácticas.
6. Practicar con las funciones de control del estado del flujo, en particular con la función `eof()` y saber usarla correctamente.

► **Actividades a realizar en casa**

Actividad: Resolución de problemas.

- **Obligatorios:**

- 1 (Copiar todo lo que se lee). `CopiaTodo.cpp`
- 2 (Copiar lo que se lee, exceptuando las vocales). `CopiaExceptoVocales.cpp`
- 3 (Contar caracteres leídos). `CuentaCaracteres.cpp`
- 4 (Contar líneas no vacías). `CuentaLineasNoVacias.cpp`
Si usa la función `peek`: `CuentaLineasNoVacias-peek.cpp`
- 5 (Mostrar líneas no vacías). `MuestraLineasNoVacias.cpp`
- 6 (Eliminar separadores en cada línea). `ComprimeLineas.cpp`
- 7 (Copiar líneas no marcadas). `CopiaLineasSinAlmohadilla.cpp`
- 8 (Copiar enteros). `CopiaEnteros.cpp`
- 10 (Copiar enteros separados por una marca). `CopiaEnterosSeparados.cpp`
- 11 (Sumar enteros). `SumaEnteros.cpp`
- 12 (Contar el numero de apariciones). `CuentaCaracteresConcretos.cpp`
- 13 (Contar palabras de una longitud dada). `CuentaPalabrasLongitudConcreta.cpp`

- *Opcionales:*

9 (Copiar enteros en orden inverso). `CopiaEnteros-OrdenInverso.cpp`

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

El profesor irá llamando a los alumnos para que expliquen al resto de compañeros los ejercicios de esta sesión.

► **Recomendaciones**

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

Recomendamos el uso de la redirección de entrada, salida y encauzamiento para facilitar las pruebas y practicar de manera intensiva con los flujos de datos.

En el caso de realizar el ejercicio 9 use la clase `Pila`.

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion12.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion12.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 12

Flujos asociados a string. Ficheros de texto

► **Objetivos**

1. Comprender el concepto de flujo asociado a un `string` y de qué manera pueden emplearse para simplificar ciertas tareas que podrían resultar complejas usando otras herramientas.
2. Comprender cómo se realiza la E/S en C++ mediante flujos asociados a ficheros de datos que contienen **texto**.
3. Practicar con las funciones de control del estado del flujo, en particular con la función `eof()` y saber usarla correctamente.
4. Saber discriminar el nivel de procesamiento adecuado: carácter, palabra o línea.
5. Seguir practicando con operadores y métodos sobre flujos de E/S.
6. Seguir practicando con programas que reciben argumentos desde la línea de órdenes y gestionarlos correctamente.
7. Conocer cómo pueden combinarse datos almacenados en diferentes ficheros.

► **Actividades a realizar en casa**

Actividad: Resolución de problemas.

En esta sesión de prácticas se trabajará sobre los problemas propuestos en la **Relación de Problemas VI: Ficheros (I)** (página [RP-VI.1](#)).

- **Obligatorios:**

- 1 (Numerar líneas). `NumerarLineas.cpp`
- 3 (Mezclar ficheros ordenados). `MezclaFichero.cpp`
- 6 (Ampliación de la clase `Lista`). `VI_Demo-Lista-ES.cpp`
- 7 (Ampliación de la clase `Matriz2D-1`). `VI_Demo-Matriz-ES.cpp`
- 8 (Ampliación de la clase `Matriz2D-1` - versión alternativa).
`Matriz2D_1_FicheroSinCabecera_sstream.cpp`
- 9 (Ampliación de la clase `RedCiudades`). `VI_Demo-RedCiudades.cpp`
- 17 (Imágenes PGM). `InfoPGM.cpp`

- *Opcionales:*

- 2 (Diferencias entre ficheros). `Diferencias.cpp`
- 4 (Calcular sumas de grupos de datos). `SumasPorGrupos.cpp`
- 5 (Particionar un fichero por número de líneas). `ParteFicheroPorNumLineas.cpp`
- 10 (Mensajes ocultos) `MensajesOcultos.cpp`
- 11 (Mensajes ocultos - alternativo) `MensajesOcultos-2.cpp`
- 12 (Buscar palabra). `Busca.cpp`
- 13 (Mostrar encabezamientos). `Cabecera.cpp`
- 14 (Mostrar últimas líneas). `Final.cpp`
- 15 (Encriptar / desencriptar fichero). `Codifica.cpp`
- 16 (Eliminar comentarios). `EliminaComentarios.cpp`

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

El profesor irá llamando a los alumnos para que expliquen al resto de compañeros los ejercicios de esta sesión.

► **Recomendaciones**

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

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion13.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion13.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Sesión 13

Ficheros binarios

► **Objetivos**

1. Comprender cómo se realiza la E/S en C++ mediante flujos asociados a ficheros que gestionan los datos a bajo nivel (sucesión de bytes).
2. Leer de fichero y escribir en ficheros usando memoria intermedia (*buffers*) gestionados por el programador.
3. Comprender y practicar con las funciones de lectura y escritura sin formato, en flujos binarios.
4. Seguir practicando con programas que reciben argumentos desde la línea de órdenes y cómo gestionarlos correctamente.
5. Seguir practicando con las funciones de control del estado del flujo.
6. Seguir practicando con flujos asociados a `string`.
7. Seguir practicando en la manera en que pueden combinarse datos almacenados en diferentes ficheros y con distintos formatos

► **Actividades a realizar en casa**

Actividad: Resolución de problemas.

En esta sesión de prácticas se trabajará sobre los problemas propuestos en la **Relación de Problemas VII: Ficheros (II)** (página [RP-VII.1](#)).

- **Obligatorios:**

- 1 (Conversión texto \Rightarrow binario -uno a uno-). `CopiaEnteros.cpp`
- 2 (Conversión texto \Rightarrow binario -en bloques-). `CopiaEnteros-Bloques512.cpp`
- 5 (Particionar un fichero por número de bytes). `ParteFicheroPorNumBytes.cpp`
- 6 (Reconstruye un fichero previamente particionado). `Reconstruye.cpp`
- 11 (Binarizar imagen). `Binariza.cpp`
- 12 (Aumentar contraste en una imagen). `AumentaContraste.cpp`

- *Opcionales:*

3 Suma enteros de un fichero binario.

1. `SumaEnterosBinario-VectorInt.cpp` (vector de `int`)

2. `SumaEnterosBinario-VectorBytes.cpp` (vector de `bytes`)

4 (Conversión texto \Leftrightarrow binario). `Text2bin.cpp` y `Bin2text.cpp`

8 (Ordena fichero - **implemente sólo un método**). `OrdenaAlumnos.cpp`

9 (Formatos de fichero). `FormatoFichero.cpp`

10 (Verificar fichero PGM). `VerificaPGM.cpp`

13 (Ampliación de la clase `RedCiudades`). `VII_Demo-RedCiudades.cpp`

14 (Puntos, colección de puntos y circunferencia) `VII_Demo-ColeccionPuntos.cpp`

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

El profesor irá llamando a los alumnos para que expliquen al resto de compañeros los ejercicios de esta sesión.

► **Recomendaciones**

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

► **Normas detalladas**

Deberá entregar únicamente el fichero `sesion13.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `sesion13.mak`.

También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.

No incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter `~`

No se admitirán soluciones en las que haya una carpeta diferenciada por ejercicio (con su correspondiente estructura de subcarpetas `src`, `include`, ...).

Proyecto: Empresa de distribución

En el ejercicio 7 se pide escribir dos programas que calculan nuevos datos usando otros almacenados en diferentes ficheros. Es un caso típico que encontrará en muchas aplicaciones y en futuros diseños de bases de datos: dos ficheros independientes y otro que los relaciona. Por ejemplo:

- *vendedores* y *artículos*, unidos por las *ventas*,
- *corredores* y *carreras*, unidos por las *participaciones*,
- *usuarios* y *libros*, unidos por los *préstamos*,
- ...

Para este ejercicio se proporcionan los ficheros de datos:

- **Vendedores** (binario) creado a partir de `EjemploVendedores.txt` (texto),
- **Articulos** (binario) creado a partir de `EjemploArticulos.txt` (texto),
- **Ventas** (binario) creado a partir de `EjemploVentas.txt` (texto),

y los programas que se usaron para generarlos (por si desea añadir o modificar datos): `CreaVendedores.cpp`, `CreaArticulos.cpp` y `CreaVentas.cpp` además de programas usados para listar su contenido: `MuestraVendedores.cpp`, `MuestraArticulos.cpp` y `MuestraVentas.cpp`.

Los programas a desarrollar se llamarán:

1. `VentasPorVendedor.cpp`
2. `VendedoresVIP.cpp` para generar el fichero `VendedoresVIP`.
Para listar el contenido de este fichero, escribir `MuestraVendedoresVIP.cpp`.

Recomendamos un fichero *makefile* (`empresa_distribucion.mak`) para este proyecto.



DECSAI

Departamento de Ciencias de la Computación e I.A.

Universidad de Granada

METODOLOGÍA DE LA PROGRAMACIÓN

Grado en Ingeniería Informática

GRUPO A

Profesor: Francisco José Cortijo Bon

`cb@decsai.ugr.es`



Relaciones de Problemas

Curso 2017/2018

RELACIÓN DE PROBLEMAS I. Punteros

1. Describir la salida de los siguientes programas:

a)

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

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

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

b)

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

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

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

c)

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

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

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

d)

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

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

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

2. Represente gráficamente la disposición en memoria de las variables del programa mostrado en la figura 20, e indique lo que escribe la última sentencia de salida.
3. Declare una variable *v* como un vector de 1000 enteros. Escriba un programa que recorra el vector y modifique todos los enteros *negativos* cambiándolos de signo.

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

Nota: Para inicializar aleatoriamente el vector con valores enteros entre -50 y 50, por ejemplo, puede emplear el siguiente fragmento de código (o la clase que genera números aleatorios - *Fundamentos de Programación*):

```
1 #include <iostream>
2 using namespace std;
3
4 struct Celda {
5     int d;
6     Celda *p1, *p2, *p3;
7 };
8
9 int main (int argc, char *argv[])
10 {
11     Celda a, b, c, d;
12
13     a.d = b.d = c.d = d.d = 0;
14
15     a.p1 = &c;
16     c.p3 = &d;
17     a.p2 = a.p1->p3;
18     d.p1 = &b;
19     a.p3 = c.p3->p1;
20     a.p3->p2 = a.p1;
21     a.p1->p1 = &a;
22     a.p1->p3->p1->p2->p2 = c.p3->p1;
23     c.p1->p3->p1 = &b;
24     (*(c.p3->p1)).p2->p3).p3 = a.p1->p3;
25     d.p2 = b.p2;
26     (*(a.p3->p1)).p2->p2->p3 = (*(a.p3->p2)).p3->p1->p2;
27
28     a.p1->p2->p2->p1->d = 5;
29     d.p1->p3->p1->p2->p1->p1->d = 7;
30     (*(d.p1->p3)).p3->d = 9;
31     c.p1->p2->p3->d = a.p1->p2->d - 2;
32     (*(c.p2->p1)).p2->d = 10;
33
34     cout << "a="<<a.d<<" b="<<b.d<<" c="<<c.d<<" d="<<d.d<<endl;
35 }
```

Figura 20: Código asociado al problema 2

```
#include <cstdlib>
#include <ctime>

...
const int MY_MAX_RAND = 50; // Queremos valores -50<=n<=50
time_t t;

...
srand ((int) time(&t)); // Inicializa el generador con
                        // el reloj del sistema

...
for int (i=0; i<TOPE; i++)
    v[i] = (rand() % ((2*MY_MAX_RAND)+1)) - MY_MAX_RAND;
```

Acerca de `srand()`, `rand()` y `time()`: <http://www.cplusplus.com>

RELACIÓN DE PROBLEMAS I. Punteros

4. Modifique el código del problema 3 para controlar el final del bucle con un puntero a la posición siguiente a la última.
5. Con estas declaraciones:

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

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

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

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

No se puede usar el operador [] , es decir, se debe resolver usando aritmética de punteros.

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

Para ello proponemos un algoritmo que consiste en:

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

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

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

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

Notas:

- La cadena debe recorrerse usando aritmética de punteros y sin usar ningún entero.
 - Usar la función `getline()` para la lectura de la cadena (Cuidado: usar el método público de `istream` sobre `cin`, o sea `cin.getline()`). Ver <http://www.cplusplus.com/reference/istream/istream/getline/>
8. Consideremos una cadena de caracteres “clásica”. Escriba un programa que lea una cadena y la imprima pero saltándose la primera palabra, *evitando escribirla carácter a carácter*. Considere que puede haber ninguna, una o más palabras, y si hay más de una palabra, están separadas por espacios en blanco.
9. Considere una cadena de caracteres “clásica”. Escriba la función `longitud_cadena`, que devuelva un *entero* cuyo valor indica la longitud de la cadena: el número de caracteres desde el inicio hasta el carácter nulo (no incluido). Tome como modelo la función `strlen` de `cstring`: la función accede a la cadena a través de un parámetro formal de tipo `const char *`

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

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

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

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

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

Notas:

- Emplead como criterio para determinar el orden el código ASCII de los caracteres que se están comparando. Tome como modelo la función `strcmp` de `cstring`.
 - No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.
12. Considere dos cadenas de caracteres “clásicas”. Escriba la función `copiar_cadena`, que copiará una cadena de caracteres en otra. El resultado de la copia será el primer argumento de la función. La cadena original (segundo argumento) **no** se modifica.

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

Notas:

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

RELACIÓN DE PROBLEMAS I. Punteros

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

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

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

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

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

Notas:

- La cadena original **no** se modifica.
 - Se supone que hay suficiente memoria en la cadena resultado.
 - Si la longitud es demasiado grande (se sale de la cadena original), se devolverá una cadena de menor tamaño (la que empieza en `p` y llega hasta el final de la cadena).
 - No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.
15. Escriba funciones para eliminar separadores en una cadena:
- `elimina_blanco_iniciales`: los elimina del inicio de la cadena,
 - `elimina_blanco_finales`: los elimina del final de la cadena,
 - `elimina_blanco_extremos`: los elimina del principio y del final,
 - `elimina_blanco_intermedios`: elimina los separadores internos, y
 - `elimina_todos_blanco`: los elimina todos.

Todas las funciones tienen un esquema común:

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

Notas:

- La cadena original **no** se modifica.
 - Se supone que hay suficiente memoria en la cadena resultado.
 - No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.
16. Escriba la función `invertir`. Recibe una cadena de caracteres y devuelve una nueva cadena, resultado de invertir la primera.

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

Notas:

- La cadena original **no** se modifica.
 - No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.
17. Escribir un programa que lea una cadena de caracteres, encuentre y registre el inicio de cada palabra, y finalmente muestre el primer carácter d cada palabra.

Para registrar el inicio de cada palabra, usen un *array* de punteros a carácter (cada puntero contendrá la dirección del primer carácter de una palabra). En la figura 21 mostramos el estado de la memoria para la función `main` justo antes de calcular el inicio de cada palabra. Los vectores mostrados se han declarado:

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

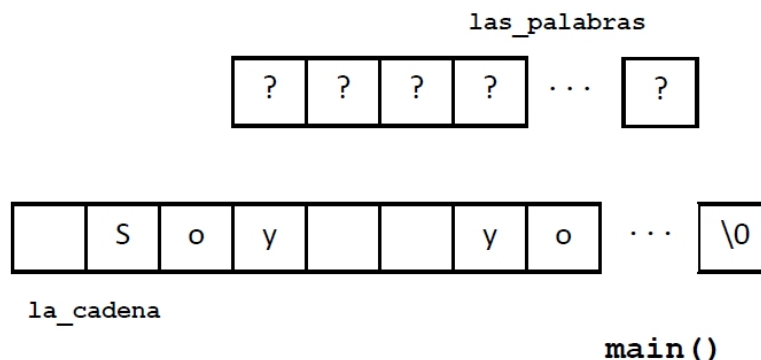


Figura 21: Estado de la pila antes de calcular el inicio de cada palabra.

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

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

donde:

- La función devuelve el número de palabras en `cadena`.
- `cadena` es un puntero a la cadena donde se van a buscar las palabras, y
- `palabras` es un puntero a un *array* de datos `char *` de manera que `palabras[0]` contendrá la dirección de la primera letra de la primera palabra de `cadena`, `palabras[1]` contendrá la dirección de la primera letra de la segunda palabra de `cadena`, ...

En la figura 22 mostramos el estado de la memoria antes de la finalización de la función `encuentra_palabras`. En este ejemplo la función devolverá 2 y se llamó:

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

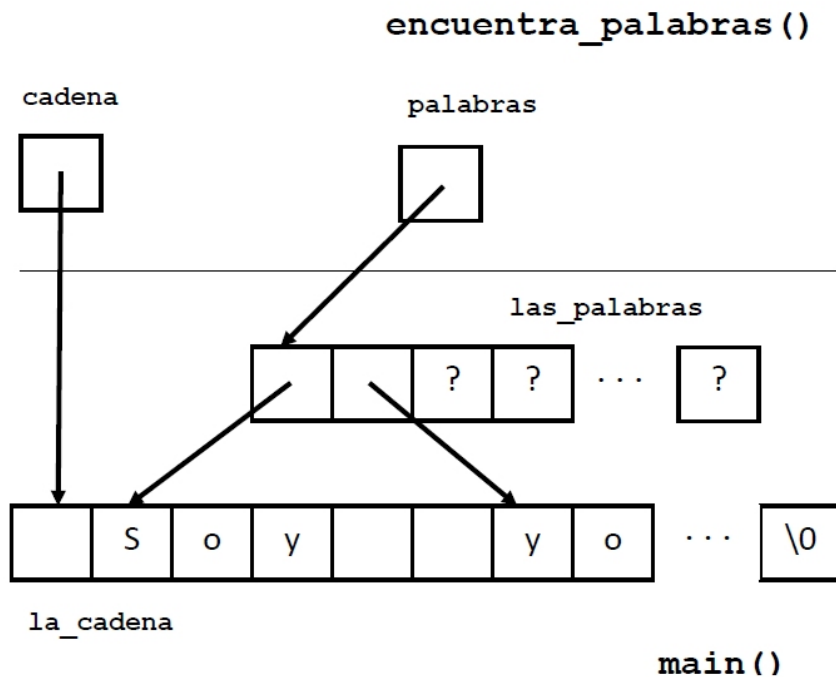


Figura 22: Estado de la pila antes de devolver el control a `main`

Notas:

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

18. Este ejercicio es una ampliación del ejercicio 17. Ahora estamos interesados en saber no solo dónde empieza cada palabra, sino también dónde acaba.

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

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

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

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

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

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

En la figura 23 mostramos el estado de la memoria antes de la finalización de la función `encuentra_palabras`. En este ejemplo la función devolverá 2 y se llamó:

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

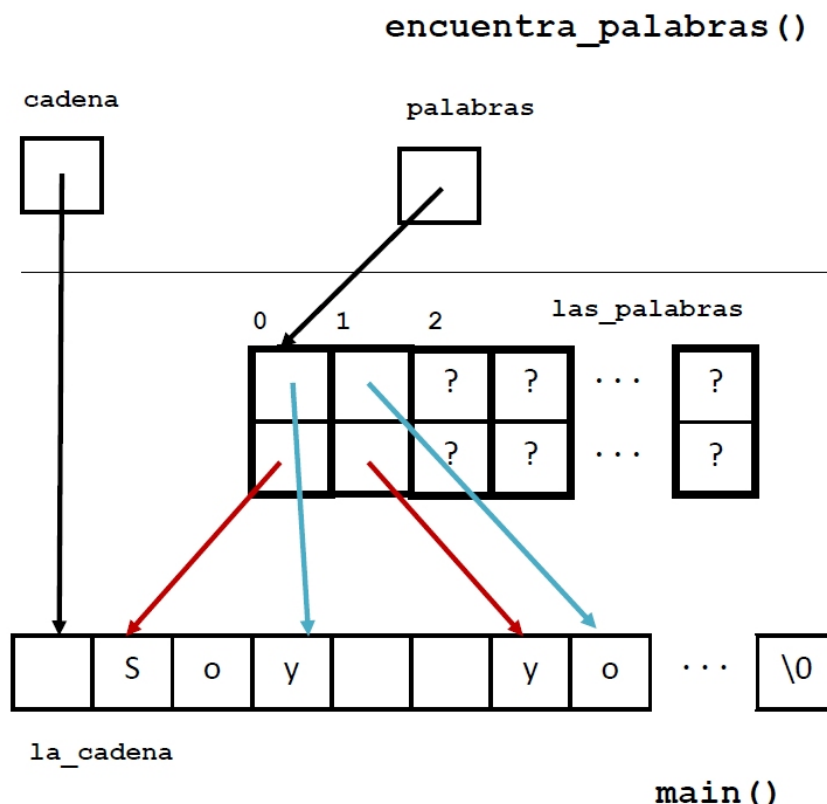


Figura 23: Estado de la pila antes de devolver el control a `main`

RELACIÓN DE PROBLEMAS I. Punteros

- a) Escriba la (nueva) función `encuentra_palabras`.
- b) Escriba la función `muestra_palabras` para mostrar las palabras registradas por la función `encuentra_palabras`. Decida la cabecera de la función.

Notas:

- la zona de memoria referenciada por `cadena` **no** se modifica.
 - Controle la ocupación del array de punteros `las_palabras`. Si se llenara, muestre un mensaje de aviso, y desde entonces ya no se considerarán más palabras, pero el programa no interrumpe su ejecución.
19. Escriba una función para leer y devolver un número entero. La función leerá el número usando una cadena, comprobará que todos los caracteres son dígitos y la convertirá a un entero. Si es correcto, lo devolverá; si no lo es, volverá a pedirlo.

Escriba otra función para leer y devolver un número entero que pertenece a un intervalo. La función puede recibir:

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

Probad las funciones escribiendo un programa.

20. Considere la siguiente declaración:

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

Escriba un programa que rellene aleatoriamente el vector y que calcule el menor y el mayor valor entre dos posiciones dadas.

Modularizar la solución usando, al menos, funciones para:

- a) Rellenar el vector con valores aleatorios entre dos valores comprendidos entre 1 y 500.
- b) Calcular el mínimo y el máximo valor del vector. La función recibirá las posiciones inicial y final entre las que realizar la búsqueda y devolverá (mediante referencias) los valores calculados.

Intente generalizar la solución (rellenar sólo una parte del vector, generar aleatorios entre dos valores dados, etc.).

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

La función tendrá como prototipo:

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

RELACIÓN DE PROBLEMAS I. Punteros

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

Considere la siguiente declaración:

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

Escriba un programa que rellene aleatoriamente el vector (completamente, o una parte) y que calcule el mayor valor entre dos posiciones:

- Si el programa se ejecuta sin argumentos, se rellenará completamente (`TOPE` casillas) y se calculará el mayor valor de todo vector (entre las casillas 0 y `TOPE-1`).
- Si el programa se ejecuta con un argumento (`n`), se rellenarán `n` casillas, y calculará el mayor valor entre ellas (entre las casillas 0 y `n-1`).
- Si el programa se ejecuta con dos argumentos (`n` y `d`), se rellenarán `n` casillas, y calculará el mayor valor entre las casillas 0 y `d`.
- Si el programa se ejecuta con tres argumentos (`n`, `i` y `d`), se rellenarán `n` casillas, y calculará el mayor valor entre las casillas `i` y `d`.
- Si el programa se ejecuta con más de tres argumentos muestra un mensaje de error y no hace nada más.

Nota: Modularice la solución con funciones.

22. Escriba la función

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

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

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

En la figura 24 mostramos el resultado de la función cuando se `.ordena` el vector `vec` entre las casillas 0 y 5.

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

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

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

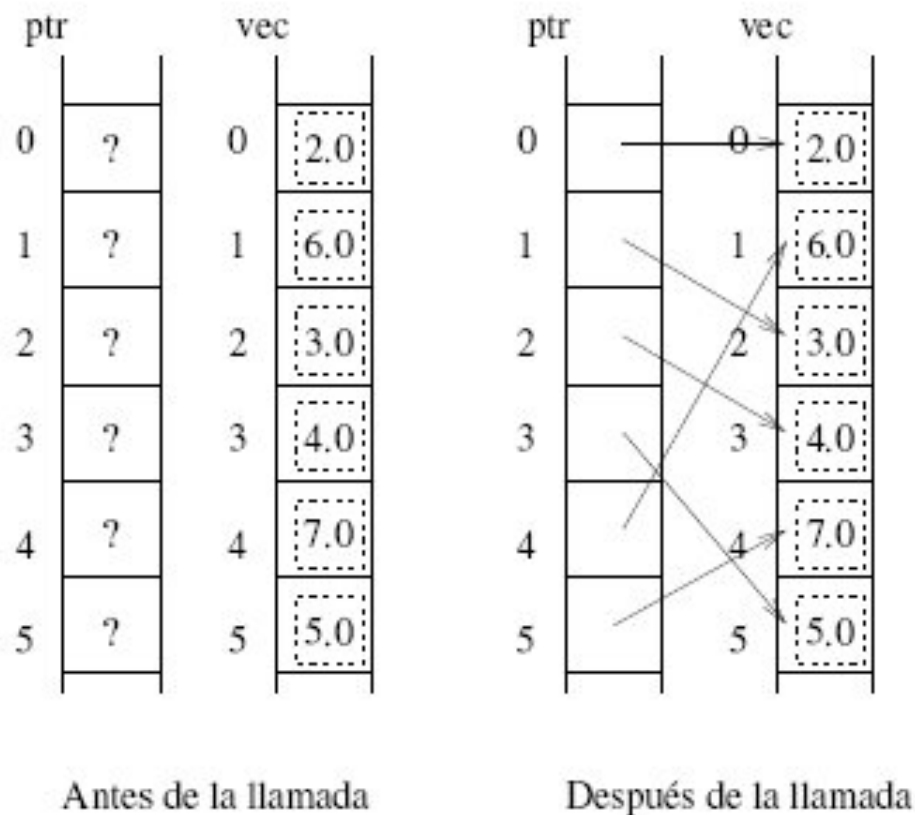


Figura 24: Resultado de ordenar el vector de punteros

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

Nota: Iniciar aleatoriamente el vector `vec`. **Nota:** Modularice la solución con funciones.

23. Escribir un programa que rellene aleatoriamente dos vectores ordenados de datos `int` y los mezcle.

- Si el programa se ejecuta sin argumentos se generarán números aleatorios entre 1 y 100 (ambos incluidos)
- Si se ejecuta con un argumento se supone que éste es el mayor valor aleatorio permitido y se generarán valores aleatorios entre 1 y el valor introducido.
- Finalmente, si se ejecuta con dos argumentos, éstos serán los extremos del intervalo que delimita los valores aleatorios.

RELACIÓN DE PROBLEMAS I. Punteros

Declarar `v1` y `v2`, dos vectores de datos `int` con una capacidad de 100.

Para cada vector pedir (y filtrar adecuadamente) cuántas casillas se van a ocupar y llamar a la función cuyo prototipo es:

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

Rellena con datos generados aleatoriamente `util` casillas del vector `v`. Los datos aleatorios están comprendidos entre `min` y `max` (ambos incluidos). Al finalizar la función el vector `v` está **ordenado**.

Implementar una función para ordenar el vector. Usar el método que desee: burbuja, inserción o selección.

Se mostrará el contenido de cada vector usando la función:

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

Muestra el contenido del vector `v`, que tiene `util` y `util_v2` datos.

Los datos se muestran por líneas, separados por espacios, y en cada línea hay `n` datos (posiblemente la última línea tenga menos).

Antes de presentar los datos y en una línea separada, se muestra `mensaje`.

A continuación se procede a realizar la mezcla ordenada de los dos vectores usando la función:

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

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

- Se supone que `mezcla` tiene suficiente capacidad para albergar la mezcla (al menos `util_v1 + util_v2` casillas).
- El valor devuelto es el número de casillas usadas en `mezcla`. Considere una solución alternativa usando *referencias*.

Finalmente se muestra el resultado. Usar la función `MuestraVector()`

Las funciones no pueden usar el operador `[]`, es decir, se debe usar aritmética de punteros.

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

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

RELACIÓN DE PROBLEMAS I. Punteros

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

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

Después de leer los datos de ventas el programa mostrará:

- a) El número total de operaciones de venta.
- b) La sucursal que más unidades ha vendido y cuántas unidades.
- c) Listado en el que aparecerán: código de sucursal y número total de productos vendidos en la sucursal. Aparecerán únicamente las sucursales que hayan vendido algún producto.
- d) El número de sucursales que hayan vendido algún producto.
- e) Número total de unidades vendidas (calculado como suma de las ventas por sucursales).
- f) Producto más vendido y cuántas unidades.
- g) Listado en el que aparecerán: código de producto y número total de unidades vendidas. Aparecerán únicamente los productos que hayan tenido alguna venta.
- h) Cuántos tipos de producto han sido vendidos.
- i) El número total de unidades vendidas (calculado como suma de las ventas por producto).
- j) Tabla-resumen con toda la información. Por ejemplo:

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

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

Para poder guardar en memoria la información requerida para los cálculos proponemos una matriz bidimensional **ventas** con tantas filas como número (máximo) de sucursales y columnas como número (máximo) de productos. La casilla (s, p) de esta matriz guardará en número

total de unidades vendidas por la sucursal s del producto p (o el número de unidades vendidas del producto p en la sucursal s).

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

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

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

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

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

RELACIÓN DE PROBLEMAS II. Memoria dinámica

Ejercicios sobre vectores dinámicos

1. Deseamos guardar un número indefinido de valores `int`. Utilizaremos un *vector dinámico* que va creciendo conforme necesite espacio para almacenar un nuevo valor.

Escribir una función que realoje y redimensione el vector dinámico, ampliándolo, cuando no haya espacio para almacenar un nuevo valor:

- a) *en una casilla*
- b) *en bloques de tamaño TAM_BLOQUE*
- c) *duplicando su tamaño*

La función que redimensiona el vector tendrá el siguiente prototipo:

```
int * Redimensiona (int * p, TipoRedimension tipo, int & cap);
```

- `p` contiene la dirección de memoria del primer elemento del vector que se va a realojar y redimensionar.
- el segundo argumento, `tipo`, indica el tipo de redimensión que se va a aplicar. Proponemos usar un valor del tipo enumerado:

```
enum class TipoRedimension {DeUnoEnUno, EnBloques, Duplicando};
```

- el tercer argumento, `cap`, es una **referencia** a la variable que indica la capacidad (número de casillas reservadas) del vector dinámico: recuerde que esta operación **modifica** la capacidad del vector.

Para la resolución de este ejercicio proponemos que en la función `main`

- se reserve inicialmente `TAM_INICIAL` casillas,
- se lea un número indeterminado de valores (terminando cuando se introduzca `FIN`),
- se añaden los valores al vector conforme se van leyendo. En el momento de añadir un valor al vector, si se detecta que no se puede añadir porque el array está completo, entonces se llama a la función `Redimensiona` para poder disponer de más casillas.

Escribir la función `main` para que admita argumentos desde la línea de órdenes de manera que el programa pueda ejecutarse:

- a) Sin argumentos. En este caso, el tipo de redimensión será *de uno en uno*.
- b) Con un argumento. El valor admitido será 1, 2 ó 3 de manera que 1 indica de uno en uno, 2 indica en bloques y 3 indica duplicando.

2. Considere las siguientes definiciones de constantes y tipos de datos para gestionar vectores dinámicos. En la figura 25 mostramos un vector dinámico *v* que responde a esta representación en memoria propuesta.

```
// Tipo enumerado para representar tipos de redimensionamiento
enum class TipoRedimension {DeUnoEnUno, EnBloques, Duplicando};

// Capacidad inicial
const int TAM_INICIAL = 10;

// Tamaño del bloque para redimensionar (modalidad EnBloques)
const int TAM_BLOQUE = 5;

typedef int TipoBase; // Tipo de los datos almacenados

typedef struct {
    TipoBase * datos; // Puntero para acceder a los datos
    int usados; // Num. casillas usadas
    int capacidad; // NUm. casillas ocupadas

    // PRE: 0 <= usados <= capacidad
    // Inicialmente, capacidad = TAM_INICIAL
} VectorDinamico;
```

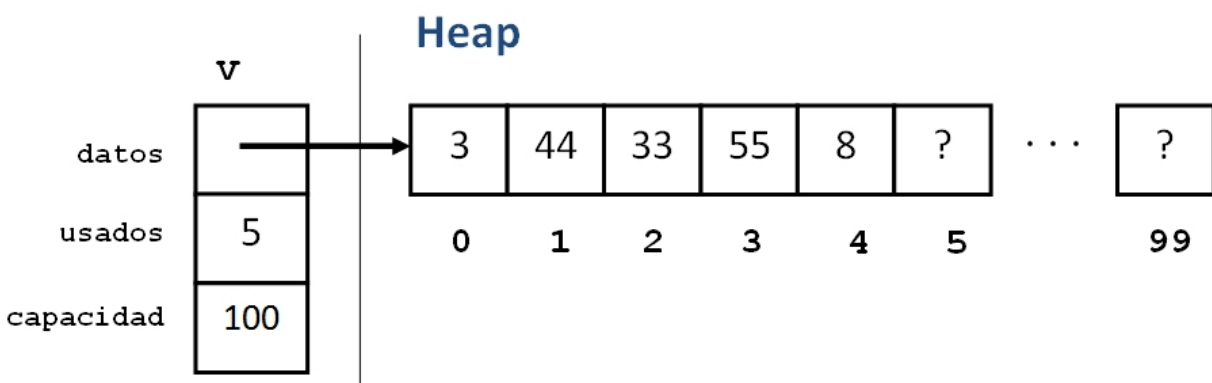


Figura 25: Vector dinámico con 100 casillas reservadas y 5 usadas

Se deben construir las funciones cuyos prototipos se enumeran a continuación. Estas funciones dotarán de total funcionalidad a los datos de tipo *VectorDinamico*.

```
// Crear vector dinámico
VectorDinamico CreaVectorDinamico (int cap_inic=TAM_INICIAL);

// Libera la memoria ocupada por un vector dinámico
void LiberaVectorDinamico (VectorDinamico &v);

// Devuelve el número de casillas usadas en un vector dinámico
int UsadosVectorDinamico (const VectorDinamico v);

// Devuelve el número de casillas reservadas en un vector dinámico
int CapacidadVectorDinamico (const VectorDinamico v);

// Devuelve el valor almacenado en una casilla del vector dinámico
// PRE: 0<=num_casilla<UsadosVectorDinamico(v)
TipoBase ElementoVectorDinamico (const VectorDinamico v,
                                   int num_casilla);

// Muestra un vector dinámico
void MuestraVectorDinamico (const VectorDinamico v);

// Añade un valor al vector dinámico
// PRE: hay espacio disponible en el vector dinámico
void AniadeVectorDinamico (VectorDinamico &v,
                           const TipoBase valor);

// Redimensiona un vector dinámico de acuerdo a tipo.
// Copia todos los datos en el nuevo vector.
void RedimensionaVectorDinamico (VectorDinamico &v,
                                 TipoRedimension tipo);

// Redimensiona un vector dinámico para que no haya espacio libre.
// Copia todos los datos en el nuevo vector.
// POST: CapacidadVectorDinamico(v) = UsadosVectorDinamico(v)
void ReajustaVectorDinamico (VectorDinamico &v);
```

Modularizar la solución en los ficheros `VectorDinamico.h` y `VectorDinamico.cpp`.

Reescribir la solución del ejercicio 1 usando las funciones desarrolladas.

- Retomamos el ejercicio 18 de la *Relación de Problemas I*. En ese ejercicio, cuando el *array* de punteros que delimitan las palabras de una cadena se llena, no se tienen en cuenta las siguientes palabras de la cadena y no se registran.

Ahora planteamos reescribir la solución usando un vector dinámico para `las_palabras`. Ahora, cuando la función `encuentra_palabras` se encuentra con que no puede registrar nuevas palabras, deberá redimensionar el vector dinámico de datos `info_palabra` para poder seguir registrando palabras.

Usad redimensionamiento por bloques de tamaño fijo. Antes de finalizar la función, deberá reajustar el espacio reservado para ocupar lo estrictamente necesario.

En la figura 26 mostramos el estado de la memoria antes de la finalización de la función `encuentra_palabras`. En este ejemplo la función devolverá 2 y se llamó:

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

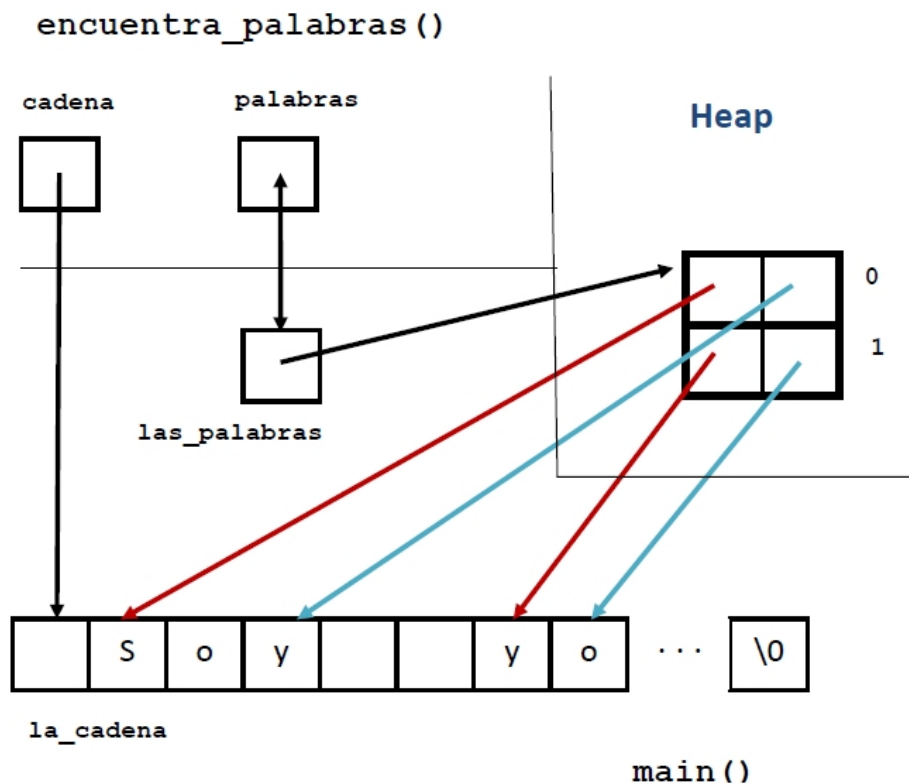


Figura 26: Estado de la pila antes de devolver el control a `main`

4. Modificar la solución propuesta para el problema de la *criba de Eratóstenes* (ver guión de prácticas de **Fundamentos de Programación**).

El programa calculará en primer lugar los primeros números primos menores que un número natural dado, *p*, y los guardará en un vector dinámico. Ese vector tendrá las casillas estrictamente necesarias.

Utilice el vector dinámico de primos para realizar la descomposición en factores primos de un número entero dado, *n*.

Nota: Modularice la solución en funciones.

Escribir la función `main` para que admita argumentos desde la línea de órdenes de manera que el programa pueda ejecutarse:

- a) *Con un argumento:* el programa calculará la descomposición en factores primos del número dado (*n*). En este caso se tomará, por omisión, el valor 100 para *p* (los primos que se calculan son menores que *p*=100).

Por ejemplo:

```
% descomposicion 135
```

calcula la descomposición de *n*=135 usando los números primos menores que *p*=100.

- b) *Con dos argumentos:* el programa calculará la descomposición en factores primos del primer número, usando los números primos menores que el segundo.

Por ejemplo:

```
% descomposicion 135 300
```

calcula la descomposición de *n*=135 usando los números primos menores que *p*=300.

Considerar todos los posibles casos de error.

5. Escriba un programa que lea una secuencia indeterminada de cadenas de caracteres (termina al encontrar en la lectura el *fin de fichero*) y las guarda en memoria (incluidas las líneas vacías), de manera que sean accesibles a través de un vector dinámico de cadenas clásicas (vector dinámico de punteros a `char`).

El programa mostrará el número total de líneas, líneas no vacías y párrafos.

El vector dinámico empezará teniendo una capacidad de 10 e irá creciendo de uno en uno.

Nota: Recomendamos la ejecución usando la redirección de entrada para poder leer las líneas desde un fichero de texto.

Cuestiones técnicas sobre la lectura de los datos.

Los ficheros de texto están organizados en líneas. Cada línea finaliza con el carácter *salto de línea*. El final del fichero está indicado con un carácter especial, *EOF* (End Of File).

Nota: Para la lectura de una línea le sugerimos usar la función `getline` y un dato `string`, aunque puede usar igualmente el método `getline` sobre el objeto `cin` y guardar el resultado en un array de datos `char` (cada línea será una cadena clásica).

RELACIÓN DE PROBLEMAS II. Memoria dinámica

La función `getline` se usa habitualmente como si fuera una función `void` aunque no lo es (consultar el manual). Podría emplearse *como* si fuera una función `bool` (tampoco lo es, pero en la práctica puede simplificarse su comportamiento de esta manera). Supondremos entonces que si en la lectura de la cadena encuentra el *fin de fichero* devuelve `false`. Si está redireccionada la entrada, el ciclo:

```
string cadena;
.....
while (getline (cin, cadena)) {
    .....
}
```

es capaz de leer una línea del fichero en cada iteración, y leerlas y procesarlas todas. Si una línea está vacía (en el fichero solo hay un salto de línea) su longitud será cero.

Una vez leída cada línea (en un dato `string`) habrá que crear un vector dinámico de caracteres -con la capacidad estrictamente necesaria- y copiar los caracteres desde el `string` para formar una cadena clásica. A continuación, añadir esa cadena al vector dinámico de cadenas clásicas.

Una vez finalizada la lectura de todas las líneas se procederá a su procesamiento.

En la figura 27.A mostramos un ejemplo en el que se ha copiado en un vector de cadenas clásicas (`v`) un total de cinco líneas (dos de ellas vacías). Estas líneas podrían haber estado en un fichero de texto, dispuestas como se indica en la figura 27.B. En esa figura, *EOF* indica la marca de *fin de fichero* que se introduce manualmente desde el teclado con la combinación de teclas `Ctrl+D` en Linux (`Ctrl+Z` en Windows).

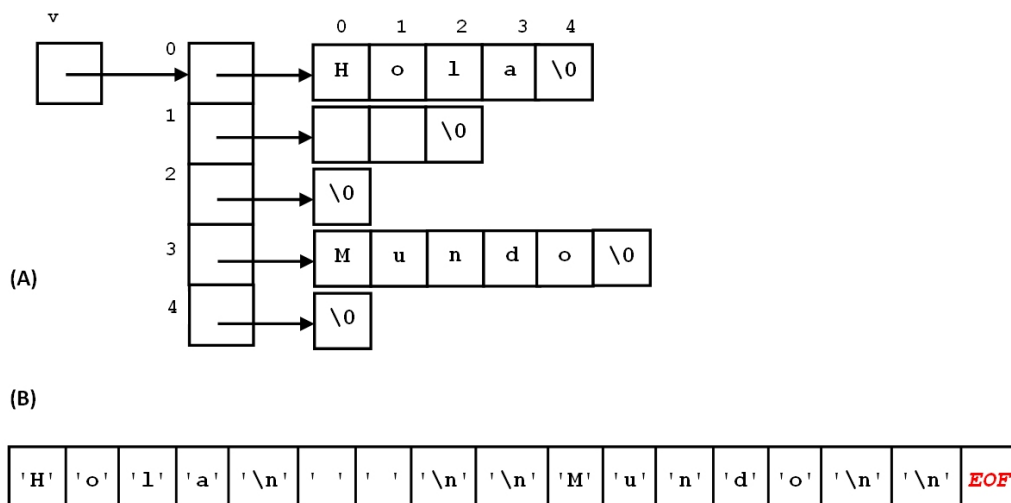


Figura 27: Cada línea del fichero (B) se guarda en una cadena clásica, accesible desde `v`, un vector dinámico de punteros a `char` (A)

Ejercicios sobre matrices dinámicas

6. Supongamos que para definir matrices bidimensionales dinámicas de datos de tipo `TipoBase` usamos una estructura como la que aparece en la figura 28 (tipo `Matriz2D_1`, filas independientes) en la que ilustramos cómo se almacena en memoria una matriz dinámica de 10 filas y 15 columnas.

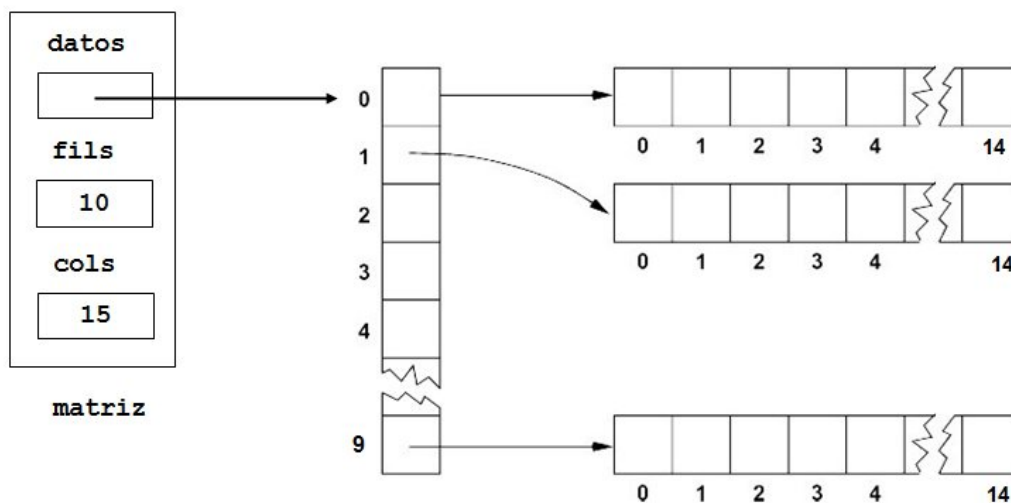


Figura 28: Tipo `Matriz2D_1`: datos guardados en filas independientes

- Construir una función que cree (reserve memoria para) una matriz de este tipo, recibiendo el número de filas y columnas. El contenido de la matriz queda *indefinido*.
- Construir una función que libere la memoria ocupada por una matriz de este tipo. La función deja la matriz *vacía*.
- Construir una función que cree una matriz (como en el apartado 6a) pero además lea del teclado los valores y los copie en la matriz. La matriz se rellena completamente.
- Construir una función que muestre los valores guardados en la matriz.
- Construir una función que devuelva un elemento de la matriz dinámica, indicando su fila y su columna.
- Construir una función que modifique el valor de un elemento de la matriz dinámica, indicando su fila y su columna, y el nuevo valor.
- Construir una función que reciba una matriz de ese tipo, y cree y devuelva una copia (copia *profunda*). La copia es una *nueva* matriz.

RELACIÓN DE PROBLEMAS II. Memoria dinámica

- h) Construir una función que extraiga una submatriz. Como argumento de la función se introduce desde qué fila y columna y hasta qué fila y columna se debe realizar la copia de la matriz original. La submatriz devuelta es una *nueva* matriz.
- i) Construir una función que elimine una fila de una matriz. Obviamente, no se permiten “huecos” (filas vacías). La función devuelve una *nueva* matriz.
- j) Construir una función como el anterior, pero que en vez de eliminar una fila, elimine una columna. La función devuelve una *nueva* matriz.
- k) Construir una función que devuelva (*nueva* matriz) la traspuesta de una matriz.
- l) Construir una función que reciba una matriz y devuelva una *nueva* matriz con las filas “invertidas”: la primera fila de la nueva matriz será la última fila de la primera, la segunda será la penúltima, y así sucesivamente.

Modularice la solución en dos ficheros: `Matriz2D_1.h` y `Matriz2D_1.cpp` y escriba un fichero con una función `main` que ilustre el uso de las funciones.

7. Supongamos que ahora decidimos utilizar una forma diferente para representar las matrices bidimensionales dinámicas a la que se propone en el ejercicio 6. Usaremos una estructura semejante a la que aparece en la figura 29 (tipo `Matriz2D_2`, todas las casillas consecutivas formando una única fila) en la que ilustramos cómo se almacena en memoria una matriz dinámica de 10 filas y 15 columnas.

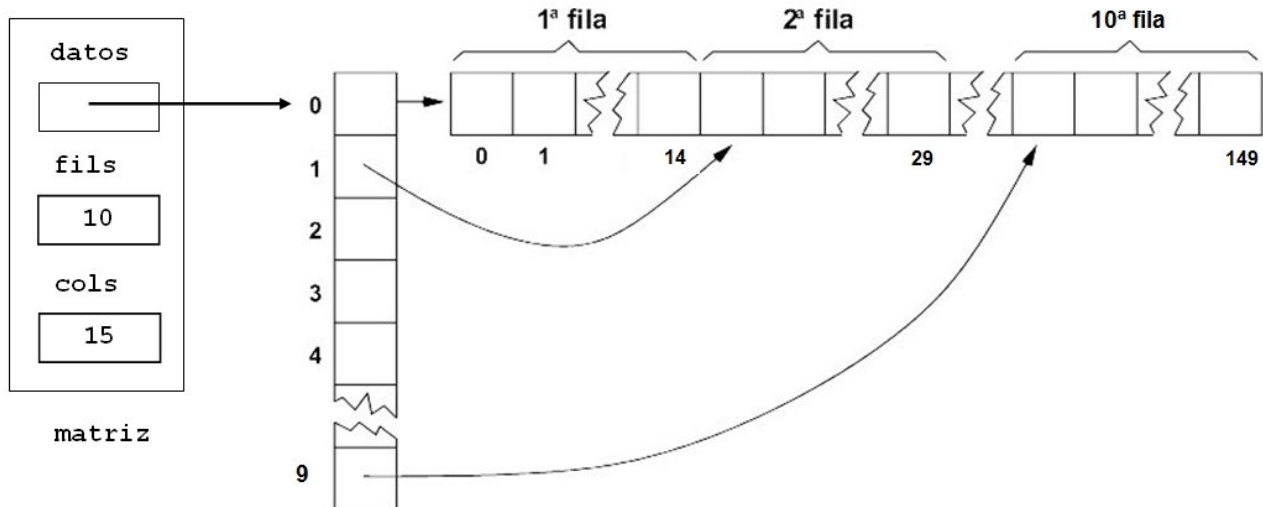


Figura 29: Tipo `Matriz2D_2`: datos guardados en una sola fila

Reescribir todos los módulos propuestos en el ejercicio 6.

Modularice la solución en dos ficheros: `Matriz2D_2.h` y `Matriz2D_2.cpp` y escriba un fichero con una función `main` que ilustre el uso de las funciones.

Nota: La función `main` debería ser, básicamente, la misma que empleó en el ejercicio 6.

8. Una vez implementados los módulos que gestionan matrices dinámicas bidimensionales (ejercicios 6 y 7),
- Construir una función que, dada una matriz `Matriz2D_1`, realice una copia de la misma en una matriz `Matriz2D_2` y la devuelva.
 - Desarrollar una función que realice el paso inverso, convertir de `Matriz2D_2` a `Matriz2D_1` y la devuelva.
9. *(Examen ordinario FP 2018)* **El Problema de la Asignación.**

Una empresa de servicios dispone de n técnicos y n pedidos por atender. La empresa dispone de una matriz $B^{n \times n}$ de tarifas donde cada b_{ij} indica el precio que cobra el técnico i por atender el pedido j . Supondremos que b_{ij} es un entero mayor estricto que 0 y menor o igual que 100.

Resolver el problema implica asignar los técnicos a los pedidos y esta asignación se puede modelizar mediante una matriz $X^{n \times n}$ donde $x_{ij} = 1$ significa que el técnico i atiende el pedido j , y $x_{ij} = 0$ en caso contrario. Una asignación válida es aquella en la que a cada técnico solo le corresponde un pedido y cada pedido está asignado a un único técnico. Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^n \sum_{j=1}^n x_{ij} \times b_{ij}$$

Se pide implementar un programa que permita obtener una asignación (asignar valores a la matriz X). Para ello, se utilizará la siguiente estrategia: **asigne a cada técnico el pedido más económico entre los que están disponibles**. Al final, el programa debe mostrar la asignación (la matriz X) y el coste de la misma.

Por ejemplo: si

$$B = \begin{bmatrix} 21 & 12 & 31 \\ 16 & 14 & 25 \\ 12 & 18 & 20 \end{bmatrix} \quad \text{entonces (empezando por el técnico 1)} \quad X = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

El coste de dicha asignación es $12 + 16 + 20 = 48$.

El programa pedirá el número de pedidos/técnicos (n) y los valores de la matriz de costes B . Finalmente, mostrará las asignaciones efectuadas y el coste total.

Todas las matrices y vectores serán estructuras de datos **dinámicas**.

10. *(Examen extraordinario FP 2018)* **El problema del viajante de comercio.**

Un diligente agente de comercio tiene que visitar periódicamente a sus n clientes. Cada uno de ellos vive en una ciudad distinta. Su jefe le ordena que prepare el plan de viaje del mes siguiente según las siguientes reglas:

- Tiene que visitar a todos los clientes. Debe realizar una única visita a cada uno.
- El recorrido se hará de forma que el coste total sea el menor posible.

- No importa por qué ciudad comience, pero debe finalizar en la misma ciudad en la que comenzó. Por tanto, el recorrido forma un ciclo. Y, obviamente, no puede volver a la ciudad inicial hasta haber visitado todas las demás.

Para resolver el problema se necesita conocer el número de ciudades, n , la ciudad inicial y el coste del viaje que hay entre cada par de ciudades. Los costes de los desplazamientos se encuentran en una matriz C de tamaño $n \times n$, donde C_{ij} es el coste de viajar desde la ciudad i a la j . Obviamente, estos valores son todos positivos, $C_{ij} > 0$.

Este es un problema muy difícil de resolver para el que existen distintas soluciones aproximadas. Uno de esos algoritmos, llamado **heurística del vecino más cercano**, funciona así:

Se selecciona una ciudad de partida, a , y se busca b , la ciudad más económica para llegar desde a . A continuación, se busca la ciudad aún no visitada más económica desde b . Y así hasta completar la visita a todas las ciudades. El coste del recorrido es la suma de todos los costes intermedios.

Por ejemplo, supongamos un problema con $n = 4$ cuya matriz de costes es:

$$C = \begin{bmatrix} - & 10 & 15 & 9 \\ 11 & - & 13 & 8 \\ 17 & 21 & - & 15 \\ 26 & 7 & 14 & - \end{bmatrix}$$

Por simplicidad, asumimos que los nombres de las ciudades son 1, 2, ..., n . Si partimos de la ciudad 1, entonces la solución encontrada es: 1, 4, 2, 3 con un coste de $9 + 7 + 13 + 17 = 46$. Si el viaje comienza por la ciudad 3, el recorrido final sería 3, 4, 2, 1 con un coste de $15 + 7 + 11 + 15 = 48$.

El programa pedirá primero el número de ciudades, n , y los valores de la matriz de costes C . A continuación pedirá la ciudad inicial. Luego calculará el itinerario y posteriormente su coste. Finalmente mostrará el itinerario calculado y su coste.

Todas las matrices y vectores serán estructuras de datos **dinámicas**.

11. (Este ejercicio está basado en el ejercicio 24 de la Relación de Problemas I)

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

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

El programa debe leer un número *indeterminado* de datos de ventas: la lectura de datos finaliza cuando se encuentra el valor -1 como código de sucursal. **Recomendación:** Leer los datos utilizando la redirección de entrada. Usad para ello un fichero de texto como los disponibles en la página de la asignatura.

RELACIÓN DE PROBLEMAS II. Memoria dinámica

- No se conoce a priori el número de operaciones de venta. Tampoco se conoce el número de sucursales que se van a procesar, ni el código de éstas (algunas sucursales puede que no hayan realizado ventas). Se sabe que los códigos de sucursales son números entre 1 y 100.
- Tampoco se conoce a priori los productos que se han vendido, ni el código de éstos (algunos productos puede que no hayan vendido). Se sabe que los códigos de producto son caracteres entre 'a' y 'j'.

Después de leer los datos de ventas el programa mostrará la misma información que en el ejercicio 24 de la Relación de Problemas I.

Para poder guardar en memoria la información necesaria para los cálculos proponemos una **matriz bidimensional dinámica** `ventas` con tantas filas *útiles* (con memoria reservada) como número de sucursales activas (con ventas).

Si la sucursal s ha estado activa, la casilla (s, p) de esta matriz guardará el número total de unidades vendidas por la sucursal s del producto p (o el número de unidades vendidas del producto p en la sucursal s).

Notas finales (ver ejercicio 24):

- a) Modularizar con funciones la solución desarrollada.
- b) Emplear datos `int` para los índices de las filas y `char` para las columnas.
- c) Usar los vectores `ventas_sucursal` y `ventas_producto`.

Ejercicios sobre listas

Los siguientes ejercicios gestionan listas enlazadas. Hay una serie de tareas comunes (crear listas, liberar la memoria ocupada, contar el número de nodos, etc.) que se repiten en muchos de los ejercicios y que podrían ser útiles en otros programas que gestionaran listas.

Con objeto de evitar duplicar código, construyan la biblioteca `libLista.a` que ofrezca a través de `Lista.h` los *tipos de datos* y las *funciones* que considere adecuadas para la gestión de listas enlazadas.

Esta biblioteca se enlazará con el fichero objeto que contiene la función `main` que resuelve cada uno de los ejercicios indicados para generar el correspondiente ejecutable. Se construye a partir de:

- `Lista.h`: declaración de los tipos `PNode` y `Lista`, y los prototipos de las funciones.

Los tipos `PNode` y `Lista` son **dos alias** del tipo *puntero a Nodo*:

```
Tipo de los elementos de la lista
// Bastará cambiar el tipo asociado al alias TipoBase
// y recompilar para poder gestionar otro tipo de lista.

typedef double TipoBase;

// Cada nodo de la lista es de tipo "Nodo"

struct Nodo {
    TipoBase valor;
    Nodo *sig;
};

typedef Nodo * PNode; // Para los punteros a nodos
typedef Nodo * Lista // Para la lista
```

- `Lista.cpp`: implementación de las funciones.

12. Escriba un programa para que lea una secuencia con un número indefinido de valores `double` hasta que se introduzca un valor negativo.

Estos valores (excepto el último, el negativo) los almacenará en una estructura de celdas enlazadas (una *lista*) y después mostrará los valores almacenados.

A continuación calculará, sobre los datos almacenados en la lista:

- a) el *número de datos* almacenados.
- b) la *media* de los datos almacenados.
- c) la *varianza* de los datos almacenados.

RELACIÓN DE PROBLEMAS II. Memoria dinámica

Nota: Deberá implementar una función para liberar la memoria ocupada. Esta función se empleará en todos los programas que gestionen listas.

Nota: Es aconsejable implementar una función que compruebe si la lista está vacía.

Funciones a implementar (sugerencia):

```
void LeeLista (Lista & l);
void PintaLista (const Lista l);
void LiberaLista (Lista & l);
bool ListaVacía (const Lista l);
int CuentaElementos (const Lista l);
```

Estas funciones serán útiles para solucionar muchos problemas que gestionen listas: deberían ser parte de la biblioteca.

Para efectuar los cálculos indicados, sugerimos implementar las funciones:

```
double Media (const Lista l);
double Varianza (const Lista l);
```

Estas funciones pueden acompañar a la función `main` en el mismo fichero. No recomendamos su inclusión en la biblioteca.

13. Escribir un programa que lea una secuencia de valores (o los genere aleatoriamente), los almacene en una lista, determine si la secuencia está ordenada, y los ordene.

La ordenación se debe efectuar sobre la propia lista, sin emplear ninguna estructura de datos auxiliar (array dinámico, otra lista, etc.) simplemente cambiando la posición de los nodos.

Utilice, por ejemplo, el método de *ordenación por selección*.

Funciones a implementar (sugerencia) y que se añaden a la biblioteca:

```
bool EstaOrdenada (const Lista l);
void OrdenaSeleccionLista (Lista &l);
```

14. Considere una secuencia **ordenada** de datos almacenada en una *lista*.

- a) Implemente una función para insertar un nuevo dato en su posición correcta. En el caso que la lista ya tuviera ese valor, se insertará el nuevo delante de la primera aparición de éste.
- b) Implemente una función para, dado un dato, eliminar la celda que lo contiene. En el caso que la lista tuviera ese valor repetido, se eliminará la primera aparición de éste.

Funciones a implementar (sugerencia):

```
void InsertaOrdenadamente (Lista &l, TipoBase v);
void EliminaValor (Lista &l, TipoBase v);
```

15. Considere dos secuencias de datos **ordenadas** almacenadas en sendas *listas*. Implemente una función para *mezclar ordenadamente* las dos secuencias en una nueva, de forma que las dos listas originales se queden *vacías* tras realizar la mezcla y la lista resultante contenga todos los datos.

Observe se trata de una variante del algoritmo *mergesort*. Ahora se exige la modificación de las secuencias originales: en esta versión los datos se “mueven” hacia la lista resultante en lugar de copiarlos.

Funciones a implementar (sugerencia):

```
void MezclaListas (Lista &l, Lista &l1, Lista &l2);
```

Nota: Esta función no realiza ninguna operación de reserva ni liberación de memoria.

16. Implementar un sistema sencillo de gestión de tareas y recursos.

El sistema dispone de una serie limitada de **recursos** que se emplean para dar servicio a una serie indeterminada de **tareas** que solicitan servicios.

- Recursos = cajas de un hipermercado. Tareas = cada cliente esperando en la cola única.
- Recursos = consultas médicas en urgencias. Tareas = Pacientes esperando a ser atendidos, en una cola única.
- Recursos = impresoras. Tareas = trabajos esperando a ser impresos en una cola.
- ...

En esta aplicación, cualquier recurso puede dar servicio a cualquier tarea, siempre que el primero esté libre.

No existe una cola individualizada por recurso sino una única cola en la que se colocan las tareas y desde la que se asignan a los recursos que van quedando libres.

Las tareas se clasifican en tres tipos: 1 ó *crítica*, 2 ó *normal* y 3 ó de *baja prioridad*. La prioridad indica el lugar que se le asigna en la cola de tareas pendientes:

- a) si llega una tarea de tipo 1 se coloca la última de entre las de ese tipo y delante de la primera de tipo 2.
- b) Si llega una tarea de de tipo 2 se coloca la última de entre las de ese tipo y delante de la primera de tipo 3.
- c) si llega una tarea de tipo 3 se coloca la última de todas las tareas.

El programa debe mostrar un menú con una serie de opciones:

- 1.- **Llega nueva tarea.** Ocurre cuando surge una nueva tarea. El programa solicita el tipo de tarea, le asigna un identificador y la coloca en la lista de tareas pendientes.
- 2.- **Asignar tarea a recurso.** Ocurre cuando hay recursos disponibles y tareas pendientes. El sistema solicita el identificador de recurso al que debe ser asignada la tarea.

- 3.- **Recurso solicita tarea.** Ocurre cuando un recurso puede dar servicio a una nueva tarea. Esta situación se produce cuando el recurso termina con una tarea y está disponible para admitir una nueva tarea, cuando estaba en estado de pausa (opción 4) y puede volver a admitir tareas o cuando se habilita (activa) un nuevo recurso. El programa solicita el identificador de recurso que vuelve a estar disponible.
- 4.- **Recurso entra en pausa.** Ocurre si un recurso pasa a estar temporalmente en pausa. Esta circunstancia se produce cuando a) un recurso estaba cumplimentando una tarea y la termina ó b) estaba esperando una tarea. En cualquier caso, desde que se le asigna este estado no está disponible para admitir nuevas tareas. El programa solicita el identificador de recurso que va a quedar fuera de servicio. Este estado se cambia cuando se activa la opción 3 para el recurso.
- 5.- **Activar recurso.** Ocurre cuando un recurso estaba desactivado (no pausado) y pasa a estar en espera de tarea.
- 6.- **Desactivar recurso.** Ocurre cuando un recurso pasa a estar desactivado, fuera de servicio (no pausado). Solo puede ocurrir si es recurso está esperando tarea o pausado.
- 7.- **Finalizar.** Se prepara la finalización de la ejecución del programa. No admite nuevas tareas. Desde ese momento sólo es posible dar servicio a las tareas que están pendientes. En el momento en el que no queden tareas pendientes, ni recursos ocupados, el programa finaliza su ejecución.

En todo momento se debe mostrar la lista de tareas pendientes, y el estado de los recursos.

En la figura 30 mostramos los posibles estados en los que puede estar un recurso, y las transiciones permitidas de acuerdo a las opciones del menú.

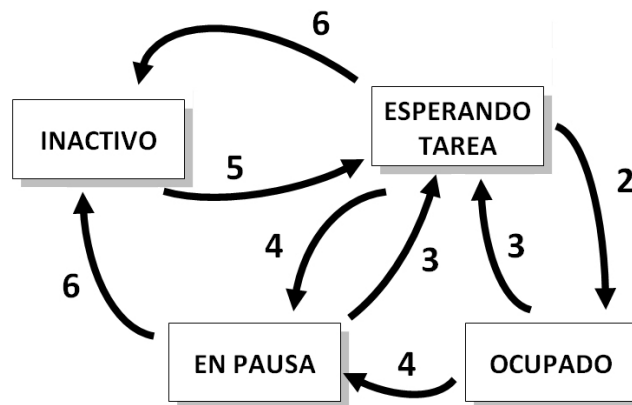


Figura 30: Estados posibles de los recursos y transiciones permitidas

El número máximo de recursos, el número inicial de recursos disponibles, así como su código (un número entero) se fijará en la línea de órdenes, como argumento a `main`.

RELACIÓN DE PROBLEMAS III. Clases (I)

Los ejercicios propuestos tienen como finalidad que el alumno practique con los constructores y el destructor de una clase, así como con métodos que permitan el acceso a los datos y la gestión de los objetos de la la clase.

Todos los ejercicios deben estar **completamente implementados y modularizados**. Significa que debe existir, para cada clase:

- Un fichero `.h` con las declaraciones.
- Un fichero `.cpp` con las definiciones.

Además, deberá escribirse un fichero `.cpp` con la función `main` que contenga ejemplos sobre el uso de la clase.

1. Implementar modularmente la clase `VectorDinamico` para trabajar con vectores de datos de tipo `TipoBase`. Los vectores dinámicos tendrán un tamaño arbitrario, y no definido a priori.

El tipo de redimensionamiento (de uno en uno, en bloques o duplicando el tamaño) será una **propiedad** más de cada objeto. Podrá consultarse/modificarse en tiempo de ejecución con los métodos adecuados.

Proponer una *representación* para la clase e implementar los siguientes *métodos*:

- a) Constructor sin argumentos, que crea un vector dinámico con un número de casillas predeterminado.
- b) Constructor con un argumento, que crea un vector dinámico con un número de casillas indicado en el argumento.
- c) Constructor de copia
- d) Destructor.
- e) Método (valor devuelto: `bool`) que consulta si el vector está vacío.
- f) Métodos para consultar el número de casillas ocupadas/reservadas.
- g) Método para devolver el valor que ocupa una posición dada.
- h) Método para añadir un valor (siempre al final). El método se encargará de redimensionar el vector si no tuviera espacio disponible.
- i) Método para ajustar el tamaño del vector. El objetivo es que la capacidad del vector coincida con el número de casillas ocupadas.
- j) Métodos para establecer/consultar el tipo de redimensionamiento.

Escribir una función `main` que permita probar la clase.

RELACIÓN DE PROBLEMAS III. Clases (I)

2. Implementar la clase `Matriz2D_1` para elementos de tipo `TipoBase`.

Empleando la representación básica conocida, implementar los siguientes *métodos*:

- a) Constructor sin argumentos, que crea una matriz *vacía*.
- b) Constructor con un argumento, que crea una matriz *cuadrada* con el número de filas y columnas indicado en el argumento.
- c) Constructor con dos argumentos, que crea una matriz con el número de filas indicado en el primer argumento y con el número de columnas indicado en el segundo.
- d) Constructor con tres argumentos, que crea una matriz con el número de filas indicado en el primer argumento y con el número de columnas indicado en el segundo argumento. Además inicia todas las casillas de la matriz al valor especificado con el tercer argumento.
- e) Constructor de copia.
- f) Destructor.
- g) Método (valor devuelto: `bool`) que consulta si la matriz está vacía.
- h) Métodos para escribir/leer un valor. Responderán a los prototipos:
`void ModificarValor (int fila, int col, TipoBase val);`
`TipoBase LeerValor (int fila, int col);`
- i) Método que inicializa todas las casillas de la matriz al valor indicado como argumento. Si no se especifica ninguno, inicia todas las casillas al valor *nulo*.

Escribir una función `main` que permita probar la clase.

3. Implementar la clase `Matriz2D_2` para elementos de tipo `TipoBase`.

Se trata de implementar los mismos métodos que en el problema 2.

Escribir una función `main` que permita probar la clase.

4. Implementar la clase `Lista` para trabajar con listas (de tamaño arbitrario, y no definido a priori, cuyos nodos residen en el *heap*) de datos de tipo `TipoBase`.

Proponer una *representación* para la clase (basada en el almacenamiento de los nodos en memoria dinámica) e implementar los siguientes *métodos*:

- a) Constructor sin argumentos, que crea una lista *vacía*.
- b) Constructor con un argumento, que crea una lista con un número de nodos indicado en el argumento.
- c) Constructor con dos argumentos, que crea una lista con un número de nodos indicado en el primer argumento. Inicia todos los nodos de la lista al valor indicado en el segundo argumento.
- d) Constructor de copia.
- e) Destructor.
- f) Método (valor devuelto: `bool`) que consulta si la lista está vacía.

RELACIÓN DE PROBLEMAS III. Clases (I)

g) Método para consultar el número de nodos de la lista.

h) Método para insertar un valor en la lista. Modifica la lista.

Responderá al siguiente prototipo:

```
void Insertar (TipoBase valor, int pos);
```

de manera que inserta un nuevo nodo en la lista con valor `val` en la posición `pos` (1 para el primer nodo, 2 para el segundo, etc.). La posición seguirá el siguiente convenio: `pos` indica el número de orden que ocupará el nuevo nodo que se va a insertar.

Algunos ejemplos (si `TipoBase` es `int`):

Antes: < 6, 8, 4, 3, 2, 9 > Insertar (5, 2) Después: < 6, 5, 8, 4, 3, 2, 9 >

Antes: < 6, 8, 4, 3, 2, 9 > Insertar (5, 6) Después: < 6, 8, 4, 3, 2, 5, 9 >

Antes: < 6, 8, 4, 3, 2, 9 > Insertar (5, 7) Después: < 6, 8, 4, 3, 2, 9, 5 >

Antes: < 6, 8, 4, 3, 2, 9 > Insertar (5, 1) Después: < 5, 6, 8, 4, 3, 2, 9 >

i) Método para borrar un nodo en la lista. Responderá al siguiente prototipo:

```
void Borrar (int pos);
```

de manera que borra el nodo que ocupa la posición `pos` (1 para el primer nodo, 2 para el segundo, etc.)

j) Método para añadir un valor en la lista. La adición siempre se hace al final de la lista. Modifica la lista. Responderá al siguiente prototipo:

```
void AniarValor (TipoBase valor);
```

k) Métodos para leer/escribir un valor.

```
TipoBase LeerValor (int pos);
```

```
void ModificarValor (int pos, TipoBase val);
```

de tal manera que `pos` indica la posición del nodo (1 para el primer nodo, 2 para el segundo, etc.)

l) Método que inicializa todos los nodos al valor indicado como argumento.

Escribir una función `main` que permita probar la clase.

5. Implementar la clase `Pila`.

Una *pila* es una estructura de datos que permite la gestión de problemas en los que la gestión se realiza empleando un protocolo **LIFO** (last in first out).

Proponer una *representación* para la clase (basada en el almacenamiento de los nodos en memoria dinámica) e implementar los siguientes *métodos*:

a) Constructor sin argumentos, que crea una pila *vacía*.

b) Constructor de copia.

c) Destructor.

d) Método (valor devuelto: `bool`) que consulta si la pila está *vacía*.

RELACIÓN DE PROBLEMAS III. Clases (I)

- e) Método para añadir un valor. La pila se modifica.
- f) Método para sacar un valor. Obtiene (devuelve) el elemento extraído. La pila se modifica.
- g) Método para consultar qué elemento está en el **tope** de la pila. La pila no se modifica.

Escribir una función `main()` que permita probar la clase.

6. Implementar la clase `Cola`.

Una *cola* es una estructura de datos que permite la gestión de problemas en los que la gestión se realiza empleando un protocolo **FIFO** (**F**irst **i**n **f**irst **o**ut).

Proponer una *representación* para la clase (basada en el almacenamiento de los nodos en memoria dinámica) e implementar los siguientes *métodos*:

- a) Constructor sin argumentos, que crea una cola *vacía*.
- b) Constructor de copia.
- c) Destructor.
- d) Método (valor devuelto: `bool`) que consulta si la cola está *vacía*.
- e) Método para añadir un valor. La cola se modifica.
- f) Método para sacar un valor. Obtiene (devuelve) el elemento extraído. La cola se modifica.
- g) Método para consultar qué elemento está en la **cabecera** de la cola.
La cola no se modifica.

Escribir una función `main()` que permita probar la clase.

RELACIÓN DE PROBLEMAS IV. Clases (II)

Los ejercicios propuestos tienen como finalidad que el alumno practique con la sobrecarga de operadores, especialmente la del **operador de asignación** empleando código reutilizable y sepa distinguirlo claramente del constructor de copia. Otros operadores que se van a sobrecargar son:

- operadores de acceso `[]` y `()`
- operadores relacionales
- operadores aritméticos
- operadores combinados
- operadores sobre flujos

Muchos de estos ejercicios amplían las clases diseñadas e implementadas como solución a los ejercicios propuestos en la *Relación de Problemas III (Clases I)*. En cualquier caso, todos los ejercicios deben estar completamente implementados y modularizados continuando y complementando el trabajo ya realizado. Significa que debe existir, para cada clase:

- Un fichero `.h` con las declaraciones.
- Un fichero `.cpp` con las definiciones.

Además, deberá escribirse un fichero `.cpp` con la función `main` que contenga ejemplos sobre el uso de la clase.

1. Ampliar la clase `VectorDinamico` de datos de tipo `TipoBase` implementando los siguientes métodos:
 - a) Sobrecarga del operador de asignación (con código reutilizable).
 - b) Una sobrecarga alternativa del operador de asignación, que recibe como argumento un dato de tipo `TipoBase` e inicia **todo** el vector al valor especificado.
 - c) Sobrecargar el operador `[]` para que sirva de operador de acceso a los elementos del vector dinámico y pueda actuar de *lvalue* y *rvalue*.
 - d) Sobrecargar los operadores relacionales binarios `==` y `!=` para comparar dos vectores dinámicos. Dos vectores serán iguales si tienen el mismo número de casillas ocupadas y los contenidos son iguales y en las mismas posiciones.
 - e) Operadores relacionales binarios `>`, `<`, `>=` y `<=` para comparar dos vectores dinámicos. Usar un criterio similar al que se sigue en la comparación de dos cadenas de caracteres clásicas.
 - f) Sobreescibir los operadores `<<` y `>>` para leer/escribir un vector dinámico.

Notas:

RELACIÓN DE PROBLEMAS IV. Clases (II)

- Para la implementación del operador >> leerá una secuencia indefinida de valores, hasta que se introduzca el valor *. Los valores se leerán en una cadena de caracteres, y sólo se convertirán al tipo **TipoBase** cuando se verifique que son válidos para su almacenamiento (no se ha introducido el terminador *).
- Los valores siempre se guardarán *al final*.

2. Ampliar la clase **Matriz2D-1** de datos de tipo **TipoBase** con los siguientes *métodos*:

- a) Sobrecarga del operador de asignación (con código reutilizable).
- b) Una sobrecarga alternativa del operador de asignación, que recibe como argumento un dato de tipo **TipoBase** e inicia **toda** la matriz al valor especificado.
- c) Sobrecargar el operador () para que sirva de operador de acceso a los elementos de la matriz dinámica y pueda actuar de *lvalue* y *rvalue*.
- d) Sobrecargar los operadores unarios + y -.
- e) Sobrecargar los operadores binarios + y - para implementar la suma y resta de matrices. Si los dos operadores son de tipo **Matriz2D-1** sólo se hará la suma o la resta si las dos matrices tienen las mismas dimensiones. Si no fuera así, devolverá una matriz vacía. Se admite la posibilidad de que algún operando fuera de tipo **TipoBase**.
Importante: ninguno de los operandos se modifica.
- f) Sobrecargar los operadores combinados += y -= de manera que el argumento explícito sea de tipo **TipoBase** y modifiquen la matriz convenientemente.
- g) Sobrecargar los operadores == y != para comparar dos matrices dinámicas: serán iguales si tienen el mismo número de filas y columnas, y los contenidos son iguales y en las mismas posiciones.
- h) Sobreescibir el operador << para mostrar el contenido de una matriz dinámica.

3. Ampliar la clase **Matriz2D-2**.

Empleando la representación básica conocida, se trata de implementar los mismos métodos que en el problema 2.

4. Ampliar la clase **Lista** de datos de tipo **TipoBase** con los siguientes *métodos*:

- a) Sobrecarga del operador de asignación (con código reutilizable).
- b) Una sobrecarga alternativa del operador de asignación, que recibe como argumento un dato de tipo **TipoBase** e inicia **toda** la lista al valor especificado.
- c) Sobrecargar el operador [] para que sirva de operador de acceso a los elementos de la lista y pueda actuar de *lvalue* y *rvalue*. El índice hace referencia a la posición, de tal manera que 1 indica el primer nodo, 2 el segundo, etc.)
- d) Sobrecargar los operadores combinados += y -= en los que el argumento explícito sea de tipo **TipoBase** y añadan o eliminen de la lista un nodo con el valor dado por el argumento explícito.
 - El nuevo nodo se añade al final de la lista.

RELACIÓN DE PROBLEMAS IV. Clases (II)

- Si hay más de una instancia del valor a borrar, se borra la primera de ellas.
 - Si no hubiera ninguna instancia del valor a borrar no se hace nada.
- e) Sobrecargar los operadores binarios + y – de manera que se apliquen a dos listas, a una lista y un dato de tipo **TipoBase** o a un dato de tipo **TipoBase** y a una lista, creando una *nueva* lista.
- El operador + cuando se aplica a dos listas crea una nueva lista, uniendo a la primera el contenido completo de la segunda.
 - El operador – cuando se aplica a dos listas crea una nueva lista, eliminando de la primera *la primera aparición* de los valores que aparecen en la segunda. Si no hubiera ninguna instancia del valor a borrar, no se hace nada.
 - Estudie si tiene sentido implementar las tres versiones (lista/lista, lista/valor y valor/lista) para los dos operadores.
 - Los operandos no se modifican.
- f) Sobrecargar los operadores relacionales para comparar dos listas. Los criterios de comparación entre dos listas serán idénticos a los que determinan la relación entre dos cadenas de caracteres clásicas.
- g) Sobreescibir los operadores << y >> para leer/escribir una lista.
- Para la implementación del operador >> leerá una secuencia indefinida de valores, hasta que se introduzca el valor *. Los valores se leerán en una cadena de caracteres, y sólo se convertirán al tipo **TipoBase** cuando se verifique que son válidos para su almacenamiento (no se ha introducido el terminador *).
 - Los valores siempre se guardarán *al final*.

5. Ampliar la clase **Pila** de datos de tipo **TipoBase** con los siguientes *métodos*:

- a) Sobrecarga del operador de asignación (con código reutilizable).
- b) Sobrecargar el operador unario ~ para obtener el elemento de tipo **TipoBase** que ocupa la cabecera de la pila. La pila no se modifica.
- c) Sobrecargar el operador combinado += para añadir un dato de tipo **TipoBase** a la pila.
- d) Sobrecargar el operador unario -- para eliminar un dato de tipo **TipoBase** de la pila.
- e) Sobreescibir el operador <<

6. Ampliar la clase **Cola** de datos de tipo **TipoBase** con los siguientes *métodos*:

- a) Sobrecarga del operador de asignación (con código reutilizable).
- b) Sobrecargar el operador unario ~ para obtener el elemento de tipo **TipoBase** que ocupa la primera posición de la cola. La cola no se modifica.
- c) Sobrecargar el operador combinado += para añadir un dato de tipo **TipoBase** a la cola.
- d) Sobrecargar el operador unario -- para eliminar un dato de tipo **TipoBase** de la cola.
- e) Sobreescibir el operador <<

7. Implementa la clase **Conjunto** que permita manipular un conjunto de elementos de tipo **TipoBase**. Debe entenderse que se está empleando el concepto *matemático* de conjunto.

Para la representación interna de los datos usar una **lista** de celdas enlazadas. El orden de los elementos no es importante desde un punto de vista teórico, pero aconsejamos que se mantengan los elementos **ordenados** para facilitar la implementación de los métodos.

La clase **Conjunto** debe contener, al menos, las siguientes operaciones:

- a) Constructor sin argumentos: crea un conjunto vacío.
- b) Constructor con un argumento de tipo **TipoBase**: crea un conjunto con un único elemento (el proporcionado como argumento).
- c) Constructor de copia (empleando código reutilizable).
- d) Destructor (empleando código reutilizable).
- e) Método que consulta si el conjunto está *vacío*.
- f) Método para añadir un elemento al conjunto. Prototipo:

```
void Aniadir (TipoBase valor);
```

Añade al conjunto el valor indicado.
Si ya existe, no hace nada (no puede haber elementos repetidos).
- g) Método para eliminar un elemento del conjunto. Prototipo:

```
void Eliminar (TipoBase valor);
```

Elimina del conjunto el valor indicado.
Si no pertenece al conjunto, no hace nada.
- h) Sobregarga del operador de asignación (empleando código reutilizable).
- i) Método que nos diga cuantos elementos tiene el conjunto.
- j) Método que reciba un dato de tipo **TipoBase** y consulte si pertenece al conjunto.
- k) Sobrecargar los operadores relacionales binarios **==** y **!=** para comparar dos conjuntos. Dos conjuntos serán iguales si tienen el mismo número de elementos y los mismos valores (independientemente de su posición).
- l) Operador binario **+** para calcular la **unión** de dos conjuntos. Responderá a las siguientes situaciones:
 - Si **A** y **B** son datos de tipo **Conjunto**, **A+B** será otro dato de tipo **Conjunto** y contendrá $A \cup B$
 - Si **A** es un dato de tipo **Conjunto** y **a** es un dato de tipo **TipoBase**, **A+a** será un dato de tipo **Conjunto** y contendrá $A \cup \{a\}$
 - Si **A** es un dato de tipo **Conjunto** y **a** es un dato de tipo **TipoBase**, **a+A** será un dato de tipo **Conjunto** y contendrá $\{a\} \cup A$
- m) Sobreescibir el operador binario **-** para calcular la **diferencia** de dos conjuntos. Responderá a las siguientes situaciones:

RELACIÓN DE PROBLEMAS IV. Clases (II)

- Si A y B son datos de tipo **Conjunto**, $A - B$ será otro dato de tipo **Conjunto** y contendrá $A - B$, o sea, el resultado de quitar de A los elementos que están en B .
 - Si A es un dato de tipo **Conjunto** y a es un dato de tipo **TipoBase**, $A - a$ será un dato de tipo **Conjunto** y contendrá $A - \{a\}$, o sea, el resultado de eliminar del conjunto A el elemento a .
- n) Sobreescibir el operador binario $*$ para calcular la **intersección** de dos conjuntos. Responderá a las siguientes situaciones:
- Si A y B son datos de tipo **Conjunto**, $A * B$ será otro dato de tipo **Conjunto** y contendrá $A \cap B$
 - Si A es un dato de tipo **Conjunto** y a es un dato de tipo **TipoBase**, $A * a$ será un dato de tipo **Conjunto** y contendrá $A \cap \{a\}$
 - Si A es un dato de tipo **Conjunto** y a es un dato de tipo **TipoBase**, $a * A$ será un dato de tipo **Conjunto** y contendrá $\{a\} \cap A$
- ñ) Añadir un módulo de interface entre las clases **Conjunto** y **VectorDinámico** que ofrezca dos funciones, con los siguientes prototipos:

VectorDinamico ConjuntoToVectorDinamico (**Conjunto** & c);

Devuelve en un **VectorDinamico** los datos de un **Conjunto**

Conjunto VectorDinamicoToConjunto (**VectorDinamico** & v);

Devuelve en un **Conjunto** los datos de un **VectorDinamico**

- o) Sobreescibir los operadores $<<$ y $>>$ para leer/escribir un **Conjunto**.
- Para la implementación del operador $>>$ leerá una secuencia indefinida de valores, hasta que se introduzca el valor $*$. Los valores se leerán en una cadena de caracteres, y sólo se convertirán al tipo **TipoBase** cuando se verifique que son válidos para su almacenamiento (no se ha introducido el terminador $*$).
 - Evidentemente, **no se permiten elementos repetidos**.
8. Reescribir la solución al ejercicio 16 de la *Relación de Problemas II* (Aplicación de gestión de tareas y recursos) modularizando la solución usando clases.

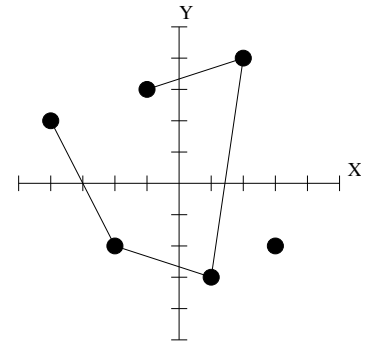
9. *(Examen Junio 2012)*

Para poder gestionar figuras planas en una aplicación de gráficos 2D deseamos crear una estructura de datos capaz de representar adecuadamente esas figuras. La estructura escogida será la de una línea poligonal construida en base a una serie de puntos. Dispondremos de las clases **Punto** y **PoliLinea**

A continuación mostramos la declaración de ambas clases (sólo la representación interna) y un ejemplo de un punto en las coordenadas $(3, -2)$ y de una polilínea definida por los puntos $(-4, 2)$, $(-2, -2)$, $(1, -3)$, $(2, 4)$ y $(-1, 3)$

```
class Punto {
    int x, y;      // Coordenadas de un punto 2D
    ....
};

class PoliLinea {
    Punto *p;      // Vector de puntos
    int num;       // Número de puntos
    ....
};
```



1. Métodos básicos

Implemente los siguientes métodos básicos para la clase `PoliLinea`:

- El constructor sin parámetros (crea una línea poligonal vacía) y el destructor.
- El constructor de copia y el operador de asignación.

2. Operadores varios

- Implemente el operador de acceso `[]`, que permite acceder (tanto para lectura como para escritura) a un dato de tipo `Punto` en una `PoliLinea`
- Implemente los operadores lógicos de igualdad `==` y desigualdad `!=`. Dos datos `PoliLinea` son iguales si tienen el mismo número de puntos, éstos son iguales y están dispuestos en el mismo orden o en orden inverso (en definitiva, son iguales cuando al representarse gráficamente se obtiene la misma figura).
- Sobrecargar el operador `+` para poder añadir un punto a una línea poligonal de manera que podamos ejecutar operaciones de la forma:
 - polilínea + punto. Se crea una **nueva** polilínea añadiendo un punto al final de la misma.
 - punto + polilínea. Se crea una **nueva** polilínea añadiendo un punto al inicio de la misma.

En ambos casos, la `PoliLinea` inicial **no** se modifica.

Si fuera preciso implementar algún método para la clase `Punto`, deberá hacerlo.

3. Métodos y funciones para E/S

Considere el siguiente formato que permite almacenar una `PoliLinea` en un fichero de texto:

RELACIÓN DE PROBLEMAS IV. Clases (II)

```
POLILINEA
# Comentario opcional
N
X1 Y1
X2 Y2
X3 Y3
...
Xn Yn
```

El fichero siempre comienza por la cadena POLILINEA
El comentario es opcional y, en caso de estar:
Comienza por el carácter #
Sólo ocupa una línea
No hay límite de caracteres
N es el número de puntos que tiene la polilínea
Después de N tenemos la lista de coordenadas de cada punto
Cada punto se escribe en una nueva línea y las coordenadas están separadas por un espacio

- a) Implementar un método para cargar en memoria una *PoliLinea* desde un fichero:

```
void LeerPolilinea (const char *nombre);
```

- b) Implementar un método para escribir en un fichero una *PoliLinea*:

```
void EscribirPolilinea (const char *nombre,
                        const char *comentario=0);
```

10. (*Examen Junio 2013*) Se desea resolver el problema de sumar un número indeterminado de enteros no negativos, con la dificultad añadida de que dichos números pueden llegar a ser muy largos, siendo imposible usar el tipo *int* del lenguaje. Para resolverlo, se propone crear una clase *BigInt* (entero largo) que puede almacenar un entero no negativo de longitud indeterminada.

La clase representará un entero mediante un array -de longitud variable- de objetos de tipo *int*, reservado en memoria dinámica, para poder almacenar todos y cada uno de los dígitos de un entero de longitud indeterminada. Tenga en cuenta que un objeto *int* puede almacenar un rango muy amplio de valores, sin embargo, por simplicidad, será el tipo que usaremos para almacenar cada dígito -del 0 al 9- del *BigInt*.

Además, los dígitos del *BigInt* se almacenarán de forma que el menos significativo -las unidades- se situará en la posición cero del array. Por ejemplo, a continuación se presentan dos ejemplos: los enteros largos 9530273759835 y 0. Observe que para el primer caso se ha reservado y usado un array de objetos *int* de longitud 13 y, para el cero, un array de longitud uno, con el dígito 0.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|--|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
| 5 | 3 | 8 | 9 | 5 | 7 | 3 | 7 | 2 | 0 | 3 | 5 | 9 | |

| |
|---|
| 0 |
| 0 |

Considerando este problema, **resuelva las siguientes preguntas:**

1. Métodos básicos

- a) Implemente el constructor sin parámetros y el destructor. Dicho constructor crea un entero largo con valor cero.

RELACIÓN DE PROBLEMAS IV. Clases (II)

- b) Implemente un constructor que crea un *BigInt* a partir de un objeto de tipo *unsigned int*. Para resolverlo, deberá:
 - i) Crear una función *Uint2Cadena* **recursiva** que transforma un objeto de tipo *unsigned int* a una cadena de caracteres (*cadena-C*) que representa el entero.
 - ii) Implementar el constructor llamando a la función *Uint2Cadena*, usando el resultado para inicializar el objeto *BigInt*.
- c) Implemente el constructor de copia y operador de asignación.

2. Sobrecarga de operadores

- a) Sobrecargue el operador de suma para que, a partir de dos objetos *BigInt*, se obtenga un nuevo objeto que corresponde al entero largo resultado de su suma.
- b) Sobrecargue el operador de resta para que, a partir de dos objetos *BigInt*, se obtenga un nuevo objeto que corresponde al entero largo resultado de su resta.
- c) Sobrecargue el operador de salida (operador <<) para la clase *BigInt* de forma que nos permite escribir el entero largo en un flujo de salida. Tenga en cuenta que deberá presentar todos los dígitos desde el más significativo al menos significativo (escritura habitual de enteros).
- d) Sobrecargue el operador de entrada (operador >>) para la clase *BigInt* de forma que nos permite leer el entero largo desde un flujo de entrada. Tenga en cuenta que deberá leer todos los dígitos desde el más significativo al menos significativo. El operador de entrada debe comportarse de forma similar al caso del tipo *int*, es decir, asumiendo que cada *BigInt* -secuencia de dígitos consecutivos- se encuentra separado de otros datos por “espacios blancos” (espacios, tabuladores, saltos de línea, etc.).

3. Aplicación

Supongamos que tenemos un archivo con un número indeterminado de enteros largos en formato texto, separados por “espacios blancos”. Implemente un programa “*suma.cpp*” que use la clase *BigInt* para leer todos los valores que hay en el fichero y escribir, como resultado final, la suma de todos ellos. Por ejemplo, si el archivo “*datos.txt*” contiene los siguientes enteros:

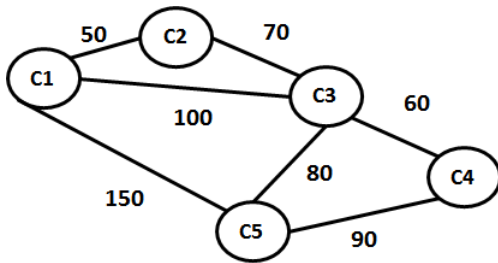
```
93478290374700000000000000000000
9327887198348931
```

Una posible ejecución del programa podría ser la siguiente:

```
% suma datos.txt
9347829037479327887198348931
```


RELACIÓN DE PROBLEMAS IV. Clases (II)

11. (*Simplificación del examen de Junio 2017*) Se desea construir la clase `RedCiudades` para almacenar datos de un conjunto de ciudades, y las distancias de los caminos directos que las conectan. Si entre dos ciudades no existe un camino directo, se almacenará un cero. Se supone que la distancia de una ciudad consigo misma será cero y que las distancias son simétricas. Un ejemplo con 5 ciudades sería:



| | C1 | C2 | C3 | C4 | C5 |
|----|-----|----|-----|----|-----|
| C1 | 0 | 50 | 100 | 0 | 150 |
| C2 | 50 | 0 | 70 | 0 | 0 |
| C3 | 100 | 70 | 0 | 60 | 80 |
| C4 | 0 | 0 | 60 | 0 | 90 |
| C5 | 150 | 0 | 80 | 90 | 0 |

Se propone la siguiente representación para la clase:

```
struct InfoCiudad {
    char * nombre; // Nombre
    int poblacion; // Num. habs.
};

class RedCiudades {
private:
    int num_ciudades; // Número de ciudades
    InfoCiudad * info; // info[i]: info de la ciudad i
    double ** distancia; // distancia[i][j]: distancia
                        // entre las ciudades i y j
public:
    // ... interfaz pública de la clase
};
```

1. Métodos básicos

Defina los siguientes métodos básicos para la clase `RedCiudades`:

- Constructor por defecto y destructor. El constructor por defecto crea una *red vacía* (escriba también el método `EstaVacía` que indique si una red está vacía).
- Constructor de copia y operador de asignación.

Escriba los siguientes métodos públicos de consulta:

- `NumCiudades`: devuelve el número de ciudades.
- `Distancia`: devuelve la distancia entre dos ciudades.
- `NombreCiudad`: devuelve el nombre de una ciudad (en realidad, la dirección de inicio).
- `PoblacionCiudad`: devuelve número de habitantes de una ciudad.

2. Métodos de cálculo

- a) Implemente el método `CiudadMejorConectada` que permita obtener la ciudad (su índice) con mayor número de conexiones directas.

En el ejemplo la ciudad mejor conectada sería la ciudad C3 con 4 conexiones.

- b) Implemente el método `MejorEscalaEntre` para que, dadas dos ciudades i y j no conectadas directamente, devuelva aquella ciudad intermedia z que permita hacer el trayecto entre i y j de la forma más económica posible. Es decir, se trata de encontrar una ciudad z tal que $d(i, z) + d(z, j)$ sea mínima ($d(a, b)$ es la distancia entre las ciudades a y b). El método devuelve -1 si no existe dicha ciudad intermedia.

Por ejemplo, si se desea viajar desde la ciudad C2 a la C5, hacerlo a través de la ciudad C1 tiene un costo de $50 + 150 = 200$ mientras que si se hace a través de la ciudad C3, el costo sería $70 + 80 = 150$.

3. Función de escritura

Escriba la función `MuestraRed` para mostrar en `cout` el contenido de una red.

Nota: No es un método de la clase.

4. Aplicación

Escriba un programa *completo* que cree redes de ciudades*.

Para cada red, el programa calculará, para todas las parejas de ciudades que *no* estén directamente conectadas, cuál es la mejor escala con una sola ciudad intermedia (su índice). Si no la tuviera, deberá indicarlo. También indicará si se trata de una red totalmente conectada.

* Para crear cómodamente una red con los conocimientos actuales -sin usar un constructor que lea el contenido de una red desde un fichero- recomendamos escribir un constructor que rellene los datos de una red *ad-hoc*, mediante sencillas instrucciones de asignación (y la correspondiente reserva de memoria previa, evidentemente).

RELACIÓN DE PROBLEMAS V. Gestión de E/S.

Los ejercicios propuestos en esta relación tiene como finalidad la práctica con flujos y operaciones sencillas de E/S. Se trabajará en todos los casos con la entrada/salida estándar usando los objetos `cin` y `cout`, por lo que podrá emplearse la redirección y/o encauzamiento. De hecho, recomendamos la ejecución de los programas usando redirección y/o encauzamiento, por comodidad.

Cuando en los enunciados de los problemas encuentre “lea una secuencia indefinida de ...de la entrada estándar” el programa debe usar `cin` para leer una secuencia de datos delimitada por el *fin del fichero*:

- Si los datos se introducen desde el teclado, debe finalizarse introduciendo manualmente el *fin del fichero* (Ctrl+D en GNU/Linux ó Ctl+Z en MS-DOS/Windows).
- Si se emplea la redirección de entrada, tomando los datos de un fichero de texto, el *fin de fichero* se inserta automáticamente en el flujo de entrada.

-
1. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y los copie literalmente en la salida estándar.
 2. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y los copie en la salida estándar, exceptuando las vocales.
 3. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y muestre en la salida estándar el número total de caracteres leídos.
 4. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y muestre en la salida estándar el número de líneas *no vacías* que hay en esa secuencia.

Nota: Se entenderá que una línea es *vacía* si contiene **únicamente** el carácter ‘`\n`’

5. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y muestre en la salida estándar únicamente las líneas *no vacías* que hay en esa secuencia.
6. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y “comprima” todas las líneas de esa secuencia, eliminando los separadores que hubiera en cada línea. Sólo se mantendrá el carácter ‘`\n`’
7. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y copie en la salida estándar las líneas que **no** comiencen por el carácter `#`

RELACIÓN DE PROBLEMAS V. Gestión de E/S.

8. Escribir un programa que lea una serie indefinida de números enteros de la entrada estándar y los copie, en el mismo orden, en la salida estándar.
 - En la secuencia de entrada se pueden usar espacios, tabuladores o saltos de líneas (en cualquier número y combinación) para separar dos números enteros consecutivos.
 - En la secuencia de salida se separan dos enteros consecutivos con un salto de línea.
9. Escribir un programa que lea una serie indefinida de números enteros de la entrada estándar y los copie, *en orden inverso*, en la salida estándar.
 - En la secuencia de entrada se pueden usar espacios, tabuladores o saltos de líneas (en cualquier número y combinación) para separar dos números enteros consecutivos.
 - En la secuencia de salida se separan dos enteros consecutivos con un salto de línea.
 - Usar un objeto `Pila` para invertir la secuencia.
10. Escribir un programa que lea una serie indefinida de números enteros de la entrada estándar y los copie, en el mismo orden, en la salida estándar.
 - En la secuencia de entrada los números están separados por el carácter `'*'`.
 - En la secuencia de salida se separan dos enteros consecutivos con un salto de línea.
11. Escribir un programa que lea un fichero como los generados en los problemas 8, 9 y 10 y que muestre en la salida estándar la suma de todos esos números.
12. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y reciba como argumento desde la línea de órdenes un dato de tipo `char`. El programa mostrará en la salida estándar el número de caracteres leídos de la entrada estándar iguales al argumento suministrado.

Por ejemplo:

```
CuentaCaracteresConcretos a < ElQuijote.txt  
mostrará el número de caracteres a que hay en ElQuijote.txt
```

13. Escribir un programa que lea una secuencia indefinida de caracteres de la entrada estándar y reciba como argumento desde la línea de órdenes un dato de tipo `int`.

El programa mostrará en la salida estándar el número de palabras leídas de la entrada estándar cuya longitud sea igual al argumento suministrado.

Por ejemplo:

```
CuentaPalabrasLongitudConcreta 10 < ElQuijote.txt  
mostrará el número de palabras en ElQuijote.txt que tienen 10 caracteres.
```

RELACIÓN DE PROBLEMAS VI. Ficheros (I)

1. Escribir un programa que reciba los nombres de dos ficheros de *texto* de la línea de órdenes. El programa creará un fichero (cuyo nombre se especifica en el segundo argumento) a partir de un fichero existente (cuyo nombre se especifica en el primer argumento) copiando su contenido y añadiendo al principio de cada línea, su número.
2. Escribir un programa similar a *diff* para comparar dos ficheros de texto. El programa imprimirá el número de la primera línea en la que difieren y el contenido de éstas.

Por ejemplo, la ejecución de

```
Diferencias Fich1 Fich2
```

producirá como resultado:

```
( 20) Fich1: formato binario. Estos ficheros son ...  
      Fich2: formato binario. Estos ficheros, aunque ...
```

si las 19 primeras líneas de *Fich1* y *Fich2* son idénticas, y la primera diferencia se encuentra en la línea 20.

Nota: Este programa puede ser útil para comprobar si después de encriptar y desencriptar un fichero (problema 15 de esta relación), obtenemos un fichero idéntico al original.

3. Escribir un programa que reciba como parámetros tres nombres de ficheros de *texto*. Los dos primeros ficheros contienen números reales y están *ordenados*. El programa tomará los datos de esos ficheros y los irá copiando ordenadamente en el tercer fichero, de forma que al finalizar esté también ordenado.
4. Se dispone de ficheros de *texto* que contienen un número indeterminado de líneas, cada una de ellas con los datos correspondientes a una serie de grupos de valores reales.
Escribir un programa que escriba en la salida estándar una línea de resultado por cada línea de entrada, y en cada línea mostrará las sumas de los valores de cada grupo que la componen.

Por ejemplo, una línea de entrada podría ser la siguiente:

```
3 2 3.1 0.4 5 1.0 1.0 1.0 1.0 1.0 2 5.2 4.7
```

donde puede observar que se distinguen tres grupos de datos (indicado por el primer número de la línea) y cada grupo empieza por un valor entero (2, 5 y 2) seguido por tantos valores reales como indique el valor entero que encabeza cada grupo:

| | | | | | | | | | | | | |
|---|---|-----|-----|---|-----|-----|-----|-----|-----|---|-----|-----|
| 3 | 2 | 3.1 | 0.4 | 5 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 2 | 5.2 | 4.7 |
|---|---|-----|-----|---|-----|-----|-----|-----|-----|---|-----|-----|

El programa deberá escribir:

```
3.5      5.0      9.9
```

RELACIÓN DE PROBLEMAS VI. Ficheros (I)

El programa se ejecutará desde la línea de órdenes y permitirá:

- a) Llamarlo sin ningún argumento. En este caso, los datos de entrada se leerán desde la entrada estándar.
- b) Llamarlo con un argumento. El argumento corresponde al nombre del archivo con las líneas de entrada.

Las restricciones que se imponen, y que se deben cumplir en la resolución son:

- a) El fichero sólo puede ser leído una vez, y no puede copiarse completo en memoria.
- b) Se desconoce *a priori* el número de líneas del fichero.
- c) Las líneas del fichero tiene una longitud *indeterminada*, aunque nunca mayor de 500.
- d) Pueden haber líneas vacías y líneas que contengan sólo espacios y otros separadores.

5. Construir un programa que divida un fichero de *texto* en diferentes ficheros indicando como argumentos el nombre del fichero original y el **máximo número de líneas** que contendrá cada fichero resultante.

Se creará un fichero de control que contendrá con los datos necesario para la reconstrucción del fichero original.

Por ejemplo, si `Fichero` contiene 1600 líneas, la ejecución de

`ParteFicheroPorNumLineas Fichero 500`

genera como resultado los ficheros `Fichero_1`, `Fichero_2`, `Fichero_3` y `Fichero_4`.

Los tres primeros contienen 500 líneas de `Fichero` y el último, las 100 restantes.

Se creará un fichero *oculto* llamado `.Fichero.ctrl` que contendrá (formato texto, en dos líneas separadas): nombre del fichero original y número de ficheros resultantes de la partición.

6. Ampliar la clase `Lista` con tres métodos que usan un fichero de texto en el que los datos (de tipo `TipoBase`) están separados por caracteres *separadores*.

- `Lista (const char * nombre);`

Constructor que recibe el nombre de un fichero de texto y rellena los nodos de la lista con los datos contenidos en el fichero.

- `void EscribirLista (const char * nombre) const;`

Guarda en el fichero de texto `nombre` el contenido de la lista. Escribe un dato por línea. Si el fichero ya existiera, se reemplaza su contenido por el de la lista. La lista no se modifica.

- `void LeerLista (const char * nombre);`

Sustituye el contenido de la lista por los valores que están en el fichero de texto `nombre`.

RELACIÓN DE PROBLEMAS VI. Ficheros (I)

7. Ampliar la clase `Matriz2D-1` con tres métodos similares a los del ejercicio 6.

Ahora, los dos primeros datos del fichero son dos datos `int` que indican el número de filas y columnas de la matriz. Los datos que se guardarán en la matriz son de tipo `TipoBase` y están separados en el fichero por caracteres *separadores*.

- `Matriz2D_1 (const char * nombre);`

Constructor que recibe el nombre de un fichero de texto y rellena las casillas de la matriz con los datos contenidos en el fichero.

- `void EscribirMatriz2D_1 (const char * nombre) const;`

Guarda en el fichero de texto `nombre` el contenido de la matriz. Si el fichero ya existiera, se reemplaza su contenido por el de la matriz. La matriz no se modifica.

- `void LeerMatriz2D_1 (const char * nombre);`

Sustituye el contenido de la matriz por los valores que están en el fichero de texto `nombre`.

8. Escribir otra versión de los métodos desarrollados en el ejercicio 7. En este caso, el fichero de texto que guarda los datos de la matriz es distinto:

No aparece el número de filas y columnas, y los datos de tipo `TipoBase` se disponen en líneas (todas las líneas tienen el mismo número de elementos) Como antes, los datos están separados por caracteres *separadores*.

El número de líneas de datos del fichero debe coincidir con el número de filas de la matriz, y el número de elementos de cada fila con el número de columnas.

Las restricciones que se imponen, y que se deben cumplir en la resolución son:

- a) El fichero sólo puede ser leído una vez, y no puede copiarse completo en memoria.
- b) Se desconoce *a priori* el número de líneas del fichero.
- c) Las líneas del fichero tiene una longitud *indeterminada*, aunque nunca mayor de 500.
- d) Pueden haber líneas vacías y líneas que contengan sólo espacios y otros separadores.
- e) El número de datos de cada línea es *indeterminado*, pero *común* a todas las líneas.
- f) No puede emplearse una matriz con un número de filas “tentativo”: la matriz ocupará en cada momento el espacio estrictamente necesario y los datos se copiarán conforme se lean cada una de las filas.

Nota: Es posible que sea necesario añadir un nuevo método a la clase `Matriz2D-1` que permita redimensionar la matriz, añadiendo una nueva fila.

Recomendación: Usar flujos asociados a `string` para procesar las líneas del fichero.

9. (*Examen de Junio 2017 - parte II*) En el ejercicio 11 de la *Relación de Problemas IV* se presentó una versión muy simple de la clase `RedCiudades` que se va a ampliar en este ejercicio.

1. Métodos de E/S

Se desea ampliar la clase con los siguientes métodos para E/S:

- `RedCiudades(const char * nombre);`

Constructor. Recibe el nombre de un fichero de texto con los datos de una red. Contiene en la primera línea la *cadena mágica* "RED" y un salto de línea. A continuación está el contenido de la red en formato de texto, tal como se indica a continuación.

- a) un número entero que indica el número de ciudades y un salto de línea,
- b) una serie de líneas (tantas como ciudades) que contienen (cada una) el índice de la ciudad -un número entero-, su nombre -una serie de caracteres sin espacios intermedios- y su población -un número entero-, y
- c) una serie de líneas (tantas como conexiones entre ciudades) que contienen dos números enteros (números de las ciudades conectadas) y un número real (distancia entre ellas).

Nota: Cada conexión entre ciudades se escribe una sola vez (no importa cuál es la ciudad de origen o destino).

- `void EscribirRedCiudades (const char * nombre) const;`

Guarda en el fichero de texto `nombre`, con el formato conocido, el contenido de la red. Si el fichero ya existiera, se reemplaza su contenido por el de la red. La red no se modifica.

- `void LeerRedCiudades (const char * nombre);`

Permite actualizar el contenido de una red con los datos de un fichero de texto con el formato conocido. El contenido de la red será sustituido.

2. Aplicación

Escriba un programa *completo* que reciba desde la línea de órdenes el nombre de un fichero de texto que contiene la descripción de una red y calcule, para todas las parejas de ciudades que *no* estén directamente conectadas, cuál es la mejor escala con una sola ciudad intermedia (su índice). Si no la tuviera, deberá indicarlo.

Deberá indicar si se trata de una red totalmente conectada.

10. A partir del contenido de un fichero de texto pueden encontrarse “mensajes ocultos” en él, seleccionando determinadas palabras del mismo. Bastaría con indicar en una sucesión de números, por ejemplo, las palabras del texto original (en realidad su número de orden) que hay que seleccionar.

Escriba un programa que reciba como argumentos los nombres de dos ficheros de texto: el primero es un texto “convencional” y el segundo contiene números enteros, sin ningún formato

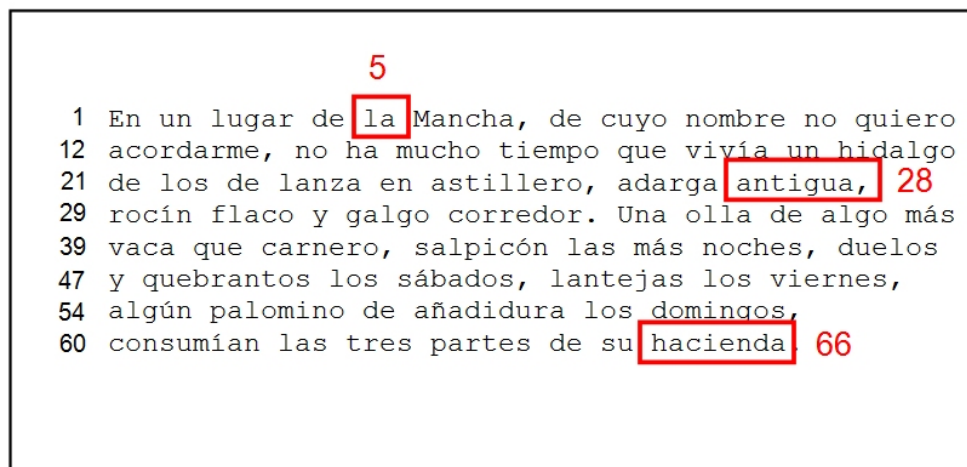
RELACIÓN DE PROBLEMAS VI. Ficheros (I)

establecido. Cada uno de estos números indica qué palabra hay que seleccionar del fichero original, y el orden de los números marca el orden de las palabras en el mensaje final. El programa mostrará en la salida estándar el mensaje oculto.

En la figura 31 indicamos los números de orden de las palabras (a la izquierda mostramos el número de orden de la primera palabra de cada línea). Si el contenido del fichero de claves fuera, por ejemplo,

5 66 28

el mensaje oculto será: `la hacienda antigua`



```

      5
1  En un lugar de la Mancha, de cuyo nombre no quiero
12 acordarme, no ha mucho tiempo que vivía un hidalgo
21 de los de lanza en astillero, adarga antigua, 28
29 rocín flaco y galgo corredor. Una olla de algo más
39 vaca que carnero, salpicón las más noches, duelos
47 y quebrantos los sábados, lantejas los viernes,
54 algún palomino de añadidura los domingos,
60 consumían las tres partes de su hacienda 66
```

Figura 31: Mensajes ocultos (1)

11. Escriba una variante del programa escrito para el ejercicio 10 en el que el segundo fichero contiene ahora dos números para cada palabra: el número de línea y el número de palabra dentro de cada línea.

En la figura 32 indicamos los números de línea (a la izquierda) y la posición de las palabras seleccionadas en la línea. Si el contenido del fichero de claves fuera, por ejemplo,

1 5 8 7 3 8

el mensaje oculto será: `la hacienda antigua`

12. Escribir un programa similar a `grep` que busque una cadena en una serie de ficheros de texto. La cadena a buscar y los ficheros en los que buscar se proporcionan en la línea de órdenes. Cada vez que encuentre la cadena buscada, debe indicar el fichero en el que es localizada, el número de línea y la línea completa que la contiene.

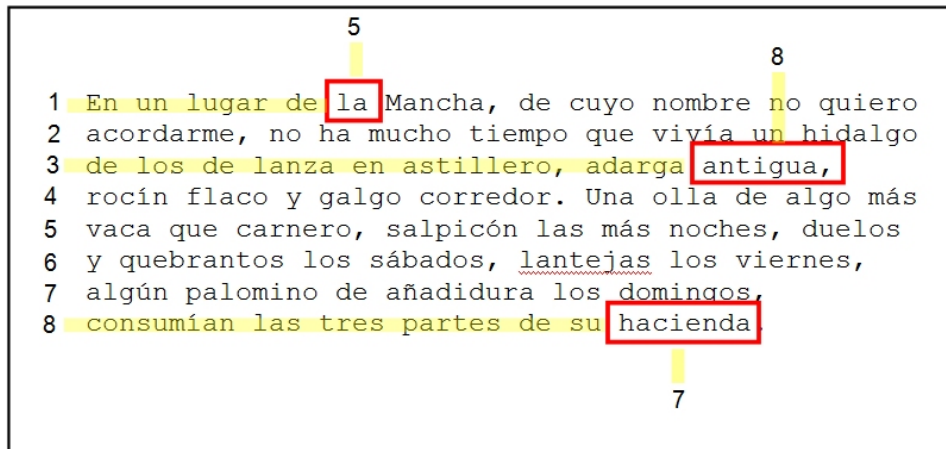


Figura 32: Mensajes ocultos (2)

Por ejemplo:

`Busca Olga fich1 fich2 fich3`

busca la cadena Olga en los ficheros fich1, fich2 y fich3.

Un resultado posible sería:

fich1 (línea 33): Mi amiga Olga ha aprobado MP aunque no

fich3 (línea 2): ya se lo dije ayer a Olga, pero ni caso

fich3 (línea 242): finalmente, Olga se puso a estudiar

Las restricciones que se imponen, y que se deben cumplir en la resolución son:

- a) El número de ficheros que se pueden proporcionar es *ilimitado*.
 - b) Cada uno de los ficheros sólo puede ser leído una única vez, y no pueden copiarse completos en memoria.
 - c) Se desconoce *a priori* el número de líneas de los ficheros.
 - d) Las líneas del fichero tiene una longitud *indeterminada*, aunque nunca mayor de 500.
13. Implementar un programa que similar a `head` que muestre las primeras líneas de un fichero de texto, tal como se encuentran en el fichero.

Por ejemplo, la ejecución de

`Cabecera 15 reconstruye.cpp`

mostrará las primeras 15 líneas del fichero de texto `reconstruye.cpp`

Se aplican las mismas restricciones que las indicadas en el problema 12 (excepto la primera, evidentemente).

RELACIÓN DE PROBLEMAS VI. Ficheros (I)

14. Implementar un programa que similar a `tail` que muestre las últimas líneas de un fichero de texto, tal como se encuentran en el fichero.

Por ejemplo, la ejecución de

```
Final 15 reconstruye.cpp
```

mostrará las últimas 15 líneas del fichero de texto `reconstruye.cpp`

Se aplican las mismas restricciones que las indicadas en el problema 12 (excepto la primera, evidentemente).

15. Escribir un programa que permita encriptar y desencriptar el contenido de un fichero de texto. Para *encriptar* sustituiremos cada letra (mayúsculas y minúsculas) por la letra que está p posiciones más adelante en el alfabeto (para las últimas letras ciclamos el alfabeto). Los caracteres que no sean letras se quedarán igual. Para *desencriptar* la sustitución será a la inversa. La llamada al programa se realizará con este esquema:

```
Codifica <ficheroE> <ficheroS> <p> <tipo>
```

donde:

- `<ficheroE>` y `<ficheroS>` son los nombres de los ficheros de entrada y salida, respectivamente
 - `<p>` es el número entero positivo que se aplica para codificar/descodificar cada uno de los caracteres.
 - `<tipo>` es una cadena de caracteres que puede valer: `enc` para encriptar y `desenc` para desencriptar.
16. Escriba un programa que reciba un número indeterminado de ficheros con extensión `.cpp` ó `.h` y genere nuevos ficheros eliminando todos los comentarios de los primeros.
- El nombre de los nuevos ficheros se forma añadiendo la palabra `_UNCOMMENT` antes de la extensión `.cpp` ó `.h`. Por ejemplo, a partir del fichero `Codifica.cpp` se genera `Codifica_UNCOMMENT.cpp`
17. **PGM** es el acrónimo de *Portable Gray Map file format*. Una imagen PGM almacena una imagen de niveles de gris, es decir, un valor numérico que indica la luminosidad para cada píxel. Un valor de luminosidad viene determinado por un número entero en el rango `[0, 255]`. Por lo tanto, un único byte será suficiente para almacenar el contenido de un píxel.
- PGM es un buen ejemplo de formato de almacenamiento de imágenes con cabecera. Podemos considerar dos partes en un fichero PGM:
- a) La **cabecera**. En esta parte se almacena la información sobre el tipo de imagen, dimensiones, etc. Esta información está en formato *texto*. Está compuesta de:
 - Una *cadena mágica* que identifica el tipo de fichero. En nuestro caso la cadena `P5` (imagen de niveles de gris con datos codificados en binario).

RELACIÓN DE PROBLEMAS VI. Ficheros (I)

- Un separador (salto de línea).
 - Un número indeterminado de comentarios. Los comentarios son de línea completa y están precedidos por el carácter #. La longitud máxima es de 70 caracteres.
 - El ancho o número de columnas (c) de la imagen: número de píxeles de una fila.
 - Un separador.
 - El alto o filas (f) de la imagen: número de píxeles de una columna.
 - Un separador.
 - El valor máximo (M) de gris. Indica que el rango de valores para cada valor de luminosidad es $[0, M]$ donde 0 correspondiente a negro y M a blanco.
 - Un separador (uno sólo, antes de la secuencia).
- b) El **contenido de la imagen**. Esta información está en formato *binario*. Corresponde a una secuencia de $f \times c$ valores de luminosidad (uno por pixel). El primero corresponde al píxel de la esquina superior izquierda, el segundo al de su derecha, etc. (el último byte corresponde al píxel de la esquina inferior derecha de la imagen): almacenamiento *por filas*.
- Si se emplea un byte (8 bits) por pixel disponemos de $2^8 = 256$ valores de luminosidad por pixel donde 0 corresponde al negro y 255 al blanco.

Un ejemplo de cabecera PGM es el siguiente:

```
P5
# Creador: Profesores de MP
# Una imagen de prueba en niveles de gris
# con 600 filas y 800 columnas
800 600
255
```

Inmediatamente detrás del separador que sigue al último valor de la cabecera (255 en el ejemplo) encontraremos una secuencia de 800×600 bytes.

Escriba un programa que reciba el nombre de un fichero PGM y muestre la información de su cabecera.

RELACIÓN DE PROBLEMAS VII. Ficheros (II)

1. Escribir un programa que reciba el nombre de dos ficheros. El programa copiará, en el mismo orden, los números que contiene el fichero de entrada en el fichero de salida.
 - El primer fichero (entrada) contiene una serie indefinida de números enteros. Es un fichero de *texto* y puede contener *espacios*, *tabuladores* o *saltos de línea* (en cualquier número y combinación) separando dos números enteros consecutivos.
 - El segundo fichero (salida) es un fichero *binario*.
 - El programa leerá los números y los copiará **de uno en uno**.
2. Escribir un programa con las mismas características que las descritas en el problema 1 pero que escriba en el fichero de salida **bloques de 512 bytes**.
3. Escribir un programa que lea un fichero *binario* como los generados en los problemas 1 y 2 y que muestre en la salida estándar la suma de todos esos números. Para la lectura se empleará un **buffer de 512 bytes**.

Escribir dos soluciones:

- a) Usando un vector de 512 bytes como buffer, pero declarado como un array de `int`.
 - b) Usando un vector de 512 bytes como buffer y declarado como tal, como un array de `unsigned char`.
4. Escriba dos programas para transformar ficheros con datos correspondientes a una serie de grupos de valores reales (como están descritos en el problema 4 de la **Relación de Problemas VI**), para transformar entre formato binario y texto:
 - a) Un programa que transforme un fichero de texto a binario:

```
Text2bin <FichText> <FichBin>
```

- b) Un programa que transforme un fichero de binario a texto:

```
Bin2text <FichBin> <FichText>
```

Debe optimizarse el uso de los recursos, y por tanto, se aplican las restricciones enumeradas en el problema 4 de la **Relación de Problemas VI**.

5. Construir un programa que divida un fichero de *cualquier tipo* en diferentes ficheros, indicando como argumentos el nombre del fichero original y el **máximo número de bytes** que contendrá cada fichero resultante.

Por ejemplo, si el tamaño de `Fichero` es 1800 bytes, la ejecución de

```
ParteFicheroPorNumBytes Fichero 500
```

genera los ficheros `Fichero_1`, `Fichero_2`, `Fichero_3` y `Fichero_4`.

RELACIÓN DE PROBLEMAS VII. Ficheros (II)

Los tres primeros contienen 500 bytes de `Fichero` y el último, los 300 restantes. Se creará un fichero *oculto* llamado `.Fichero.ctrl` que contendrá (formato texto, en dos líneas separadas): nombre del fichero original y número de ficheros resultantes de la partición.

Recomendación: La construcción de los nombres de los ficheros que contienen las partes (`Fichero_001`, `Fichero_002`, ...) es una tarea sencilla si se emplea un flujo asociado a una cadena. En este caso, se tratará de un objeto `ostringstream`.

6. Construir un programa que reconstruya un fichero a partir de una serie de ficheros que contienen sus “partes”. Los ficheros que pueden emplearse como origen se han creado con los programas descritos en los problemas 5 (Relación de Problemas VI) y 5 (Relación de Problemas VII) y por ese motivo se empleará el fichero de control creado por esos programas.

Por ejemplo, la ejecución de `reconstruye Fichero` genera como resultado `Fichero`. Usará `.Fichero.ctrl` para conocer los ficheros que debe usar y el orden a seguir.

Recomendación: La construcción de los nombres de los ficheros que contienen las partes (`Fichero_001`, `Fichero_002`, ...) es una tarea sencilla si se emplea un flujo asociado a una cadena. En este caso, se tratará de un objeto `ostringstream`.

7. Una empresa de distribución mantiene la información acerca de sus vendedores, productos y ventas en ficheros. La información se almacena en formato *binario* y la estructura es:

- Fichero Vendedores:

`RegVendedor`: `CodVendedor` (unsigned char), `Nombre` (50*char) y `CodZona` (unsigned char).

- Fichero Artículos:

`RegArticulo`: `CodArticulo` (10*char), `Descripcion` (30*char) y `PVP` (float).

- Fichero Ventas:

`RegVenta`: `NumFactura` (int), `CodVendedor` (unsigned char), `CodArticulo` (10*char) y `Unidades` (int).

Se supone que el fichero `Ventas` contiene las ventas realizadas en un mes.

Se trata de realizar programas para:

- a) Mostrar el total (número de ventas y cantidad total de ventas) de las ventas realizadas por un vendedor, dado su código (`CodVendedor`).
- b) Pedir una cantidad y crear un fichero (*binario*) llamado `VendedoresVIP` cuyos registros tengan la siguiente estructura:

`RegVendedorVIP`: `CodVendedor` (unsigned char), `CodZona` (unsigned char), `TotalVentas` (float).

donde `TotalVentas` es la cantidad total de ventas realizadas por el vendedor.

El fichero `VendedoresVIP` tendrá únicamente los registros de los vendedores cuyo total de ventas sea superior a la cantidad leída.

RELACIÓN DE PROBLEMAS VII. Ficheros (II)

8. Disponemos de un fichero binario (Alumnos) que almacena registros de datos de tipo `RegAlumno` con información sobre los alumnos de una clase. El tipo `RegAlumno` está declarado en `Alumnos.h` de esta manera:

```
#define MAX_DNI 15
#define MAX_CAD_NOMBRE 50
#define MAX_CAD_APELLIDO 50

struct RegAlumno{
    char DNI[MAX_DNI];
    char Nombre[MAX_CAD_NOMBRE];
    char Apellidos[MAX_CAD_APELLIDO];
    double Nota_EC;
    double Nota_EP1;
    double Nota_EP2;
    double Nota_EXAMEN;
};
```

Escriba un programa que ordene el contenido del fichero, generando un nuevo fichero que contiene los datos ordenados del fichero original.

La ordenación será creciente. Use el campo DNI como clave de ordenación. Deberá, no obstante, modularizar adecuadamente para poder usar otro campo como clave de ordenación sin tener que aplicar demasiados cambios.

El programa se ejecutará:

```
OrdenaAlumnos <fich_alumnos> <fich_ordenado> [<metodo>]
```

donde:

- `<fich_alumnos>` es el nombre del fichero de entrada (desordenado, binario).
- `<fich_ordenado>` es el nombre del fichero de salida (ordenado, binario).
- `<metodo>` es el método de ordenación. Es opcional (por defecto: *ordenación por selección*) pero en caso de especificarse se hará escribiendo `seleccion`, `insercion` ó `intercambio`.

Recomendamos implementar las funciones:

```
RegAlumno ObtenerRegistro (int pos, fstream & f);
```

Devuelve el registro de la posición `pos` del fichero de alumnos asociado al flujo `f`.

```
void ModificarRegistro (int pos, fstream & f, RegAlumno reg);
```

Sustituye el registro de la posición `pos` del fichero de alumnos asociado al flujo `f` por el dado en `reg`.

En ambos casos, `pos` indica el *número de registro* dentro del fichero de alumnos.

RELACIÓN DE PROBLEMAS VII. Ficheros (II)

9. Algunos ficheros se identifican mediante una “cabecera” especial, como los archivos PGM que tienen en la posición cero los caracteres P5. Este tipo de ficheros, y las marcas que los identifican son los que se gestionarán en este problema.

Un fichero de descripciones es un fichero *binario* que almacena las distintas *marcas* que identifican a tipos de ficheros, así como información acerca de dónde se ubican en los archivos.

Los registros del fichero *Descripciones* tienen longitud variable y su formato es:

RegDescripcion: PosInicioMarca (int), LongMarca (int), Marca (LongMarca*char) y Comentario (100*char)

Un ejemplo (en forma tabular) de los contenidos de este archivo podría ser:

| | | | |
|------|---|------|-------------------------|
| 0 | 2 | P5 | Imagen PGM |
| 0 | 2 | P6 | Imagen PPM |
| -128 | 3 | TAG | Fichero MP3 con ID3 TAG |
| 10 | 4 | DATA | Datos |

Donde podemos ver que si un archivo tiene la cadena “DATOS” (4 caracteres) a partir de la posición 10 del archivo, se trata de un archivo de tipo “Datos”. En el caso de el fichero MP3 con información ID3 TAG observe que la marca aparece 128 bytes *antes* del final del fichero.

Las tareas a realizar son:

- a) Escribir una función (**Insertar**) para añadir descripciones.

La función recibirá el nombre de un archivo de descripciones junto con una nueva entrada (un dato de tipo **RegDescripcion**) y añadirá esa entrada a dicho archivo (creándolo si es necesario).

- b) Implementar una función (**TipoArchivo**) que determine el tipo de un fichero.

La función recibirá el nombre del fichero del que queremos averiguar si tipo junto con el nombre del archivo de descripciones. Como resultado devuelve una cadena con el comentario asociado, o la cadena *Tipo desconocido* si no se ha localizado su tipo.

- c) Escribir un programa que, usando la función del apartado anterior, reciba de la línea de órdenes el nombre de un archivo y escriba en la salida estándar la descripción (*comentario*) asociado a su tipo. Tenga en cuenta los posibles casos de error.

10. En el ejercicio 17 de la *Relación de Problemas VI* trabajamos sobre la cabecera de un fichero PGM. En este ejercicio vamos también a trabajar sobre el contenido de la imagen.

En una imagen digital de niveles de gris los valores de luminosidad pueden normalizarse usando valores enteros que van desde el 0.0 (oscuridad total o negro) hasta el 1.0 (luminosidad total o blanco).

En la práctica se discretizan los valores que puede tomar el nivel de luminosidad y se escalan entre el 0 (negro) y el 255 (blanco) para poder representar cada pixel de la imagen con 1 byte (1 byte = 8 bits. Con 8 bits podemos representar $2^8 = 256$ valores diferentes. En la figura 33.A mostramos una imagen digital de niveles de gris que tiene 256 filas y 256 columnas. Cada uno de los $256 \times 256 = 65536$ puntos contiene un valor entre 0 (visualizado en negro) y 255

(visualizado en blanco). En la figura 33.B mostramos una parte de la imagen anterior (10 filas y 10 columnas) en la que se pueden apreciar, con más detalle, los valores de luminosidad en esa subimagen.

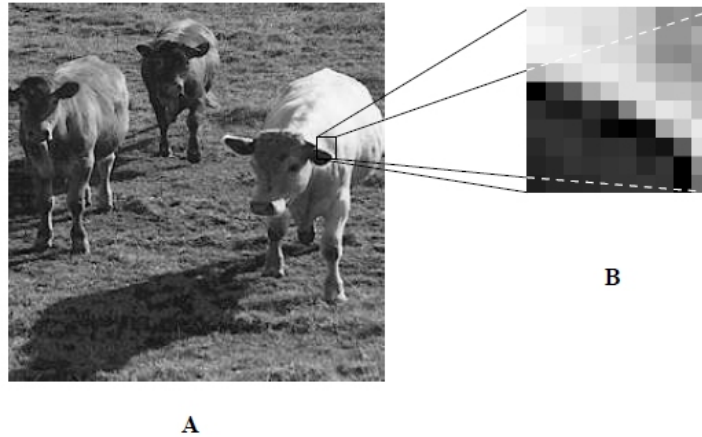


Figura 33: A. Imagen de 256 filas y 256 columnas. B) Una subimagen de tamaño 10×10

En un fichero PGM de niveles de gris, a continuación de la cabecera se almacenan los valores de luminosidad de los píxeles de la imagen. Como cada pixel necesita un byte, deberá haber exactamente $c \times f$ bytes si c es el número de columnas y f es el número de filas de la imagen.

Escriba un programa que reciba el nombre de un fichero PGM y nos indique si es correcto, esto es, si tras la cabecera hay exactamente tantos bytes como píxeles tiene la imagen.

11. La operación de **binarización** sobre una imagen consiste en transformar los valores de luminosidad para generar una nueva imagen en *blanco y negro* (literalmente) con las mismas dimensiones

Se establece un valor *umbral* de manera que si un pixel contiene un valor mayor o igual que el umbral se cambia por 255 y si no es así, por 0.

Escriba un programa que reciba el nombre de un fichero de PGM (imagen a transformar), el valor umbral y el nombre del fichero PGM resultante.

Por ejemplo, la ejecución de

```
Binariza vacas.pgm 100 vacas_binarizada_100.pgm
```

transforma la imagen guardada en `vacas.pgm` aplicando sobre ella la operación de binarización y generando el fichero `vacas_binarizada_100.pgm`

En la figura 34 pueden ver un ejemplo de esta operación.

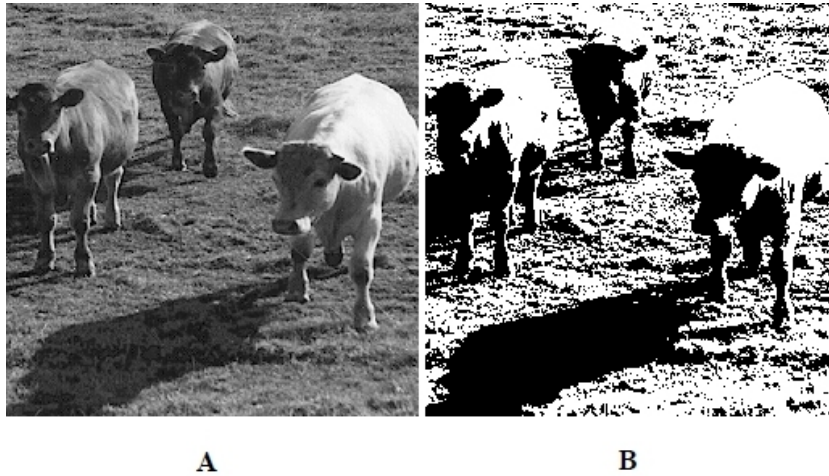


Figura 34: A) Imagen original con 256 niveles de gris. B) Imagen binarizada (únicamente 2 niveles de gris) con `umbral=100`

12. Escriba un programa para ampliar el contraste en una imagen PGM de niveles de gris.
Por ejemplo, la ejecución de

```
AumentaContraste saturno_bajo_contraste.pgm  
                  saturno_alto_contraste.pgm
```

transforma la imagen guardada en `saturno_bajo_contraste.pgm` aumentando el contraste (ampliando linealmente el rango del histograma entre 0 y 255) y generando el fichero `saturno_alto_contraste.pgm`.

En la figura 35 pueden ver un ejemplo de esta operación.

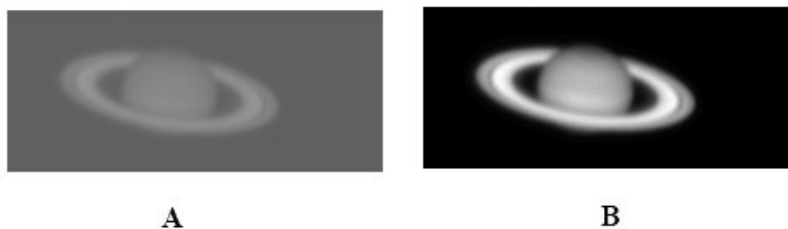


Figura 35: A) Imagen original. B) Imagen después de ampliar linealmente el contraste

RELACIÓN DE PROBLEMAS VII. Ficheros (II)

13. (*Examen de Junio 2017 - parte III*) En el ejercicio 11 de la *Relación de Problemas IV* se presentó una versión muy simple de la clase `RedCiudades` y en el ejercicio 9 se mejoró añadiendo un constructor y métodos para leer/guardar el contenido de un objeto de/en un fichero.

En este ejercicio deberá implementar una sobrecarga de los operadores de inserción en flujo << y de extracción de flujo >>.

- a) Sobrecargue el operador de salida << para poder insertar en un flujo de salida el contenido de una red en formato texto. En concreto, deberá aparecer:
- i) un número entero que indica el número de ciudades y un salto de línea,
 - ii) una serie de líneas (tantas como ciudades) que contienen el índice de la ciudad (un número entero), su nombre (una serie de caracteres sin espacios intermedios) y su población (un número entero),
 - iii) una serie de líneas (tantas como conexiones entre ciudades) que contienen dos números enteros (números de las ciudades conectadas) y un número real (distancia entre ellas).
- Nota:** Cada conexión entre ciudades se escribe una sólo vez (no importa cuál es la ciudad de origen o destino).
- b) Sobrecargue el operador de entrada >> para poder obtener desde un flujo de entrada el contenido de un objeto. Se asume el mismo formato indicado en el apartado a.

Reescriba los métodos que son precisos para hacer uso de estos nuevos operadores.

14. Considerar la clase `ColeccionPuntos2D` que se usará para almacenar y gestionar una colección de datos de tipo `Punto2D`. Considerar también la clase `Circunferencia`.
- Un objeto de la clase `ColeccionPuntos2D` es un vector dinámico de objetos de la clase `Punto2D`.
 - Un objeto de la clase `Circunferencia` está caracterizado por el valor del radio y el centro, que es un objeto de la clase `Punto2D`.

Realizar un programa que sea capaz de calcular qué puntos (pertenecientes a una colección de puntos) están dentro del círculo delimitado por una circunferencia.

El programa construirá una colección de puntos a partir de los datos descritos en un fichero de texto cuyo nombre se proporciona como argumento al programa. El formato del fichero de puntos se explica en la figura 36.

A continuación lee del teclado un número indeterminado de datos de tipo `Punto2D` de manera que se termina la lectura si el usuario introduce la marca *eof* durante la lectura. Los puntos leídos los añade al objeto `ColeccionPuntos2D` creado a partir del fichero.

A continuación lee del teclado los datos necesarios para crear una un objeto `Circunferencia`.

Finalmente muestra cuáles de los puntos almacenados en la colección `ColeccionPuntos2D` está dentro del círculo delimitado por la circunferencia.

```
PUNTOS
# Fichero de prueba
# Contiene 8 puntos

10.0 10.0
11.33 12.555
      9.77 12.66
0.0      -99.88
2.2 3.3
8.8      9.9
12.5 12.5
2.0 1.0
```

(a) Fichero

La primera línea del fichero contendrá únicamente **PUNTOS**. Pueden seguir un número indeterminado de *comentarios*. Cada línea de comentario empieza con el carácter **#**

Después se encuentran las coordenadas de los puntos. Son *números reales* delimitados por caracteres separadores (no hay separación explícita entre las parejas de números, o sea, se trata de un fichero de texto con números reales sin ningún formato establecido).

(b) Descripción

Figura 36: Formato de fichero de puntos