UNIVERSIDAD
DE GRANADA

METODOLOGÍA DE LA PROGRAMACIÓN

⇜⇝

GUIÓN DE PRÁCTICAS

⇜⇝

Grado en Ingeniería Informática
GRUPO A

Francisco José Cortijo Bon
cb@decsai.ugr.es

2019/2020

Departamento de
Ciencias de la Computación
e Inteligencia Artificial

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

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

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

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

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

Índice del Guión de Prácticas

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 tambien está contemplada en el esquema.

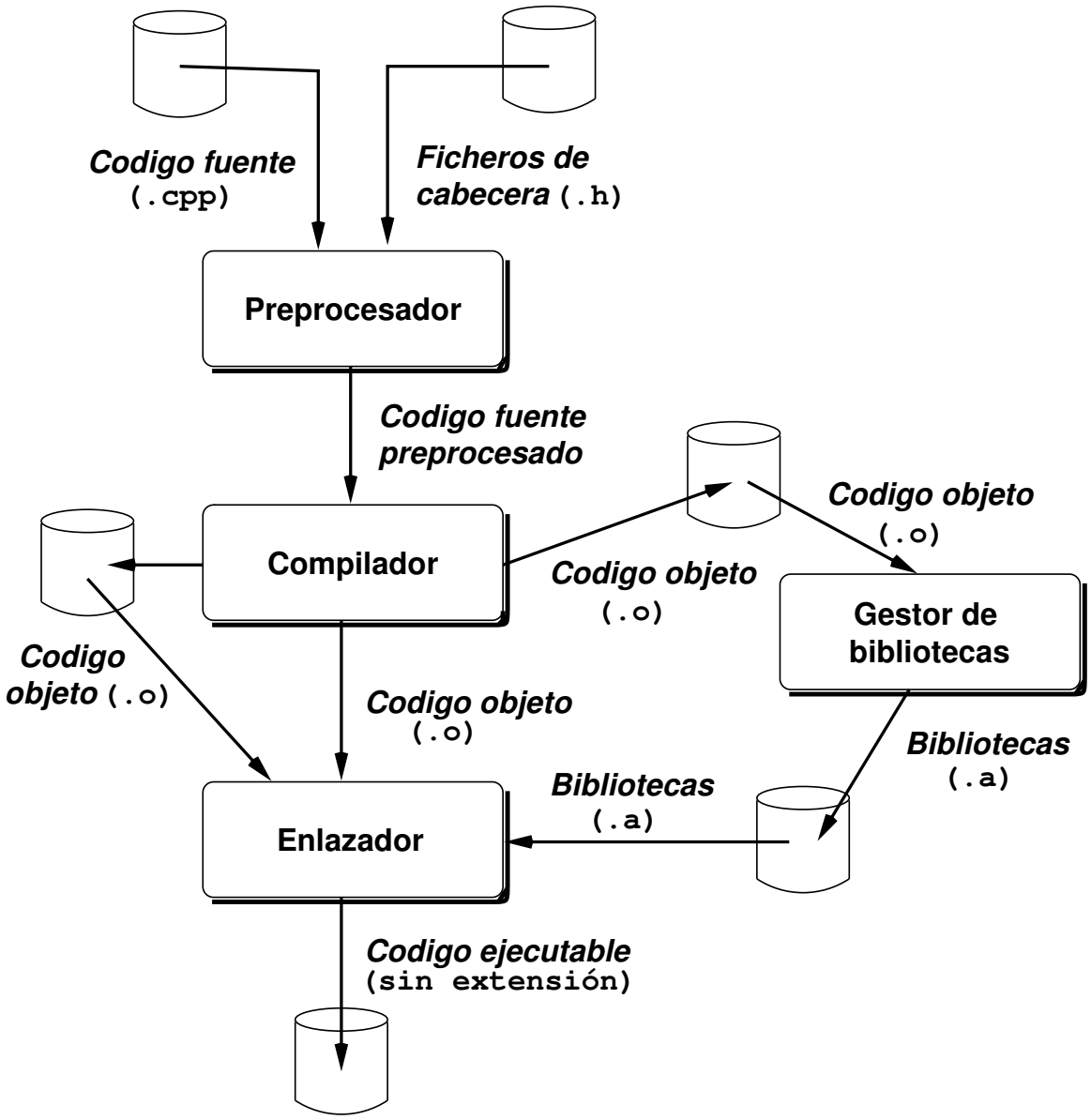


Figura 1: El proceso de compilación (generación de programas ejecutables) en C++

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 puden distinguirse las fases de preprocesado y compilación, en la práctica el preprocesado se considera como la primera fase de la compilación. Gráficamente, en la figura 2 mostramos el esquema detallado de lo que se conoce comúnmente por compilación.

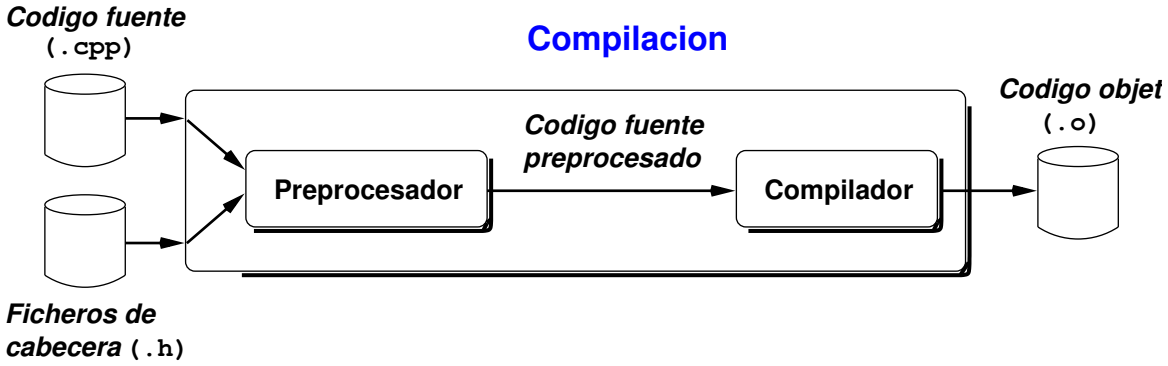


Figura 2: Fase de compilación

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 elementos externos al archivo. Las referencias externas se refieren a variables y principalmente a *funciones* que, aunque se utilizan en

el archivo -y por tanto deben estar declaradas en él- no se encuentran definidas en éste, sino en otro archivo distinto. La declaración servirá al compilador para comprobar que las referencias externas son sintácticamente correctas.

El enlazador

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

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

Bibliotecas. El gestor de bibliotecas

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

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

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

Cualquier programador puede desarrollar sus propias bibliotecas de funciones y enriquecer de esta manera el lenguaje. En la figura 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*** (**G**NU is **N**ot **U**nix). Puesto que en inglés "gnu" (en español "ñu") se pronuncia igual que "new", Richard Stallman recomienda pronunciarlo "guh-noo". En español, se recomienda pronunciarlo ñu como el antílope africano o fonéticamente; por ello, el término mayoritariamente se deletrea (G-N-U) para su mejor comprensión. En sus charlas Richard Stallman finalmente dice siempre «Se puede pronunciar de cualquier forma, la única pronunciación errónea es decirle 'linux'».

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

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

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

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

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

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

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

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

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

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

Otros programas del Proyecto GNU relacionados con nuestra materia son:

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

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

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

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

Sintaxis

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

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

donde:

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

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

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

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

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

Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

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

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

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

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

Ejercicio

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

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

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

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

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

Ejercicio

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

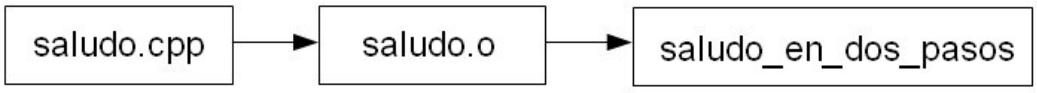


Figura 4: Diagrama de dependencias para `saludo_en_dos_pasos`

`-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 `lib`*fichero*`.a` y lo usa para enlazarlo.

Ejercicio

Ejecutar la siguiente orden:

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

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

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

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

Resumen: un ejemplo completo

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Introducción al depurador DDD

Conceptos básicos

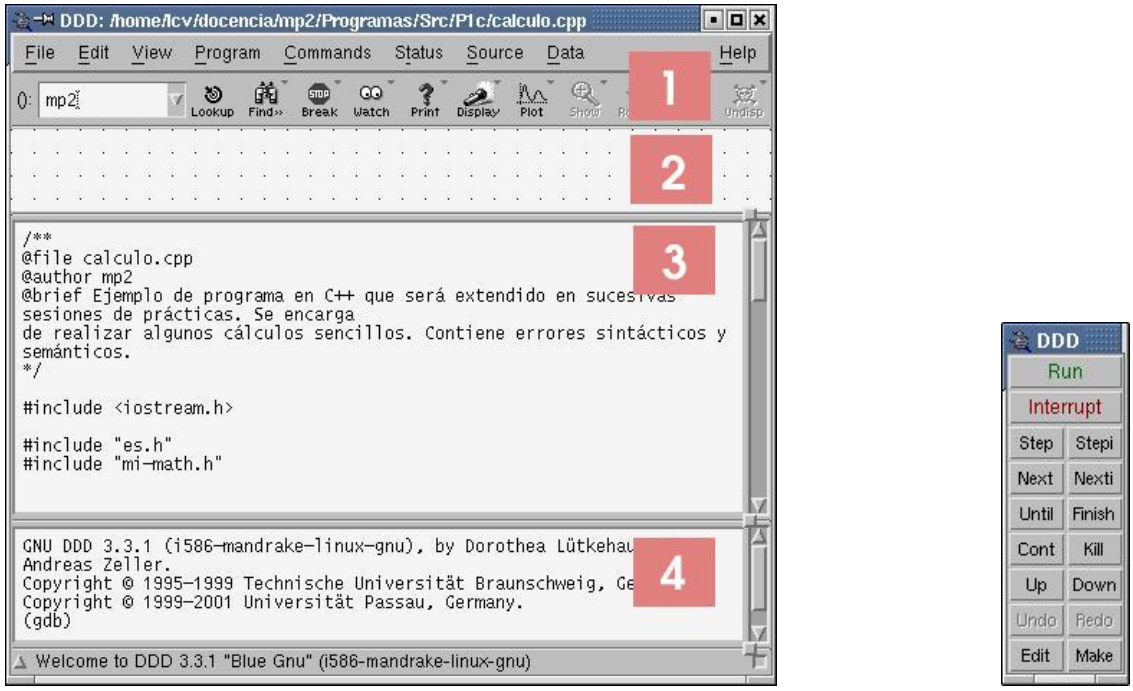
El programa `ddd` es, básicamente, una interfaz (*front-end*) separada que se puede utilizar con un depurador en línea de órdenes. En el caso que concierne a este documento, `ddd` será la interfaz de alto nivel del depurador `gdb`.

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 ruptura[1] en la primera línea ejecutable del código. Una vez colocado este punto de ruptura se puede comenzar la ejecución del programa paso a paso según lo mostrado en el cuadro 1 y teniendo en cuenta que `ddd` señala la línea de código activa con una pequeña flecha verde a la izquierda de la línea  . `ddd` también muestra la salida de la ejecución del programa en una ventana independiente (`DDD: Execution window`). Si esta ventana no

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

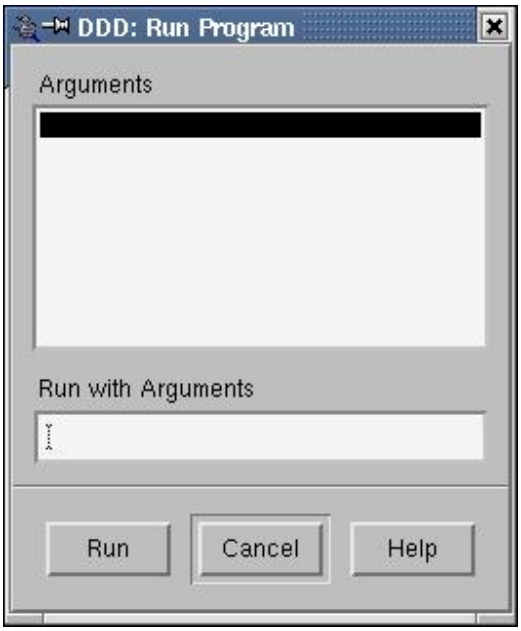


Figura 6: Ventana para pasar argumentos a `main`

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. Asímismo, `ddd` permite modificar, en tiempo de ejecución, los valores asociados a cualquier variable de un programa, bien desde la ventana del código, bien desde la ventana de visualización de datos.

Inspección de la pila

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

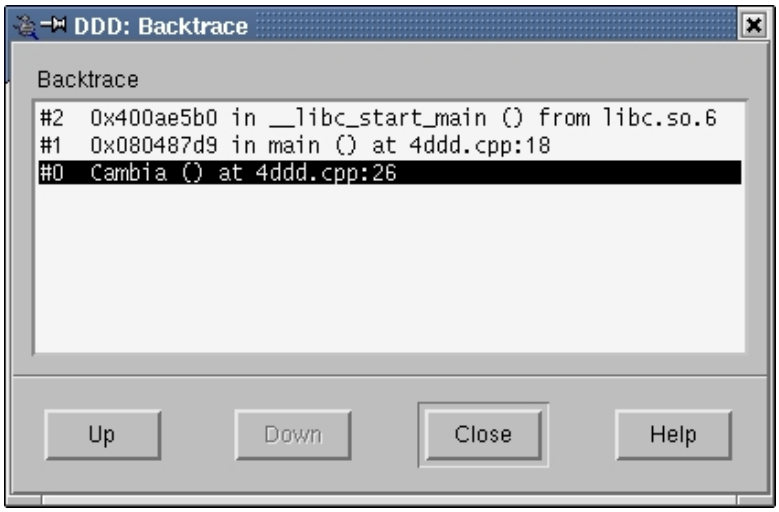


Figura 7: Ventana que muestra el estado de la pilla de llamadas a módulos

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 incluído sustituye a la línea `#include`.

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

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

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

El uso de las directivas de preprocesamiento proporciona varias ventajas:

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

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

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

`#undef`: Eliminación de constantes simbólicas.

`#include`: Inclusión de ficheros.

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

Constantes simbólicas y macros funcionales

Constantes simbólicas

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

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

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

Veamos algunos ejemplos sencillos.

```
#define TAMMAX 256
```

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

```
#define UTIL_VEC
```

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

```
#include <iostream>
```

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

Inclusión condicional de código

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

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

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

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

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

#include CABECERA
```

De esta forma, estamos seguros de que incluiremos el fichero de cabecera apropiado al sistema en el que estemos compilando. Por supuesto, debemos especificar de alguna forma el valor de la constante `SISTEMA` (por ejemplo, usando macros en la llamada al compilador, como indicamos en la página 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 claúsula `#define`). Este predicado se suele usar en su forma resumida (columna derecha):

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

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

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

sí puede ocurrir lo siguiente:

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

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

```
#ifndef (HDR)
#define HDR

    Resto del contenido del fichero de cabecera

#endif
```

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

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

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 sotware y las modificaciones incorporadas pueden afectar a otros módulos: la modificación de una función en un módulo afecta *necesariamente* a los módulos que usan dicha función, que deben actualizarse. Estas modificaciones deben *propagarse* a los módulos que dependen de aquellos que han sido modificados, de forma que el programa ejecutable final refleje las modificaciones introducidas.

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

La utilidad `make` proporciona los mecanismos adecuados para la gestión de proyectos software. Esta utilidad mecaniza muchas de las etapas de desarrollo y mantenimiento de un programa, proporcionando mecanismos simples para obtener versiones actualizadas de los programas, por complicado que sea el diagrama de dependencias entre módulos asociado al proyecto. Esto se logra proporcionando a la utilidad `make` la secuencia de mandatos que crean ciertos archivos, y la lista de archivos que necesitan otros archivos (*lista de dependencias*) para ser actualizados antes de que se hagan dichas operaciones. Una vez

especificadas las dependencias entre los distintos módulos del proyecto, cualquier cambio en uno de ellos provocará la creación de una nueva versión de los módulos dependientes de aquel que se modifica, reduciendo al mínimo necesario e imprescindible el número de modulos a recompilar para crear un nuevo fichero ejecutable.

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

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

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

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

El programa make

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

Examinando las listas de dependencia determina qué ficheros ha de reconstruir. El criterio es muy simple, se comparan fechas y horas: si el fichero fuente es más reciente que el fichero destino, reconstruye el destino. Este sencillo mecanismo (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 construidos. Para saber qué destinos debe construir comprueba las listas de dependencias. Esta reacción en cadena se llama **dependencia encadenada**.
3. Si en cualquier paso falla al construir algún destino, se detiene la ejecución, muestra un mensaje de error y borra el destino que estaba construyendo.

Ficheros makefile

Un fichero makefile contiene las órdenes que debe ejecutar la utilidad `make`, así como las dependencias entre los distintos módulos del proyecto. Este archivo de descripción es un fichero de texto.
Los elementos que pueden incluirse en un fichero makefile son los siguientes:

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

Comentarios

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

Ejercicio

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

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

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

Reglas. Reglas explícitas

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

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

El formato habitual de una regla explícita es el siguiente:

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

donde:

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

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

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

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

Ejercicio

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

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

que indica que para construir el objetivo `saludo` se requiere la existencia de `saludo.cpp` (`saludo` *depende de* `saludo.cpp`). Esta dependencia se esquematiza en el diagrama de dependencias mostrado en la figura 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
```

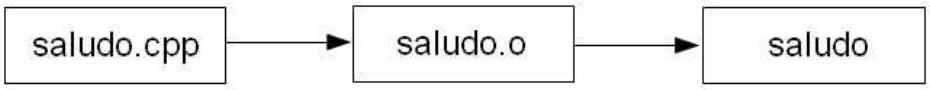


Figura 9: Diagrama de dependencias para `makefile2.mak`

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 sóla ejecución de `make`. Obsérvese cómo el nombre dado al destino simbólico no tiene importancia.

Ejercicio

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

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

Ejercicio

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

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

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

```
    move saludo unico resultado
```

Destinos .PHONY

Si en un fichero makefile apareciera una regla:

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

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

```
make clean
```

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

```
.PHONY : clean
```

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

```
make clean
```

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

Macros en ficheros makefile

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

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

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

NombreMacro = texto a expandir

donde:

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

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

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

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

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

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

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

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

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

se sustituye por:

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

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

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

y se modifican las anteriores por:

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

Ahora, la regla anterior se sustituye por:

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

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

```
HOMEDIR = ~/MP
```

Ejercicio

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

Macros predefinidas

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

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

Ejercicio

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

Sustituciones en macros

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

$(NombreMacro:TextoOriginal = TextoNuevo)

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

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

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

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

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

Macros como parámetros en la llamada a `make`

Además de las opciones básicas indicadas en la página 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_ppal_2` para que el resultado (el ejecutable `ppal_2`) lo guarde en un directorio cuyo nombre **completo** (camino absoluto, desde la raiz `/`) se indica como parámetro al fichero makefile con una macro llamada `DESTDIR`.

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

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

Ejercicio

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

Reglas implícitas

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

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

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

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

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

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

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

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

El valor de estas macros pueden ser definidas:

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

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

Ejercicio

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

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

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

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

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

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

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

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

Ejercicio

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

Reglas implícitas patrón

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

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

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

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

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

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

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

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

Reglas patrón estáticas

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

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

donde:

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

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

```
OBJETOS = f1.o f2.o

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

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

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. Apender 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 subtarea residen en un fichero fuente.

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

1. La estructura de datos se puede utilizar de forma sencilla en otros programas.

2. Las funciones relacionadas se almacenan juntas.

3. Cualquier cambio posterior en la estructura de datos, requiere que se modifique y recompile el mínimo número de ficheros y sólo los imprescindibles.

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

Organización de los ficheros fuente

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

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

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

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

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

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

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

Ejercicio

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

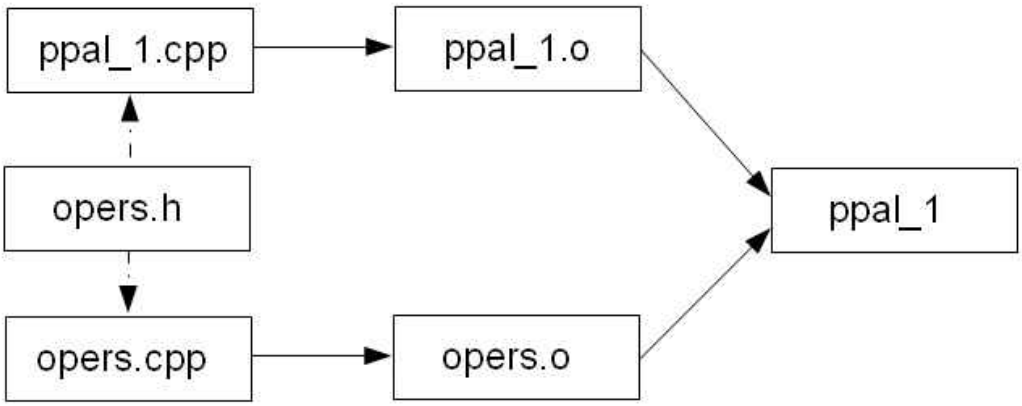


Figura 10: Diagrama de dependencias para construir `ppal_1`

Ejercicio

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

Los ficheros involucrados y sus dependencias se muestran en la figura 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.

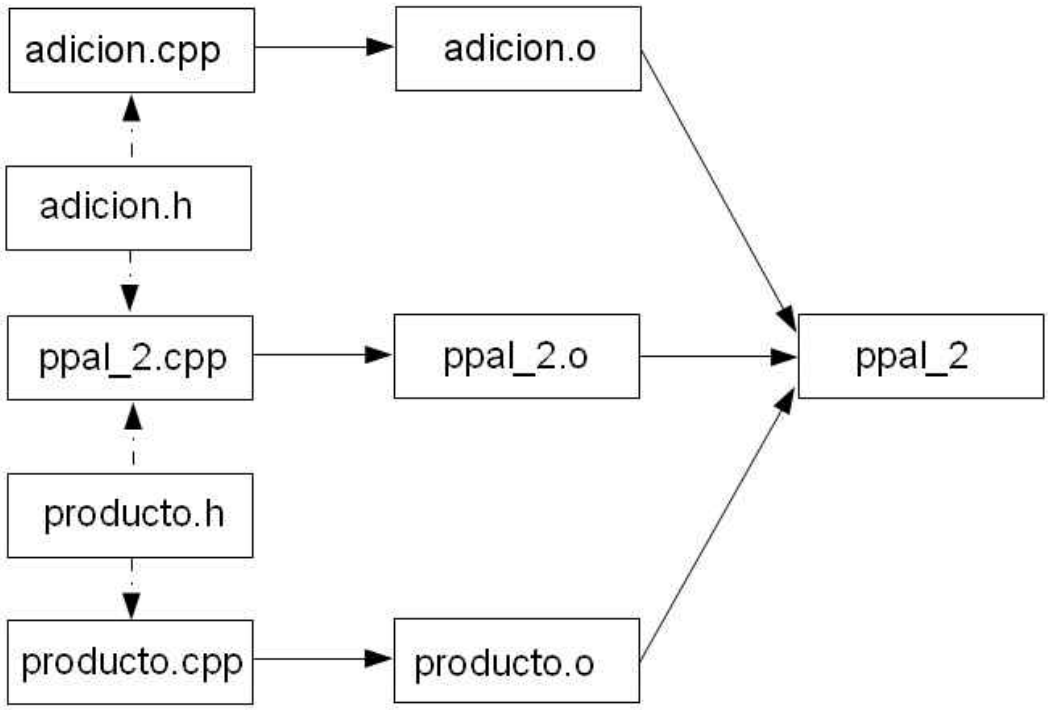


Figura 11: Diagrama de dependencias para construir `ppal_2`

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 `ppal_2` a partir de los tres módulos objeto generados.

Ejercicio

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

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

Ejercicio

Escribir un fichero llamado `makefile_ppal_2` para generar el ejecutable `ppal_2`, siguiendo el diagrama de dependencia mostrado en la figura 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 **TODOS** los programas que usan esta función, lo que llevará a utilizar diferentes versiones de la misma función si el control no es muy estricto o a un esfuerzo considerable para mantener la coherencia del software.

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

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

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

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

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

Tipos de bibliotecas

Bibliotecas estáticas

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

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

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

Bibliotecas dinámicas

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

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

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

Estructura de una biblioteca

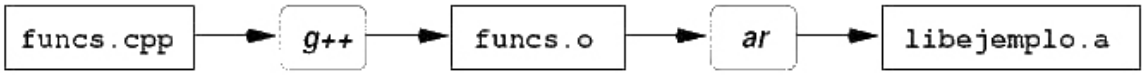
Una biblioteca se estructura internamente como un **conjunto de módulos objeto**. Cada uno de estos módulos será el resultado de la compilación de un fichero de código fuente que puede contener una o varias funciones.

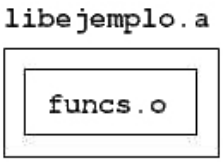
La extensión por defecto de los ficheros de biblioteca es `.a` y se acostumbra a que su nombre empiece por el prefijo `lib`. Así, por ejemplo, si hablamos de la biblioteca `ejemplo` el fichero asociado se llamará `libejemplo.a`

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

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

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.

fun01.cpp

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

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

fun10.cpp

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

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

Figura 13: Varios módulos fuente (caso 2)

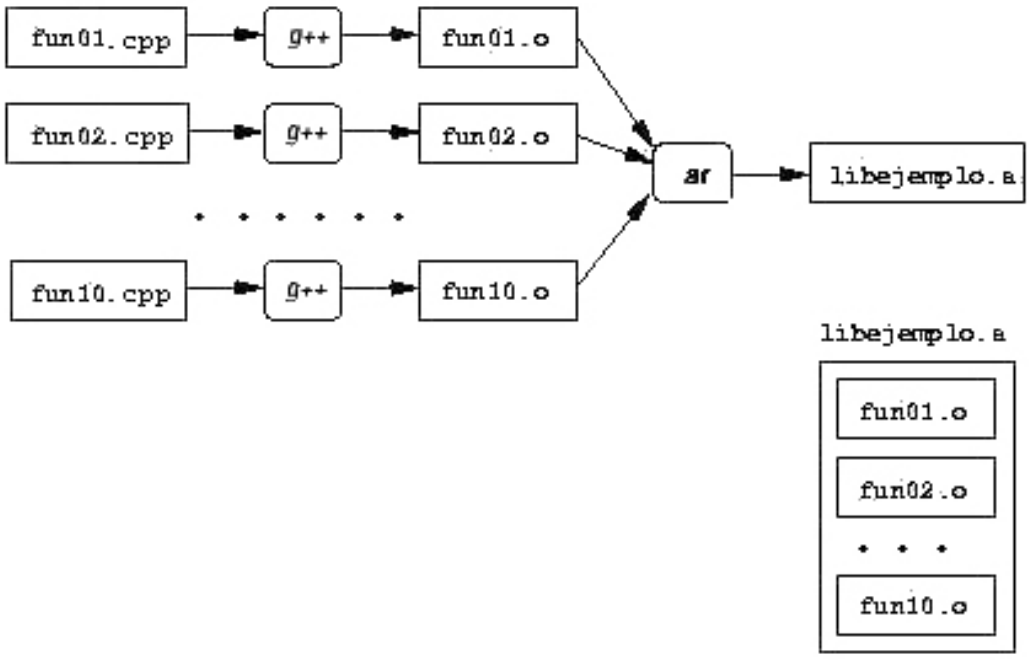


Figura 14: Construcción de una biblioteca a partir de varios módulos objeto (caso 2)

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 construído la biblioteca.

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

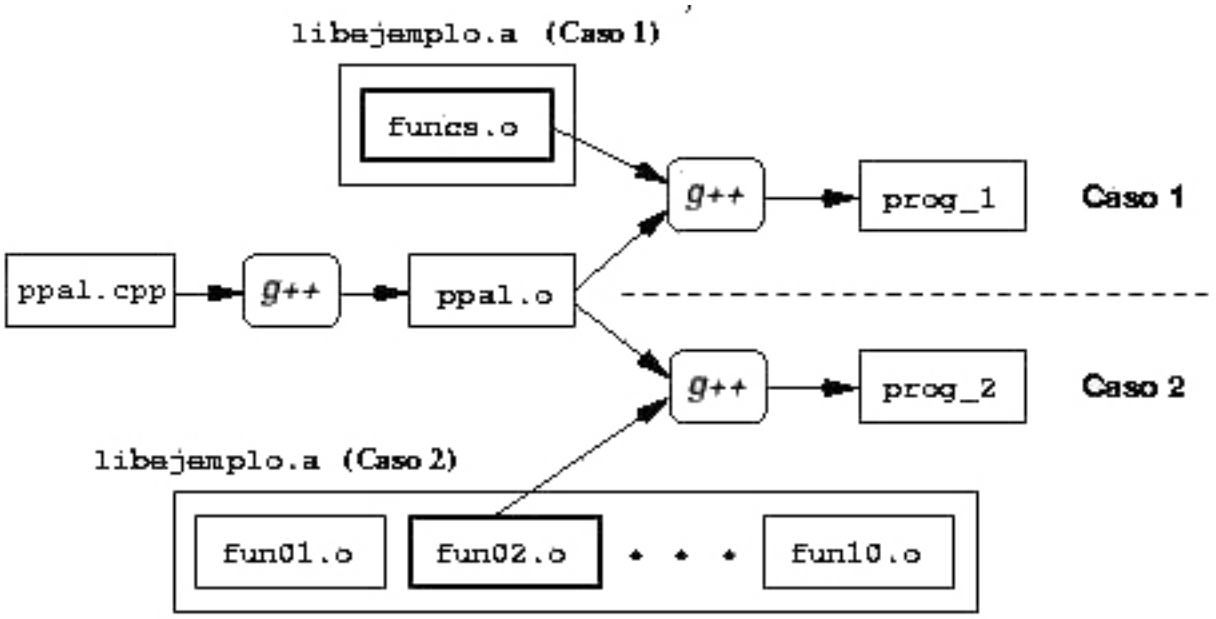


Figura 15: Construcción de un ejecutable que usa una función de biblioteca. Caso 1: biblioteca formada por un módulo objeto. Caso 2: biblioteca formada por 10 módulos objeto

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

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

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

El programa `ar`

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

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

donde:

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

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

Ejercicio

Repetir los siguientes ejemplos:

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

Ejercicio

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

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

2. Mostrar los módulos que la componen.

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

Ejercicio

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

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

2. Mostrar los módulos que la componen.

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

3. Mostrar el directorio actual.

Ejercicio

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

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

2. Mostrar los módulos que la componen.

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

3. Mostrar el directorio actual.

g++, make y ar trabajando conjuntamente

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

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

Creación de la biblioteca

Emplearemos una regla para la construcción de la biblioteca.

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

Para los dos casos estudiados en la página 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 bibilioteca

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

Ejercicio

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

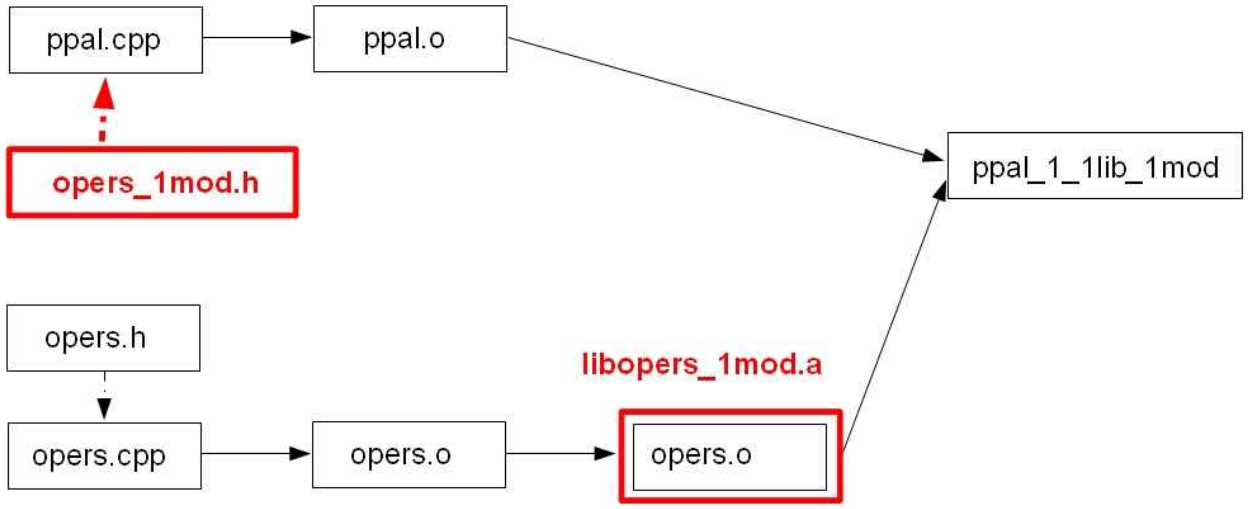


Figura 16: Construcción de un ejecutable que usa funciones de biblioteca (1). Una biblioteca formada por un módulo objeto que implementa cuatro funciones.

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_1lib_2mod` que implementa el diagrama de la figura 17

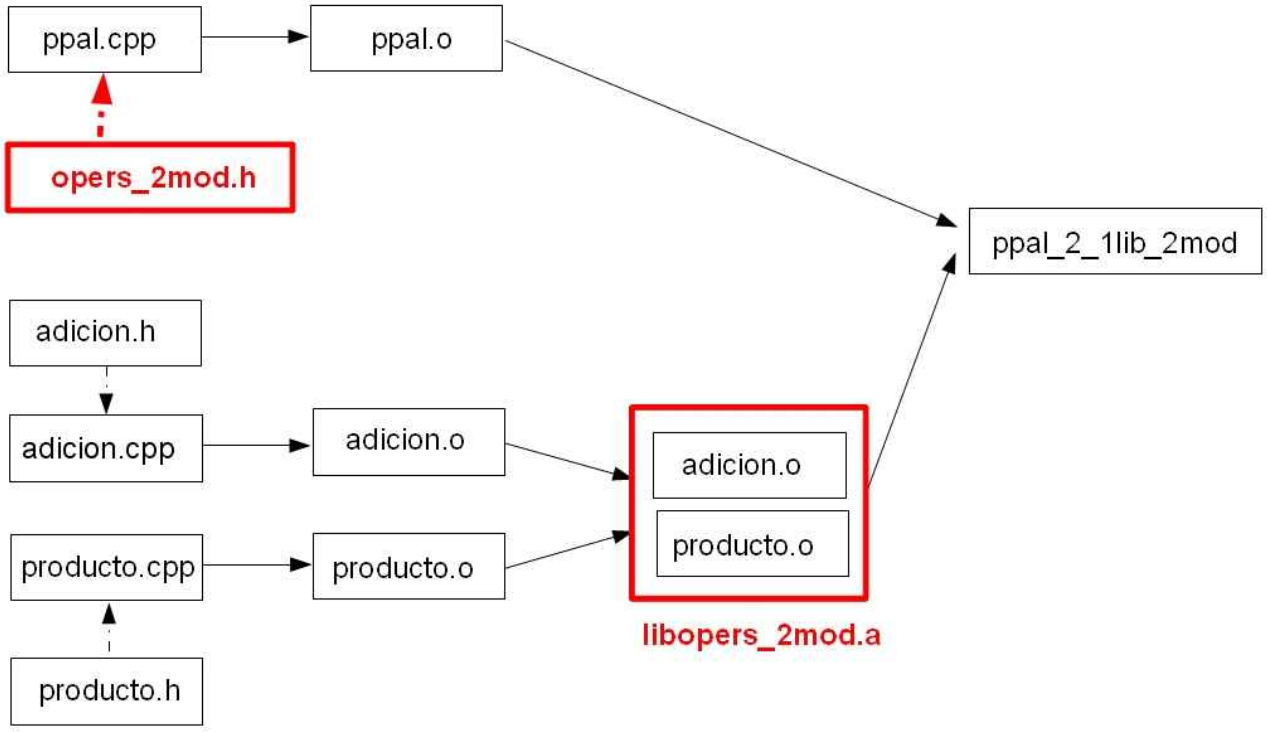


Figura 17: Construcción de un ejecutable que usa funciones de biblioteca (2). Una biblioteca formada por dos módulos objeto que implementan dos funciones cada uno de ellos.

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?

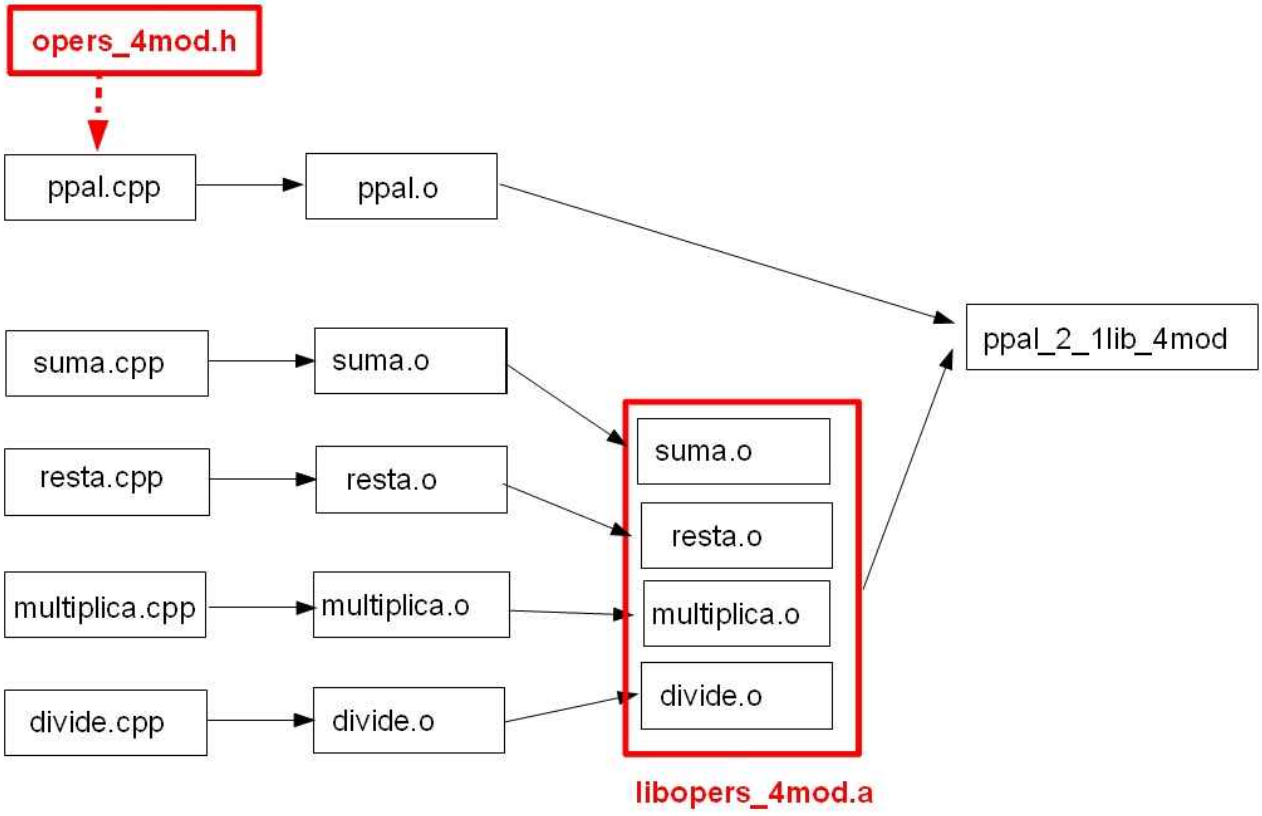


Figura 18: Construcción de un ejecutable que usa funciones de biblioteca (3). Una biblioteca formada por cuatro módulos objeto, y cada uno define una única función.

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.

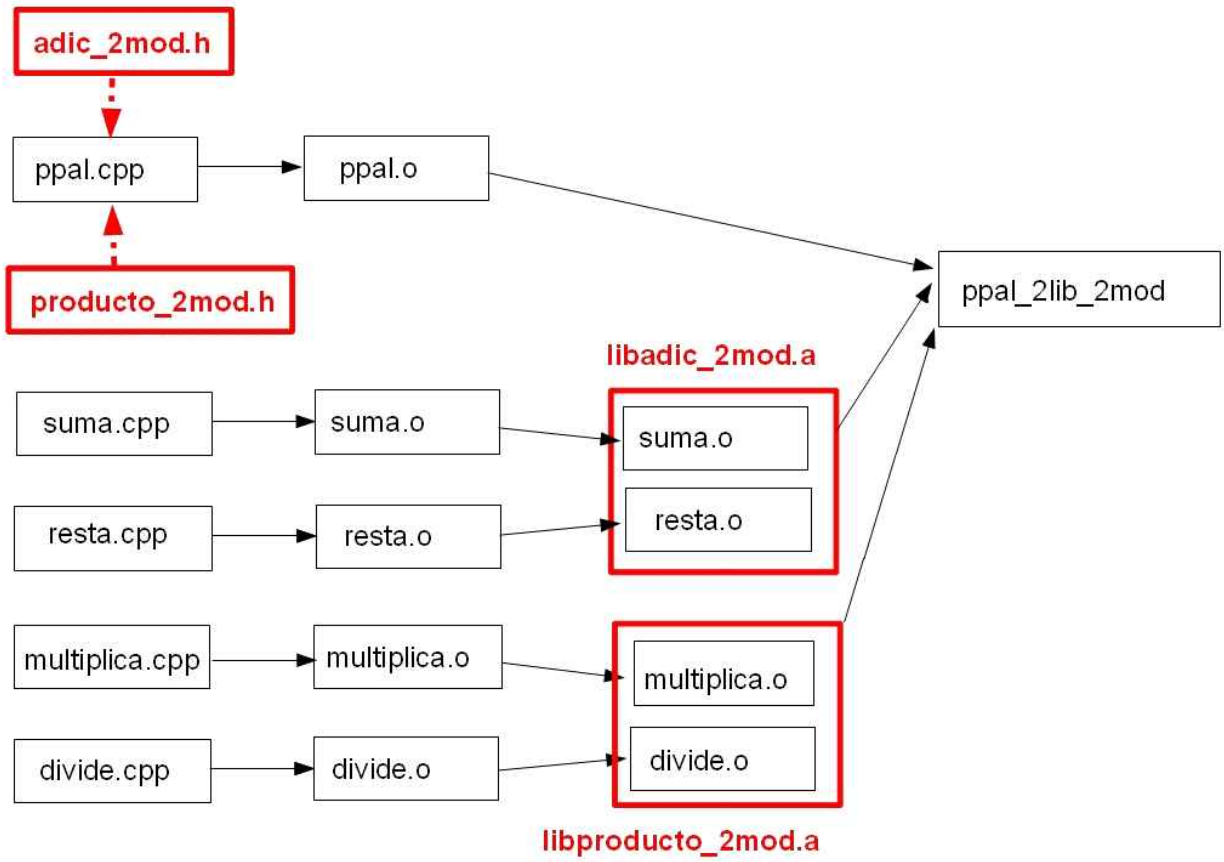


Figura 19: Construcción de un ejecutable que usa funciones de biblioteca (4). Dos bibliotecas formadas cada una por dos módulos objeto, y cada uno define dos funciones.

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 (1)

► Objetivos

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

1. Escribir funciones que reciben/devuelven punteros. En particular, punteros que contienen la dirección de memoria de algún elemento de un vector.
2. Reutilizar soluciones de problemas ya resueltos.
3. Modularizar en diferentes ficheros la solución a un problema.
4. Gestionar el proyecto empleando un fichero *makefile*.

► Actividades a realizar en casa

Actividad: Lectura de soluciones de ejercicios resueltos.

Lea la solución de los ejercicios 2, 3, 4, 5 y 6 que podrá encontrar en PRADO (sección dedicada a las sesiones de prácticas 1,2 y 3).

Actividad: Resolución de problemas.

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

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

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

► *Recomendaciones*

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

► *Normas detalladas*

1. Deberá entregar únicamente el fichero `MPsesion04.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_sesion04.mak`.
2. **MUY IMPORTANTE:** También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.
3. **No** incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter ~
4. No se admitirán otras soluciones que no cumplan escrupulosamente las normas indicadas.

Sesión 5

Punteros (2). Funciones (1)

► Objetivos

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

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

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

► Actividades a realizar en casa

Actividad: Resolución de problemas.

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

- *Obligatorios:*

  20 (Encontrar palabras) `I_EncuentraInicioPalabras.cpp`.
  22 (Formatea líneas) `I_FormateaLineas.cpp`.
  24 (Posición del mayor) `I_PosMayor.cpp`
  25 (Mezcla de *arrays*) `I_MezclaArrays.cpp`.

- *Opcionales:*

  21 (Delimita palabras) `I_EncuentraPalabras.cpp`.
  23 (Leer entero / Leer entero en un rango) `I_LeeEntero.cpp`.

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

Si cree conveniente ejecutar algún programa con *redirección de entrada* (tomando los datos de un fichero de texto) debe tener en cuenta esta advertencia:

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. 1) 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.

2) En el caso de usar **Sublime Text** seleccione `View` | `Line endings` | `Unix`

► Normas detalladas

1. Deberá entregar únicamente el fichero `MPsesion05.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_sesion05.mak`.
2. **MUY IMPORTANTE:** También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.
3. **No** incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter ~
4. No se admitirán otras soluciones que no cumplan escrupulosamente las normas indicadas.

Sesión 6

Punteros (3). Funciones (2)

► Objetivos

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

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

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

► Actividades a realizar en casa

Actividad: Resolución de problemas.

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

- *Obligatorios:*

  26 (Mezcla de *arrays* -con referencias-) `I_MezclaArrays_ref.cpp`.
  27 (Clase `SecuenciaEnteros`) `I_DemoSecuenciaEnteros.cpp`.
  33 (Ordenación "especial") `I_NuevoOrdenEnteros.cpp`.

- *Opcionales:*

  29 (Máximo y mínimo de un array) `I_MaxMin_Array.cpp`.
  30 (Sucursales) `I_Sucursales_Matriz_Clasica.cpp`.

Para poder generar correctamente los ejecutables pedidos en esta práctica será preciso escribir un fichero makefile, al que llamarán `makefile_sesion06.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.

► *Normas detalladas*

1. Deberá entregar únicamente el fichero `MPsesion06.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_sesion06.mak`.
2. **MUY IMPORTANTE:** También deberá incluirse en el paquete las carpetas **vacías** `obj`, `lib` y `bin`, para poder generar la solución correctamente.
3. **No** incluir los ficheros resultantes de las copias de seguridad que se generan automáticamente y que terminan con el carácter ~
4. No se admitirán otras soluciones que no cumplan escrupulosamente las normas indicadas.

Sesión 7

Gestión de memoria dinámica (1): Vector dinámico

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

1. En el primero (ejercicios 1, 2, 3, 4, 5 y 6) se trabaja sobre vectores dinámicos.

   Este será el trabajo a desarrollar en esta práctica.

2. En el segundo (ejercicios 7, 8, 9, 10, 11, 12, 13 y 14) se trabaja con matrices dinámicas.

3. En el tercero 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. Gestionar estructuras de datos con información heterogénea (`struct`) como una herramienta sencilla para encapsular las propiedades básicas de tipos de datos complejos.

4. 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 II:

2 (Vector dinámico) `II_Demo-VectorDinamico.cpp`.

Modularizar la solución escribiendo:

- `II_Demo-VectorDinamico.cpp` para la función `main`
- `FuncsVectorDinamico.h` para albergar la declaración del tipo `VectorDinamico` y las declaraciones (los prototipos) de las funciones.
- `FuncsVectorDinamico.cpp` con la definición de las funciones.
- Construir la biblioteca (`libFuncsVectorDinamico.a`) que ofrezca las funciones implementadas.

4 (Descomposición en factores primos con vector dinámico números primos) `II-FactoresPrimos_VectorDinamicoPrimos.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.

- Ejercicios ***opcionales*** de la Relación de Problemas II:

3 (Encuentra palabras) `II_EncuentraPalabras_MemDin.cpp`

6 (MultiConjunto) `II_Demo-MultiConjunto.cpp`.

Modularizar la solución escribiendo:

- `II_Demo-MultiConjunto.cpp` para la función `main`
- `MultiConjunto.h` para albergar la declaración del tipo `MultiConjunto` y las declaraciones (los prototipos) de las funciones.
- `MultiConjunto.cpp` con la definición de las funciones.
- Construir la biblioteca (`libMultiConjunto.a`) que ofrezca las funciones implementadas.

► *Actividades a realizar en las aulas de ordenadores*

En estas circunstancias, ninguna :-(

► *Recomendaciones*

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

► *Normas detalladas*

Deberá entregar únicamente el fichero `MPsesion07.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_sesion07.mak`.

MUY IMPORTANTE: 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 otras soluciones que no cumplan escrupulosamente las normas indicadas.

Sesión 8

Gestión de memoria dinámica (2): Matrices dinámicas

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

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

   Este será el trabajo a desarrollar en esta práctica.
3. En el tercero 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) el tipo de dato `Matriz2D` -tipos 1 y 2-. Si la modularización se hace bien, los programas que usan estos tipos de datos serán idénticos, y el usuario de la clase `Matriz2D` (la función `main`) no sabría qué implementación se ha usado -tipo 1 ó 2-.

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:*

  7 (Matriz dinámica - filas independientes) `II_Demo-Matriz2D.cpp`.

  Modularizar la solución escribiendo:

  - `II_Demo-Matriz2D.cpp` para la función `main`
  - `Matriz2D.h` para albergar la declaración del tipo `Matriz2D` y los prototipos de las funciones.
  - `Matriz2D.cpp` con la definición de las funciones.
  - Construir la biblioteca `libMatriz2D.a`.

  9 (Relieve) `II_Relieve.cpp`.

  - Escriba la función `main` en `II_Relieve.cpp` .
  - En este ejercicio intervienen las clases: `Relieve`, `Punto2D` y `ColeccionPuntos2D`. Modularice las clases en las parejas de ficheros `Relieve.h` y `Relieve.cpp`, `Punto2D.h` y `Punto2D.cpp`, y finalmente `ColeccionPuntos2D.h` y `ColeccionPuntos2D.cpp`.
  - Use la biblioteca `libMatriz2D.a` construida en los ejercicios 7 (`Matriz2D` tipo 1) ó 8 (`Matriz2D` tipo 2).

- *Opcionales:*

  8 (Matriz dinámica - datos en una fila) `II_Demo-Matriz2D_tipo2.cpp`.

  Modularizar la solución escribiendo:

  - `II_Demo-Matriz2D_tipo2.cpp` para la función `main`
  - `Matriz2D.h` para albergar la declaración del tipo `Matriz2D` y los prototipos de las funciones.
  - `Matriz2D.cpp` con la definición de las funciones.
  - Construir la biblioteca `libMatriz2D.a`.

  10 (Asignación de tareas a técnicos) `II_ProblemaAsignacion.cpp`.
  11 (Viajante de comercio) `II_ProblemaViajanteComercio.cpp`.
  12 (Viajante de comercio - Ampliación-) `II_ProblemaViajanteComercio_Ampliacion.cpp`.
  13 (Sucursales - Matriz dinámica) `II_Sucursales_MatrizDinamica.cpp`.

En los ejercicios 10, 11, 12, y 13 se usarán matrices dinámicas `Matriz2D`. Use la biblioteca `libMatriz2D.a` construida en los ejercicios 7 (`Matriz2D` tipo 1) ó 8 (`Matriz2D` tipo 2).

14 (Secuenciación genética) `II_SecuenciacionGenetica.cpp`.

Sugerimos modularizar la solución del ejercicio 14 dejando la función `main` en el fichero `II_SecuenciacionGenetica.cpp` y en los ficheros:

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

► *Actividades a realizar en las aulas de ordenadores*

En estas circunstancias, ninguna :-(

► *Recomendaciones*

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

► *Normas detalladas*

Deberá entregar únicamente el fichero `MPsesion08.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_sesion08.mak`.

MUY IMPORTANTE: 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 otras soluciones que no cumplan escrupulosamente las normas indicadas.

Sesión 9

Gestión de memoria dinámica (3): Listas enlazadas

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

1. En el primero (ejercicios 1, 2, 3, 4, 5 y 6) se trabaja sobre vectores dinámicos.
2. En el segundo (ejercicios 7, 8, 9, 10, 11, 12, 13 y 14) se trabaja con matrices dinámicas.
3. En el tercero (ejercicios 15, 16, 17, 18 , 19, 20, 21, 22, 23 y 24) se trabaja con listas enlazadas.
   Este será el trabajo a desarrollar en esta práctica.

► Objetivos

1. Seguir practicando con la reserva de memoria en el heap y su liberación. Escribir funciones que reservan memoria dinámica y devuelven la dirección del bloque reservado en el Heap.
2. Gestionar estructuras lineales dinámicas autorreferenciadas (tipo *lista*).
3. Gestionar estructuras de datos con información heterogénea (`struct`) como una herramienta sencilla para encapsular las propiedades básicas de tipos de datos complejos.
4. 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: Lectura de soluciones

16 (Estadística sobre lista) `II_EstadisticaBasica_Lista.cpp`

Dispone del fichero con la función `main` que resuelve completamente el problema.

19 (Ordenar lista) `II_OrdenarLista.cpp`

El fichero `II_OrdenarLista.cpp` resuelve el problema planteado, aunque para poder compilarlo deberá implementar todas las funciones solicitadas en el problema. Le proporcionamos una versión reducida (`II_OrdenarLista_Basico.cpp`) en la que solo se ordena con el método de *selección*.

Al finalizar esta sesión debe disponer de la biblioteca `libLista.a` completa. Las primeras funciones se enumeran en el ejercicio 15 y en los siguientes ejercicios se van incorporando nuevas funciones. En la solución que encontrará en PRADO dispone de una versión reducida de la biblioteca a la que hemos llamado `ListaMinima.a`. Use esta biblioteca como el punto de partida de `liblista.a`

Actividad: Resolución de problemas.

15 (Tipo `Lista`) `II_Demo-Lista.cpp`.

Modularizar la solución escribiendo:

- `II_Demo-Lista.cpp` para la función `main`
- `Lista.h` para albergar la declaración de los tipos asociados a la lista y los prototipos de las funciones.
- `Lista.cpp` con la definición de las funciones.
- Construir la biblioteca `libLista.a`.

17 (Listas aleatorias) `II_EstadisticaBasica_ListaAleatoria.cpp`
18 (Invertir lista) `II_Invierte_Lista.cpp`
20 (Insertar y eliminar en lista ordenada) `II_GestionarListaOrdenada.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.

► *Actividades a realizar en las aulas de ordenadores*

En estas circunstancias, ninguna :-(

► *Recomendaciones*

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

► *Normas detalladas*

Deberá entregar únicamente el fichero `MPsesion09.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_sesion09.mak`.

MUY IMPORTANTE: 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 otras soluciones que no cumplan escrupulosamente las normas indicadas.

Sesión 10

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 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: Lectura de soluciones

1 (Clase `Secuencia`). Proyecto `sesion10_previa.tar`

Dispone en PRADO del fichero `sesion10_previa.tar` que le permite generar la clase `Secuencia` y probar su funcionalidad con la función `main` de `III_Demo-Secuencia.cpp`

Actividad: Resolución de problemas.

- *Obligatorios:*

  2 (Clase *matriz dinámica -filas independientes-*) `III_Demo-Matriz2D.cpp`
  3 (Clase *Lista*) `III_Demo-Lista.cpp`
  4 (Clase *Pila*) `III_Demo-Pila.cpp`
  8 (Clases `PoliLinea` y `Punto2D`) `III_Demo-PoliLinea.cpp`

- *Opcionales:*

  5 (Clase *Cola*) `III_Demo-Cola.cpp`
  6 (`Pila` y `Cola` usando `Lista`)
  `III_Demo-Pila_Alternativa` y/ó `III_Demo-Cola_Alternativa`
  7 (Red de Metro) `III_Demo-Metropolitano`

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. Las bibliotecas básicas se llamarán como se detalla en el cuadro 2.

Clase	**Ficheros**	**Biblioteca**
Secuencia	`Secuencia.h` `Secuencia.cpp`	`libSecuencia.a`
Matriz dinámica	`Matriz2D.h` `Matriz2D.cpp`	`libMatriz2D.a`
Lista	`Lista.h` `Lista.cpp`	`libLista.a`
Pila	`Pila.h` `Pila.cpp`	`libPila.a`
Cola	`Cola.h` `Cola.cpp`	`libCola.a`

Cuadro 2: Nomenclatura de los ficheros

Nota: No se trata de la versión definitiva de la las clases. 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.

► Normas detalladas

Deberá entregar únicamente el fichero `MPsesion10.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_sesion10.mak`.

MUY IMPORTANTE: 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 otras soluciones que no cumplan escrupulosamente las normas indicadas.

Sesión 11

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 1).

► Objetivos

1. Ampliar la funcionalidad de las clases sobre la que empezamos a trabajar en la sesión de prácticas 9: secuencias, 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. Implementar **operadores aritméticos**.
8. Implementar **operadores combinados** y **relacionales**, y diferentes sobrecargas de éstos.
9. Evitar duplicar código y escribir el código de algunos operadores basándose en el código escrito para otros operadores.
10. Seguir profundizando en la compilación separada de programas.

► Actividades a realizar en casa

Actividad: Lectura de soluciones

1 (Clase *Secuencia* (2^a parte)) Proyecto: `Secuencia_v2.tar`

6 (*Red de metro* (2^a parte)) Proyecto: `Metropolitano_v2.tar`

Dispone en PRADO de los ficheros `Metropolitano_v2.tar` y `Secuencia_v2.tar` con las soluciones.

Actividad: Resolución de problemas.

- *Obligatorios:*

  2 (Clase *matriz dinámica*) `IV_Demo-Matriz2D.cpp`
  3 (Clase *Lista*) `IV_Demo-Lista.cpp`
  7 (Clase `PoliLinea`) `IV_Demo-PoliLinea.cpp`

- *Opcionales:*

  4 (Clase *Pila*) `IV_Demo-Pila.cpp`
  5 (Clase *Cola*) `IV_Demo-Cola.cpp`
  8 (Clase `RedCiudades`) `IV_Demo-RedCiudades.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 básicas se llamarán como se detalla en el cuadro 2.

► Recomendaciones

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

► Normas detalladas

Deberá entregar únicamente el fichero `MPsesion11.tar` resultante de empaquetar todos los ficheros `.cpp` y `.h` necesarios (organizados en sus correspondientes carpetas `src` e `include`) así como el fichero `makefile_sesion11.mak`.

MUY IMPORTANTE: 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 otras soluciones que no cumplan escrupulosamente las normas indicadas.

UNIVERSIDAD
DE GRANADA

METODOLOGÍA DE LA PROGRAMACIÓN

RELACIONES DE PROBLEMAS

Grado en Ingeniería Informática
GRUPO A

Francisco José Cortijo Bon
cb@decsai.ugr.es

2019/2020

Departamento de
Ciencias de la Computación
e Inteligencia Artificial

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

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

   Nota: Para inicializar aleatoriamente el vector puede emplear la clase `GeneradorAleatorioEnteros` usada en la asignatura *Fundamentos de Programación*) y que puede descargar de la página de la asignatura.

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

4. *(Adaptación del examen de la convocatoria extraordinaria de FP - Febrero 2020)*

   Un entero se dice que es **pentagonal** si puede obtenerse a partir de la fórmula $\frac{n(3n-1)}{2}$ para algún valor de $n > 0$. Los primeros 10 números pentagonales son 1, 5, 12, 22, 35, 51, 70, 92, 117 y 145. Se dice que una pareja de números pentagonales (p_i, p_j) es **metapentagonal** si la media entre ambos $\frac{p_i + p_j}{2}$ y la diferencia de ambos -en valor absoluto- $abs(p_i - p_j)$ también son números pentagonales. Observe que ambos valores son enteros mayores que cero y menores que el máximo entre p_i y p_j.

   Escribir un programa que lea un entero n ($0 < n \leq 1000$) que será el número de enteros pentagonales a considerar y calcule cuántas parejas de números metapentagonales hay entre ellos. Para ello:

   1.- Use la función

   ```
   int Pentagonal (int n);
   ```

   para generar el n-ésimo número pentagonal.

   2.- Genere y guarde en un array, `pentagonales`, los primeros n enteros pentagonales. Realice esta tarea con la función `RellenaVectorPentagonales`.

   3.- Un número p será pentagonal si está en el array `pentagonales`. Escriba la función `PosicionPentagonal` para buscar un valor p en el array `pentagonales` y que devuelva su posición (p es pentagonal), ó `-1` si no está (p no es pentagonal).

   4.- Una pareja de números pentagonales (p_i, p_j) es **metapentagonal** si la media y la diferencia entre ellos (ver definición concreta más arriba) también están en el array `pentagonales`. Escriba la función `EsParejaMetapentagonal` para realizar esta tarea: devolverá `true` si p_i y p_j son metapentagonales.

   Debe priorizar la rapidez de ejecución del programa.

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, *ordenadamente*, los valores de `v1` y `v2` en el vector `res`.

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

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

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

   La función tendrá como prototipo:

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

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

   Considere la siguiente declaración:

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

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

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

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

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

   Nota: Modularice la solución con funciones.

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

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

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

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

   Implemente tres funciones para la ordenación:

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

9. En este ejercicio se combinan las soluciones de los ejercicios 5 y 8.

   Con estas declaraciones:

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

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

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

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

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

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

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

    Notas:

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

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

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

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

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

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

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

13. 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.

14. 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.

15. 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.

16. 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.

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

```
char * extraer_cadena (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.

18. Escriba funciones para eliminar separadores en una cadena:

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

Todas las funciones tienen un esquema común:

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

Notas:

- La cadena original **no** se modifica.
- Se supone que hay suficiente memoria en la cadena resultado.
- No se puede usar el operador `[]`: se debe resolver mediante aritmética de punteros.

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

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

Notas:

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

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

    Para registrar el inicio de cada palabra, usen un *array* de punteros a carácter (cada puntero contendrá la dirección del primer carácter de una palabra). En la figura 20.A mostramos el estado de la memoria para la función `main` justo antes de llamar a la función `encuentra_palabras` (para calcular el inicio de cada palabra). En la figura 20.B mostramos el estado de la memoria al empezar la ejecución de la función `encuentra_palabras`.

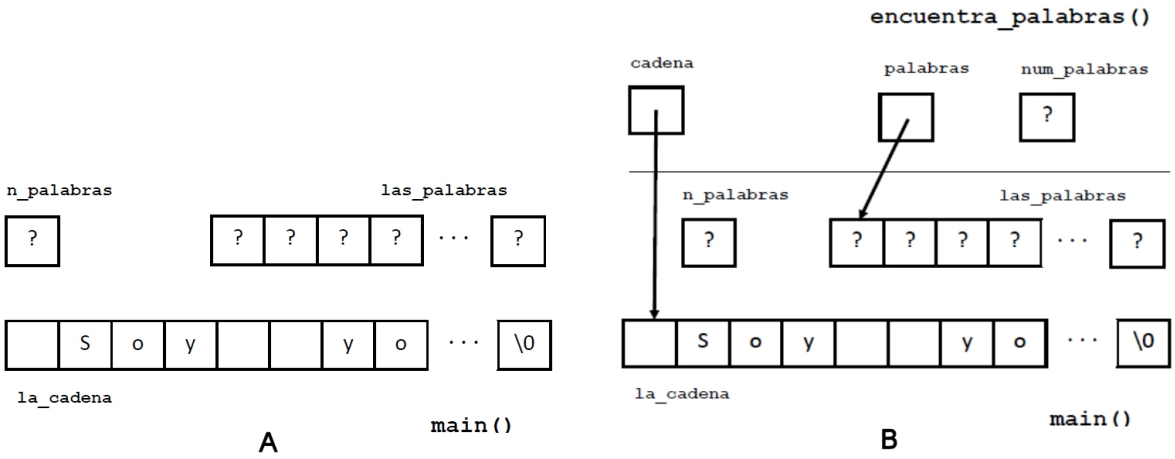


Figura 20: Estado de la pila A) antes de llamar a la función, `encuentra_palabras` y B) inmediatamente después de empezar la función

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

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

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

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

donde (ver figura 20.B):

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

En este ejemplo la función devolverá 2 y se llamó:

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

En la figura 21.A mostramos el estado de la memoria al finalizar la función `encuentra_palabras`, justo antes de devolver el control a la la función `main` y en la la figura 21.B mostramos el estado de la memoria al volver el control a `main`.

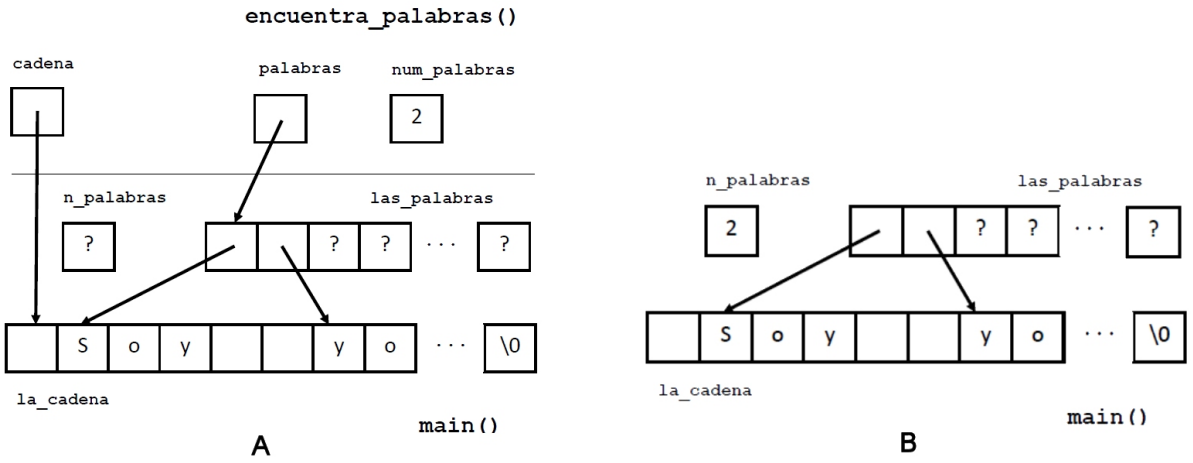


Figura 21: Estado de la pila A) antes de llamar a la función, `encuentra_palabras` y B) inmediatamente después de empezar la función

Notas:

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

21. Este ejercicio es una ampliación del ejercicio 20. Ahora estamos interesados en saber no solo dónde empieza cada palabra, sino también dónde acaba.

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

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

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

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

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

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

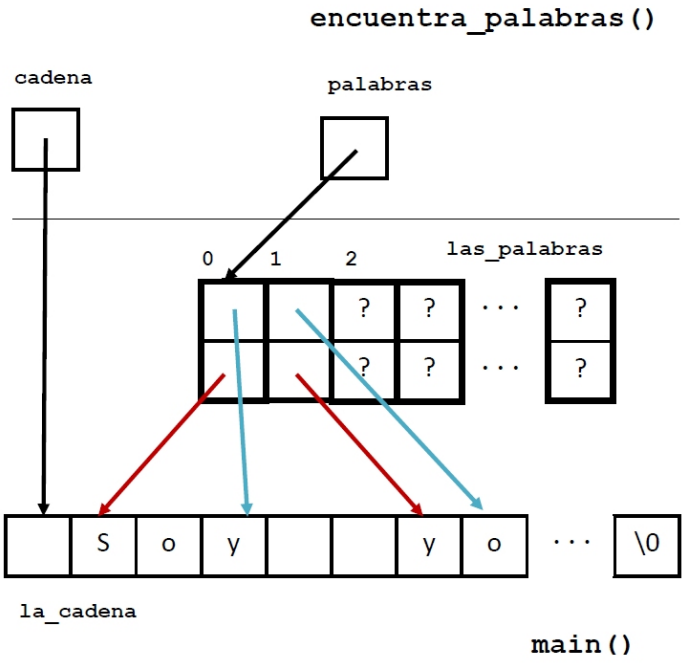


Figura 22: Estado de la pila antes de devolver el control a `main`

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:

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

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

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

Notas:

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

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

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

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

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

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

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

    Así, el siguiente código:

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

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

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

    Si el programa anterior se ejecuta con su entrada estándar redirigida tomando los datos desde un fichero mostrará el contenido de dicho fichero, línea a línea. O sea, en cada iteración del ciclo `while` se lee una línea del fichero y se muestra. Si la lectura "fracasa" (encuentra el indicador de fin de fichero) no se vuelve a entrar al ciclo `while`.

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

Por ejemplo, si el fichero contiene:

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

#include <iostream>
#include <iomanip>
```

podría generar los siguientes resultados:

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

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

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

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

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

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

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

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

23. Escriba una función para leer y devolver un número entero. La función:

    a) leerá el número usando una cadena,

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

        I) los dígitos `'0'`, `'1'`, ..., `'9'` y

        II) los caracteres `+` ó `-` (únicamente en la primera posición)

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

    c) la convertirá a un valor entero.

    Si todo es correcto, devolverá el dato `int`; 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 `int`: admitirá valores entre los dos indicados.

    Probad las funciones escribiendo un programa.

24. Este ejercicio se basa en el ejercicio 7 de esta Relación de Problemas. Reutilizar el código que pueda ser aprovechado, especialmente la función `PosMayor`.

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

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

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

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

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

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

    Nota: Modularice la solución con funciones.

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

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

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

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

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

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

Usar la función;

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

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

Usar una función como:

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

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

El contenido de un vector se mostrará usando la función:

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

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

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

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

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

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

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

26. Modificar el proyecto realizado en el ejercicio 25 de manera que ahora las funciones de mezcla quedan *resumidas* en una única función:

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

de manera que:

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

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

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

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

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

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

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

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

3.- Añada un constructor que cree una secuencia con un números dado de valores iguales.

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

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

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

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

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

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

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

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

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

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

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

int & Valor (int indice);
```

8.- Añada el método `EsIgualA`:

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

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

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

En cualquier caso, encontrará en PRADO un código que trabaja sobre la clase `SecuenciaEnteros` y que le servirá para validar la implementación realizada.

28. Escriba la función

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

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

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

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

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

En la figura 23 mostramos el resultado de la función cuando se "ordena" el vector `vec` entre las casillas 0 y 5.

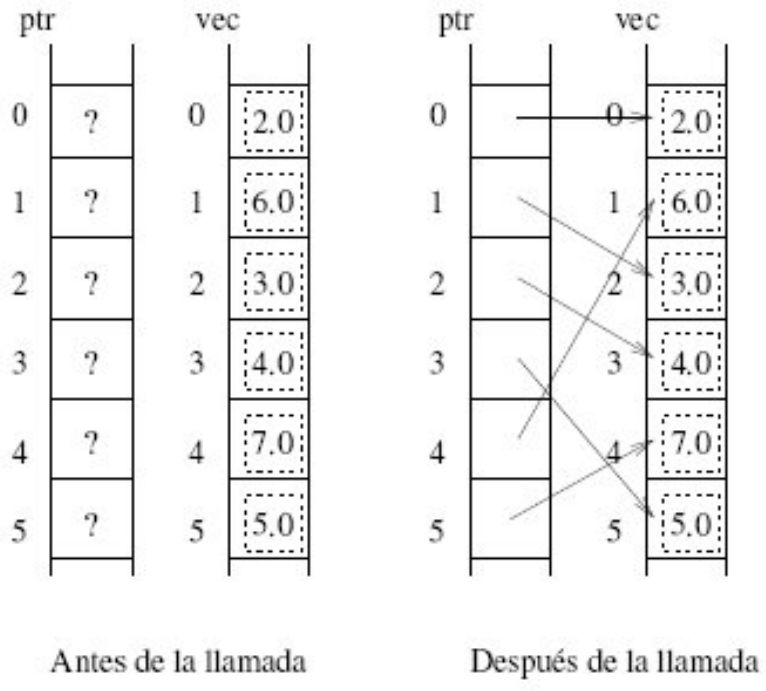


Figura 23: Resultado de ordenar el vector de punteros

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

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

Nota: Iniciar aleatoriamente el vector `vec`.

Nota: Modularice la solución con funciones.

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

    Modularizar la solución usando, al menos, funciones para:

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

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

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

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

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

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

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

    Después de leer los datos de ventas el programa mostrará:

    a) El número total de operaciones de venta.

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

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

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

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

f) Producto más vendido y cuántas unidades.

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

h) Cuántos tipos de producto han sido vendidos.

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

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

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

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

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

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

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

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

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

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

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

31. Represente gráficamente la disposición en memoria de las variables del programa siguiente e indique lo que escribe la última instrucción.

```
struct Celda {
   int d;
   Celda *p1, *p2, *p3;
}

int main (void) {

   Celda a, b, c, d;

   a.d = b.d = c.d = d.d = 0;

   a.p1 = &e;
   c.p3 = &d;
   a.p2 = a.p1->p3;
   d.p1 = &b;
   a.p3 = c.p3->p1;
   a.p3->p2 = a.p1;
   a.p1->p1 = &a;
   a.p1->p3->p1->p2->p2 = c.p3->p1;
   c.p1->p3->p1 = &b;
   (*((*(e-p3->p1)).p2->p3)).p3 = a.p1->p3;
   d.p2 = b.p2;
   (*(a.p3->p1)).p2->p2->p3 = (*(a.p3->p2)).p3->p1->p2;

   a.p1->p2->p2->p1->d = 5;
   d.p1->p3->p1->p2->p1->p1->d = 7;
   (*(d.p1->p3)).p3->d = 9;
   c.p1->p2->p3->d = a.p1->p2->d - 2;
   (*(c.p2->p1)).p2->d = 10;

   cout << "a= "  << a.d << " b= " << b.d
        << " c= " << c.d << " d= " << d.d << endl;
}
```

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

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

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

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

T	C	G	G	G	G	g	T	T	T	t	t
c	C	G	G	t	G	A	c	T	T	a	C
a	C	G	G	G	G	A	T	T	T	t	C
T	t	G	G	G	G	A	c	T	T	t	t
a	a	G	G	G	G	A	c	T	T	C	C
T	t	G	G	G	G	A	c	T	T	C	C
T	C	G	G	G	G	A	T	T	c	a	t
T	C	G	G	G	G	A	T	T	c	C	t
T	a	G	G	G	G	A	a	c	T	a	C
T	C	G	G	G	t	A	T	a	a	C	C

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

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

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

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

Dicha secuencia es una secuencia de ADN (una cadena clásica) que se forma asignando a su i-ésima posición el nucleótido (A, C, G, T) que más veces se repite en la posición i de todas las secuencias (es decir, en la columna i de la tabla) En caso de empate, se elige cualquiera de los repetidos. En el ejemplo anterior, la secuencia de consenso generada sería: `T C G G G G A T T T C C`

c) Calcule si dos tablas tienen la misma secuencia de consenso.

Implemente para ello la función `MismoConsenso`

d) Calcule la *secuencia inconexa* de una tabla.

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

Implemente el cálculo de la distancia de Hamming entre dos secuencias con una función.

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

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

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

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

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

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

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

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

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

    El programa recibe valores de la línea de órdenes:

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

34. Este ejercicio amplía la funcionalidad del ejercicio 33.

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

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

`bool MenorClasico (int a, int b)`

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

`bool MenorPrimeraCifra (int a, int b)`

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

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

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

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. La estrategia que se seguirá será una de estas tres (aunque deberá programar las tres posibilidades):

   a) *casilla a casilla*

   b) en *bloques de tamaño fijo (*`TAM_BLOQUE`*)*

   c) *duplicando* su tamaño.

   La función que redimensiona el vector tendrá el siguiente prototipo:

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

   - `p` es una referencia que permite acceder al vector dinámico que se va a redimensionar (a 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 la función **modifica** (afortunadamente) 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 24 mostramos un vector dinámico `v` que responde a la representación en memoria propuesta.

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

// Capacidad inicial
const int TAM_INICIAL = 10;

// Tamanio 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 reservadas

   // PRE: 0 <= usados <= capacidad
   // Inicialmente, capacidad = TAM_INICIAL

   TipoRedimension  tipo_redim; // Modelo de crecimiento

 } VectorDinamico;
```

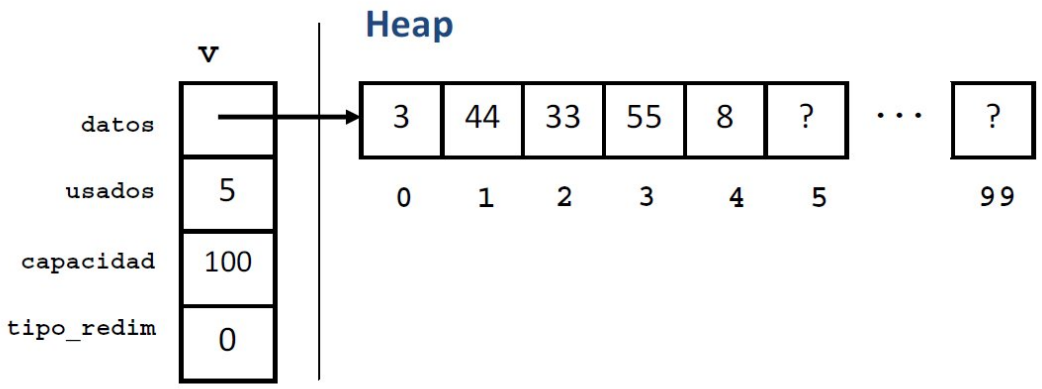


Figura 24: Vector dinámico con 100 casillas reservadas y 5 usadas

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

```
VectorDinamico CreaVectorDinamico (int cap_inic=TAM_INICIAL,
            TipoRedimension tipo=TipoRedimension::EnBloques);
```

Crear vector dinámico. Cuando se crea el vector dinámico se establece el tipo de redimensionamiento. La función `RedimensionaVectorDinamico` modifica (añade) el número de casillas reservadas. La estrategia que se seguirá para añadir casillas es una de estas tres (deberá programar las tres posibilidades): a) *casilla a casilla*, b) en *bloques de tamaño fijo (de acuerdo al valor de la constante* `TAM_BLOQUE`*)* ó c) *duplicando* su tamaño.

Opcional: Funciones para consultar/modificar el tipo de redimensionamiento.

```
int UsadosVectorDinamico (const VectorDinamico v);
```

Consultar casillas usadas. Devuelve el número de casillas *usadas* en un vector dinámico (el valor del campo `usados`).

```
int CapacidadVectorDinamico (const VectorDinamico v);
```

Consultar casillas reservadas. Devuelve el número de casillas *reservadas* en un vector dinámico (el valor del campo `capacidad`).

```
void LiberaVectorDinamico (VectorDinamico &v);
```

Liberar memoria. Libera la memoria ocupada por un vector dinámico. Use esta función cuando ya no vaya a usar el vector.

```
void EliminaTodosVectorDinamico (VectorDinamico & v)
```

Elimina todos los valores. Elimina todos los valores del vector. Al finalizar, `UsadosVectorDinamico(v)==0`. La capacidad no se modifica. El vector sigue "activo".

```
void EcualizaVectorDinamico (VectorDinamico & v,
                             const TipoBase valor);
```

Ecualización. Inicializa el vector dinámico, sustituyendo el contenido de todas las casillas por el valor `valor`.

```
string ToString (const VectorDinamico v);
```

Serialización. Crea un `string` que contiene los valores del vector dinámico. El objetivo es conseguir un formato *agradable* y compacto para poder mostrar/transmitir el contenido del vector.

```
void AniadeVectorDinamico (VectorDinamico &v,
                           const TipoBase valor);
```

Añadir un valor. Añade un valor al final del vector dinámico. Si el vector dinámico estuviera lleno se tiene que redimensionar (de acuerdo al tipo de redimensión establecido) para poder acoger al nuevo valor.

```
TipoBase & ValorVectorDinamico (VectorDinamico v,
                                const int num_casilla);
```

Consulta ó modifica el valor de una casilla dada. Si se utiliza como *rvalue* se emplea para consultar el valor de la casilla `num_casilla`. Si se utiliza como *lvalue* se emplea para modificar el valor de la casilla `num_casilla`. Se establece la precondición:
`PRE: 0<=num_casilla<UsadosVectorDinamico(v)`

```
void InsertaVectorDinamico (VectorDinamico &v,
             const TipoBase valor, const int num_casilla);
```

Inserción. Inserta un valor en una posición dada. Los valores que están desde esa posición (incluida) hasta la última se "desplazan" una posición hacia índices mayores. Si el vector dinámico estuviera lleno se tiene que redimensionar (de acuerdo al tipo de redimensión establecido) para poder acoger al nuevo valor. Se establece la precondición: `PRE: 0<=num_casilla<UsadosVectorDinamico(v)`

```
void EliminaVectorDinamico (VectorDinamico & v,
                            const int num_casilla);
```

Borrado. Elimina un valor en una posición dada. Los valores que están desde la posición siguiente a la dada hasta la última se "desplazan" una posición hacia índices menores. `PRE: 0<=num_casilla<UsadosVectorDinamico(v)`

```
void  RedimensionaVectorDinamico (VectorDinamico &v);
```

Redimensión. Redimensiona un vector dinámico de acuerdo al tipo registrado. El efecto evidente para el usuario de la clase es la modificación del campo `capacidad`.

```
void  ReajustaVectorDinamico (VectorDinamico &v);
```

Reajuste. Redimensiona un vector dinámico para que no haya espacio libre (todas las casillas reservadas están ocupadas). El efecto evidente para el usuario de la clase es la modificación del campo `capacidad` (después de su ejecución, `CapacidadVectorDinamico(v) == UsadosVectorDinamico(v)`).

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

Para probar esta implemantación del módulo *VectorDinámino* proponemos realizar un programa en el que la función `main` admite argumentos desde la línea de órdenes para que 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 de tamaño fijo* y 3 indica *duplicando*.

En la función `main`:

- se crea un vector dinámico con `TAM_INICIAL` casillas,
- se lee un número indeterminado de valores (terminando cuando se introduzca `FIN`),
- se añaden los valores al vector conforme se van leyendo.

Una vez se termina la introducción de datos deberá probar todas las funciones enumeradas anteriormente.

3. Retomamos el ejercicio 21 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.

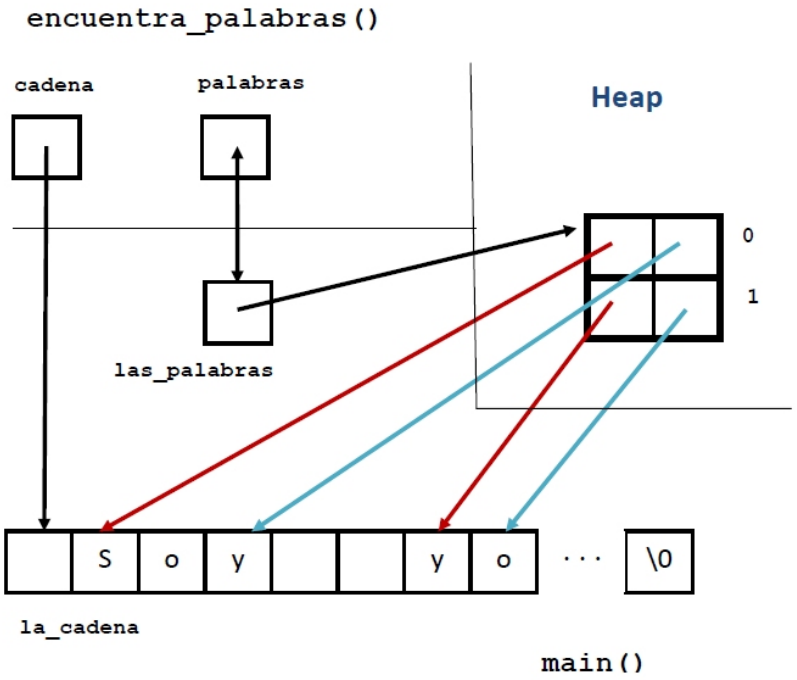


Figura 25: Estado de la pila antes de devolver el control a `main`

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 25 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);
```

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.

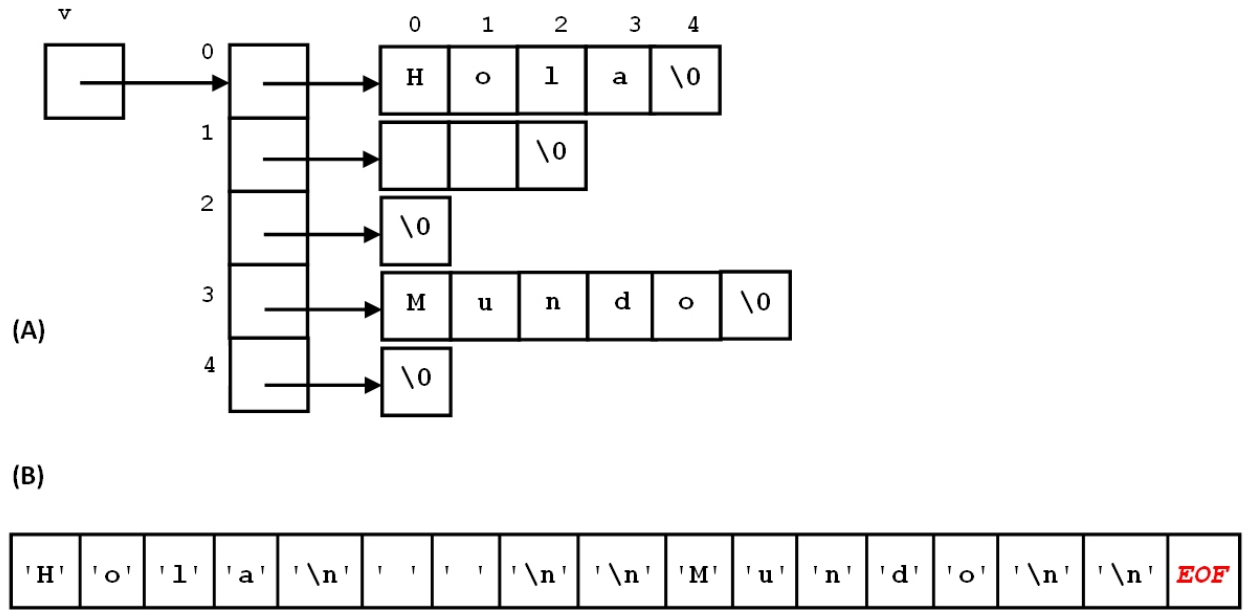


Figura 26: Cada línea del fichero (B) se guarda en una cadena clásica, accesible desde `v`, un vector dinámico de punteros a `char` (A)

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

Una vez leida 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 26.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 26.B. En esa figura, *EOF* indica la marca de *fín de fichero* que se introduce manualmente desde el teclado con la combinación de teclas `Ctrl`+`D` en Linux (`Ctrl`+`Z` en Windows).

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* (**E**nd **O**f **F**ile).

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).

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.

---

6. *(Versión adaptada del Examen Ordinario FP 2018/2019)* En matemáticas, un *multiconjunto* es una modificación del concepto de conjunto que, a diferencia de éste, permite múltiples instancias para cada uno de sus elementos. El número entero positivo de instancias de cada elemento se llama *multiplicidad* de este elemento en el multiconjunto. Por ejemplo, en $\{5, 5, 5, 3, 3, 2\}$, 5 tiene multiplicidad 3, 3 tiene multiplicidad 2 y 2 tiene multiplicidad 1.

   Los elementos de un multiconjunto se toman en un conjunto fijo U (*universo*). Dado un multiconjunto A en un universo U, para cada elemento $x \in U$, definimos $m_A(x)$ como la multiplicidad de x en A. Entonces, se pueden definir las siguientes operaciones sobre multiconjuntos:

   - El **soporte** de un multiconjunto A en U es: $Sop(A) = \{x \in U | m_A(x) > 0\}$.
   - A está **incluido** en B, denotado como $A \subseteq B$, si $\forall x \in Sop(A),\ \ m_A(x) \leq m_B(x)$.
   - La **intersección** de A y B es un multiconjunto C tal que $C = \{x : x \in Sop(A)$ Y $x \in Sop(B)\}$ y además, $m_C(x) = min\{m_A(x), m_B(x)\}\ \forall x \in Sop(C)$.
   - La **unión** de A y B es un multiconjunto C tal que $C = \{x : m_C(x) = max\{m_A(x), m_B(x)\}\ \forall x \in Sop(C)$.

   Observe que no es necesario representar explícitamente todas las repeticiones de cada valor. Así pues, en vez de representar el multiconjunto $\{5, 5, 5, 3, 3, 2\}$ en un array con enteros repetidos, resulta más conveniente trabajar con la multiplicidad de cada valor del universo U. Por lo tanto, para representar el anterior multiconjunto basta con considerar cada valor del dominio (5, 3, 2) y sus multiplicidades asociadas (3,2,1). Guardar los valores por orden no es estrictamente necesario.

   Un multiconjunto podría representarse como sigue. Observe que la representación es muy parecida a la de un `VectorDinamico`:

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

// Capacidad inicial
const int TAM_INICIAL = 10;
```

```
// Tamanio 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 * multiplicidad; // Puntero para acceder a la
                        // multiplicidad de cada valor.
   int usados;    // Num. casillas usadas
   int capacidad; // Num. casillas ocupadas

   // PRE: 0 <= usados <= capacidad
   // Inicialmente, capacidad = TAM_INICIAL

   TipoRedimension tipo_redim; // Método de redimensión

} MultiConjunto;
```

Realizar un programa que gestione datos de tipo `MultiConjunto` con $U = \{1, 2, ..., 1000\}$. El programa leerá un número indefinido (termina cuando se lee un valor negativo) de valores de U y creará un `MultiConjunto`. A continuación hará lo mismo para crear otro `MultiConjunto`.

Después, calculará el soporte de los dos datos `MultiConjunto` y calculará la inclusión, la intersección y la unión de los dos datos `MultiConjunto`.

Para ello deberá, al menos, implementar las siguientes funciones (le aconsejamos usar como modelo el ejercicio 2):

a) Escriba una función para crear un `MultiConjunto` vacío. Escriba una función para liberar los recursos que ocupa un `MultiConjunto`.

b) Implemente una función para obtener la multiplicidad de un número perteneciente a U en un `MultiConjunto` y otra para añadir un elemento x al `MultiConjunto` con cierta multiplicidad k. Si el elemento ya existe, se incrementará su multiplicidad en k.

c) Escriba funciones que implementen las tres operaciones definidas con anterioridad (soporte, inclusión e intersección). El soporte devolverá un objeto de la clase `SecuenciaEnteros`, la inclusión un bool y la intersección un dato `Multiconjunto`.

d) Implemente una función que devuelva un objeto de la clase `SecuenciaEnteros` con **todos** los elementos de un `MultiConjunto`. No se requiere que estén ordenados. Para el ejemplo anterior, se devolvería $\{5, 5, 5, 3, 3, 2\}$.

e) Implemente una función que devuelva un objeto de la clase `SecuenciaEnteros` con el *soporte* de un `MultiConjunto`. No se requiere que los valores estén ordenados. Para el ejemplo anterior, se devolvería $\{5, 3, 2\}$.

Modularice adecuadamente el proyecto.

Ejercicios sobre matrices dinámicas

7. Para trabajar con matrices bidimensionales dinámicas de datos de tipo `TipoBase` usamos una estructura como la que aparece en la figura 27 (tipo `Matriz2D`, filas independientes) en la que ilustramos cómo se almacena en memoria una matriz dinámica de 10 filas y 15 columnas.

   Nota: Las filas y columnas se numeran desde 0.

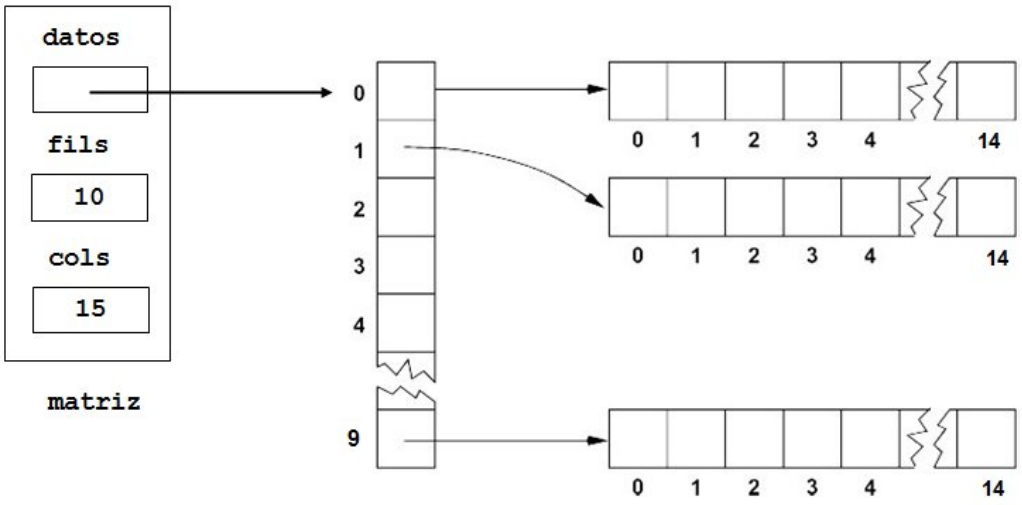


Figura 27: Tipo `Matriz2D`: datos guardados en filas independientes

Deberá escribir la biblioteca `libMatriz2D.a` para gestionar matrices dinámicas. Proporcionará la funcionalidad que se enumera a continuación. Establezca las precondiciones que sean razonables en cada función.

a) `Matriz2D CreaMatriz (int nfils=0, int ncols=0, TipoBase valor=VALOR_DEF);`

Crear matriz dinámica. Crea una matriz dinámica con `nfils` filas y `cols` columnas. Inicializa todas las casillas a un valor común, el indicado en el parámetro `valor` (`0` si `TipoBase` es `int`, `0,0` si `TipoBase` es `double`, ...). La funcionalidad sería similar a la de un *constructor*.

b) `void DestruyeMatriz (Matriz2D & matriz);`

Destruir matriz dinámica. "Destruye" una matriz dinámica y la deja en un estado no útil (vacía). La matriz queda vacía (todos sus campos a cero). La tarea de esta función sería similar a la de un *destructor*.

c) `int NumFilas (Matriz2D & matriz);`

Número de filas. Devuelve el número de filas de la matriz.

d) `int NumColumnas (Matriz2D & matriz);`

Número de columnas. Devuelve el número de columnas de la matriz.

e) `bool EstaVacia (Matriz2D & matriz);`

¿Vacía?. Devuelve `true` si la matriz está vacía.

f) `void Ecualiza (Matriz2D & matriz, TipoBase valor=VALOR_DEF);`

Ecualizar. Cambia todos los todos valores de la matriz por `valor`.

g) `void EliminaTodos (Matriz2D & matriz);`

Vacíar matriz dinámica. Deja la matriz en un estado no útil (vacía). La matriz queda con todos sus campos a cero.

h) `string ToString (Matriz2D & matriz);`

Serialización. Devuelve un `string` con el resultado de "serializar" una matriz.

i)
```
TipoBase & Valor (Matriz2D & v,
                  int num_fila, int num_columna);
```

Consulta ó modifica el valor de una casilla dada. Si se utiliza como *rvalue* se emplea para consultar el valor de la casilla de la fila `num_fila` y columna `num_columna`. Si se utiliza como *lvalue* se emplea para modificar el valor de la casilla de la fila `num_fila` y columna `num_columna`.

j)
```
void Clona (Matriz2D & destino,
            const Matriz2D & origen);
```

Copia profunda. Hace una copia **profunda** de `origen` en `destino`.

k)
```
bool SonIguales (const Matriz2D & una,
                 const Matriz2D & otra);
```

Comparación. Devuelve `true` si las matrices `una` y `otra` son exactamente iguales (dimensiones y contenidos).

l)
```
void SubMatriz (Matriz2D & resultado,
                const Matriz2D & original,
                int fila_inic, int col_inic,
                int num_filas, int num_cols);
```

Submatriz. Extrae una submatriz de `original` y la deja en `resultado`. La submatriz `resultado` es una zona rectangular de `original` que empieza en la casilla de coordenadas (`fila_inic`, `col_inic`) y que tiene (un máximo de) `num_filas` filas y `num_cols` columnas.
Considere todos los posibles casos entre `fila_inic`, `col_inic`, `num_filas`, `num_cols` y el número de filas y columnas de `original` y resuélvalos de la mejor manera posible. Por ejemplo,

- Si la matriz original tiene 5 filas y 8 columnas, y le pedimos una submatriz de 4 filas y 3 columnas desde la casilla (fila) 3, (columna) 4, no podremos construir una submatriz con 5 filas: contando desde la fila 3 (la primera) ya que sólo disponemos de dos filas (la 3 y la 4) aunque sí podemos disponer de 3 columnas (las 4. 5 y 6). La función construirá y rellenará una matriz de 2 filas y 3 columnas.
- Sobre la misma matriz, si pedimos una submatriz como la descrita antes pero desde la casilla (fila) 9, (columna) 9, no podremos construir la submatriz pedida ya que la casilla inicial no está dentro de la matriz. La función construirá y "rellenará" una matriz vacía. Lo mismo ocurre si la casilla inicial fuera, por ejemplo, la casilla 2 (fila), -2 (columna).

m) `void EliminaFila (Matriz2D & matriz, int num_fila);`

Eliminar fila. Elimina la fila `num_fila` de la matriz `matriz`.

n) `void EliminaColumna (Matriz2D & matriz, int num_col);`

Eliminar columna. Elimina la columna `num_col` de la matriz `matriz`.

ñ) `void EspejoHorizontal (Matriz2D & matriz);`

Espejo horizontal. Cambia de orden las filas de `matriz` (la primera pasa a ser la última y la última la primera, la segunda la penúltima y la penúltima la segunda, etc.).

o) `void EspejoVertical (Matriz2D & matriz);`

Espejo vertical. Cambia de orden las columnas de `matriz` (la primera pasa a ser la última y la última la primera, la segunda la penúltima y la penúltima la segunda, etc.).

Escriba dos funciones básicas que usará para la implementación de las funciones enumeradas pero que **no** ofrecerá en la biblioteca (serían algo así como funciones *privadas*):

`Matriz2D ReservaMemoria (int nfils, int ncols);`

Reservar memoria. Reserva memoria para *los datos* de una matriz dinámica con `nfils` filas y `ncols` columnas. El contenido de casillas (accesibles desde el puntero `datos`) queda INDEFINIDO. Tanto `nfil` como `nols` deben ser estrictamente positivos para poder disponer de una matriz no vacía. Si alguno de los dos valores fuera 0 no se reserva memoria, y la matriz queda vacía.

`void LiberaMemoria (Matriz2D & matriz);`

Liberar memoria. Libera la memoria ocupada por la matriz dinámica. La matriz queda vacía (todos sus campos a cero).

Modularice la solución en dos ficheros: `Matriz2D.h` y `Matriz21.cpp` y escriba un fichero con una función `main` que ilustre el uso de las funciones.

8. Supongamos que ahora decidimos utilizar una forma diferente para representar las matrices bidimensionales dinámicas a la que se propone en el ejercicio 7. Usaremos una estructura semejante a la que aparece en la figura 28 (tipo `Matriz2D`, 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.

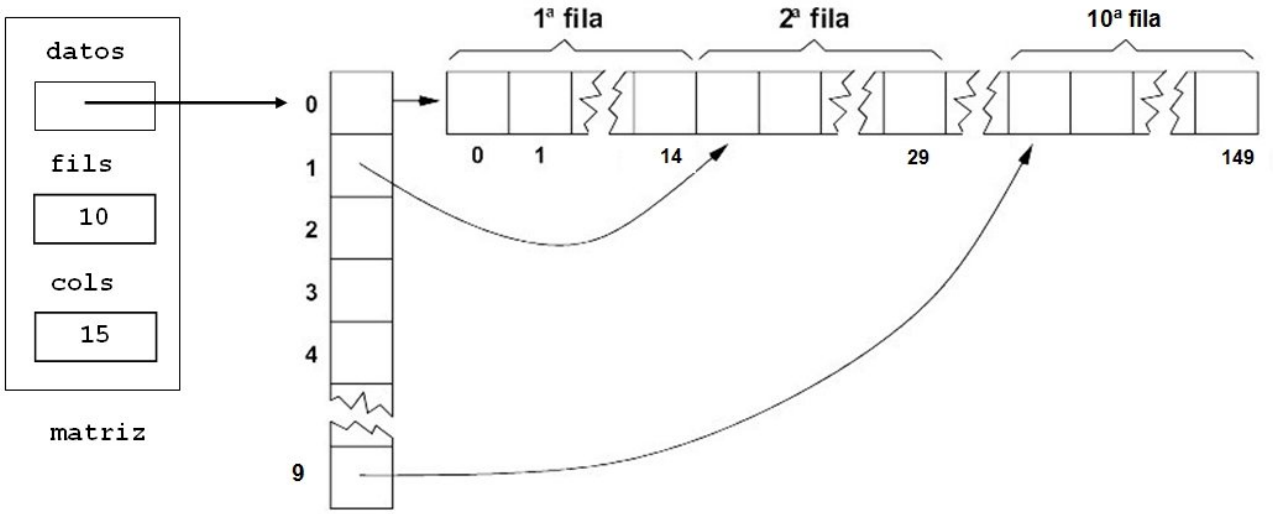


Figura 28: Tipo `Matriz2D_2`: datos guardados en una sola fila

Reescribir las funciones propuestas en el ejercicio 7.

9. *(Examen ordinario FP 2020)* **Relieve**.

   En este ejercicio reutilizará la solución a uno de los ejercicios de un examen de la asignatura **Fundamentos de Programación**. Encontrará el enunciado completo y la solución (de la pregunta) en PRADO.

   En la solución al problema intervienen las clases: `Relieve`, `Punto2D` y `ColeccionPuntos2D`. La solución que encontrará en PRADO está escrita, lógicamente, con las restricciones impuestas por el temario de la asignatura **Fundamentos de Programación**. Le pedimos que modifique esa solución usando los nuevos conocimientos adquiridos en **Metodología de la Programación**:

   - Modularice la solución en ficheros de manera que el código fuente de cada clase se organiza en dos ficheros (`.h` y `.cpp`). Escriba en un fichero independiente la función `main`.
   - Escriba un nuevo constructor para la clase `Relieve` que recibe una matriz dinámica:

     ```
     Relieve :: Relieve (const Matriz2D & m);
     ```

     Crea un objeto `Relieve` con los datos de la matriz dinámica `m`.

- En la función `main` de la solución aportada puede ver cómo se lee de `cin` el contenido de los objetos `Relieve`:

```
// Leer num. de filas y columnas del relieve
cin >> fils_r;  // filas
cin >> cols_r;  // columnas

// Constructor (todas las casillas a 0 )
Relieve r (fils_r, cols_r);

// Leer los valores de altura del objeto r
for (int f=0; f<r.FilasUtilizadas(); f++)
   for (int c=0; c<r.ColumnasUtilizadas(); c++) {
      cin >> valor_altura;
      r.ModificarAltura(f, c, valor_altura);
   }
```

  Proponemos que ahora, para cada uno de los objetos `Relieve`, una vez leidos el número de filas (`fils_r`) y columnas (`cols_r`):

  1) cree una matriz dinámica con esas dimensiones,
  2) lea de `cin` los `fils_r` $\times$ `cols_r` valores de altura, y guárdelos en la matriz dinámica recién creada,
  3) use el nuevo constructor para construir el objeto `Relieve` y finalmente
  4) "destruya" la matriz dinámica.

  Una vez creados (y "rellenos") los objetos, el resto del código proporcionado es válido.

El resto del código de la función `main` debería poder ejecutarse sin problemas.

Use las funciones de los ejercicios 7 ó 8 para la gestión de matrices dinámicas.

10. *(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 $0 < b_{ij} \leq 100$.

    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**.

11. *(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**.

12. En el ejercicio 11 se calculaba la *mejor* ruta a partir de una ciudad que se especificaba en la ejecución del programa.

    Escribir una solución que calcule la mejor ruta global partiendo desde todas y cada una de las ciudades. Mostrará la mejor ruta calculada (su itinerario y coste).

13. (*Este ejercicio está basado en el ejercicio 30 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.

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

    Notas finales (ver ejercicio 30):

    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`.

14. (*Este ejercicio está basado en el ejercicio 32 de la Relación de Problemas I*)

    En el ejercicio 32 de la Relación de Problemas I proponíamos usar una matriz bidimensional de datos `char` para guardar una colección de secuencias de ADN. Solo podíamos reservar espacio a priori, sin conocer cuántas secuencias se iban a guardar ni su longitud. En ese caso se reservaba espacio para un máximo de 2000 secuencias de ADN en la que cada secuencia podía almacenar un máximo de 100 nucleótidos (101 casillas realmente por el carácter `'\0'`). Esta reserva era común para todas las colecciones de secuencias y no podía modificarse (los valores estaban establecidos en tiempo de compilación). Si se fija en el ejemplo proporcionado (10 secuencias de 12 caracteres -13 realmente, por el carácter `'\0'`-) se ocupan únicamente $\mathbf{10 \times 13 = 130}$ casillas de las $\mathbf{2000 \times 101 = 202000}$ (el $\mathbf{0{,}06\,\%}$ de las casillas disponibles).

    Modifique la estructura de datos usada para la colección de secuencias de ADN de manera que los datos se guarden en memoria dinámica, ocupándose los estrictamente necesarios.

    Para que los cambios en el código desarrollado para la resolución del ejercicio 32 de la *Relación de Problemas I* sean mínimos recomendamos usar una estructura de datos similar a la que esquematizamos en la figura 29.

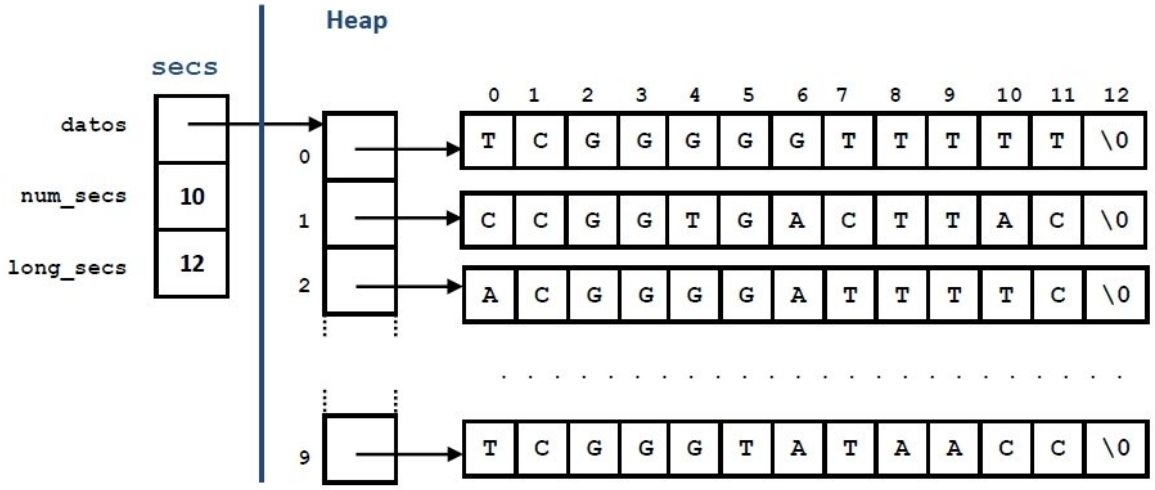


Figura 29: `secs` es un dato de tipo `ColeccionSecuenciaADN`

Reescriba la solución del ejercicio 32 declarando el tipo de datos `ColeccionSecuenciaADN` (en la figura 29 `secs` es un dato de tipo `ColeccionSecuenciaADN`).

El programa pedirá la longitud de las secuencias, creará y rellenará **dos** colecciones de secuencias. Después calculará y mostrará la secuencia de consenso de cada colección e informará si son iguales. Finalmente calculará y mostrará las secuencias inconexas.

El número de secuencias de una colección no está predeterminada[3]. Todas las secuencias deberán tener la misma longitud[4].

[3] La lectura de una colección de secuencias de ADN finalizará cuando se introduce `FIN`

[4] Deberá comprobarlo tras leer cada secuencia. Se descartarán las secuencias con otra longitud

Ejercicios sobre listas

15. Para trabajar con listas enlazadas de datos de tipo `TipoBase` emplearemos la biblioteca `libLista.a` que nos ofrecerá (a través de `Lista.h`) los *tipos de datos* y las *funciones* para la gestión de listas enlazadas.

    En la biblioteca incluiremos una serie de tareas comunes (crear una lista y destruirla, inicializarla aleatoriamente, contar el número de nodos, añadir, insertar, eliminar valores etc.) que se repiten en muchos ejercicios y que podrían ser útiles en otros programas que gestionaran listas. Esta biblioteca se enlazará con el fichero objeto que contiene la función `main` que resuelve cada uno de los ejercicios indicados y se construirá el correspondiente ejecutable.

    En el fichero de cabecera `Lista.h` escribirá la declaración de los tipos `Nodo` ,`PNodo` y `Lista`, y los prototipos de las funciones que se ofertarán en la biblioteca. En `Lista.cpp` escribirá la implementación de las funciones. Los tipos `PNodo` 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; // Tipo de los datos almacenados

// Cada nodo de la lista es de tipo "Nodo"
struct Nodo {
   TipoBase info;
   Nodo *sig;
};

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

    La versión básica de la biblioteca ofrece la funcionalidad que enumeramos a continuación. La lista de funciones se incrementará en otros ejercicios. Escriba las **precondiciones** que considere razonables en cada función.

    a) `Lista CreaLista (int num_nodos=0, TipoBase valor=VALOR_DEF);`

    Crear una lista. Crea una lista enlazada con `num_nodos` nodos. Inicializa todos los nodos a un valor común, el indicado en el parámetro `valor` (`0` si `TipoBase` es `int`, `0.0` si `TipoBase` es `double`, ...). La funcionalidad sería similar a la de un *constructor*.

    b) `void DestruyeLista (Lista & l);`

    Destruir lista. "Destruye" una lista y la deja en un estado no útil (vacía). La tarea de esta función sería similar a la de un *destructor*.

c) `void Ecualiza (Lista & l, TipoBase valor=VALOR_DEF);`

Ecualizar. Cambia todos los valores de la lista `l` y los fija todos iguales e iguales a `valor`.

d) `int NumElementos (const Lista & l);`

Número de nodos. Devuelve el número de nodos de la lista `l` (el número de valores guardados en la lista).

e) `bool EstaVacia (const Lista & l);`

¿Vacía?. Devuelve `true` si la lista está vacía.

f) `string ToString (const Lista & l);`

Serialización. Devuelve un `string` con el resultado de "serializar" una lista.

g) `TipoBase & Valor (const Lista & l, int pos);`

Consulta ó modifica el valor de un nodo dado por su posición. Si se utiliza como *rvalue* se emplea para consultar el valor del nodo de posición `pos`. Si se utiliza como *lvalue* se emplea para modificar el valor del nodo de la posición `pos`. Las posiciones deben cumplir la precondición: $1 \leq$ `pos` $\leq$ `NumElementos(l)`. El valor 1 indica la posición del primer nodo, y `NumElementos(l)` indica la posición del último nodo.

h) `void  EliminaTodos (Lista & l);`

Elimina todos los valores. Elimina todos los valores de la lista. Al finalizar, `NumElementos(l)==0` (`EstaVacia(l)==true`). La lista, aunque vacía, sigue "activa".

i) `void Clona (Lista & destino, const Lista & origen);`

Copia profunda. Hace una copia **profunda** de `origen` en `destino`.

j) `bool SonIguales (const Lista & una, const Lista & otra);`

Comparación. Devuelve `true` si las listas `una` y `otra` son exactamente iguales (dimensiones y contenidos).

k) `void SubLista (Lista & resultado, const Lista & original, int pos_inic, int num_nodos);`

Sublista. Extrae una lista de `original` y la deja en `resultado`. La lista `resultado` es una copia de una parte de `original` formada por un máximo de `num_nodos` nodos, formada a partir del nodo que ocupa la posición `pos_inic` de `original`.
Considere todos los posibles casos entre `num_nodos`, `pos_inic` y el número de nodos de `original` y resuélvalos de la mejor manera posible. Por ejemplo,

- Si la lista original tiene 5 nodos, y le pedimos una sublista de 8 nodos desde la posición 3 sólo disponemos de tres (posiciones 3, 4 y 5). La función construirá y rellenará una lista de 3 nodos.
- Sobre la misma lista, si pedimos una sublista de 6 nodos desde la posición 8, no podremos porque la posición inicial no está dentro de la lista. La función construirá y "rellenará" una lista vacía. Lo mismo ocurre si la posición inicial fuera, por ejemplo,-1.

l) `void Aniade (Lista & l, TipoBase valor);`

Añadir. Añade (al final de la lista) un nodo con el valor indicado en `valor`. El nuevo nodo será el último de la lista tras la adición.

Escriba además, dos funciones básicas que usará para la implementación de algunas de las funciones de la biblioteca, pero que **no** ofrecerá en la biblioteca (serían algo así como funciones *privadas*):

```
Lista ReservaMemoria (int num_nodos);
```

Reservar memoria. Reserva memoria para una lista enlazada con `num_nodos` nodos. El contenido de los nodos queda INDEFINIDO. El valor de `num_nodos` debe ser positivo. Si no fuera así la lista quedaría vacía.

```
void LiberaMemoria (Lista & l);
```

Liberar memoria. Libera la memoria ocupada por la lista `l`. La lista queda vacía.

Finalmente, escriba una función `main` que permita probar las funciones de la biblioteca.

16. Escriba un programa que lea una secuencia indefinida de valores `double` (termina si se introduzca un valor negativo). Los valores leídos (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: el *número de datos* almacenados, su *media* y su *varianza*.

    Después, creará dos nuevas listas: una con los valores pares y la otra con los valores impares de la lista. Finalmente, con cada una de las listas muestre los valores almacenados y calcule el número de datos que la compone y calcule su media y su varianza.

    Para efectuar los cálculos indicados en este problema concreto 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 ¿por qué?. Respecto al cálculo de las listas con pares/impares ¿es interesante disponer de sendas funciones que devuelvan una lista con los pares/impares? ¿es eficiente? ¿las incluiría en la biblioteca? ¿Por qué?.

17. En este ejercicio vamos a trabajar con lista enlazadas que se inicializan con valores aleatorios.

    a) Añada a la biblioteca una función que crea una lista con números aleatorios:

    ```
    Lista CreaListaAleatoria (int num_nodos=0,
                              int min=0, int max=100);
    ```

    Crear una lista aleatoria. Crea una lista enlazada con `num_nodos` nodos. Inicializa los nodos con valores aleatorios comprendidos entre `min` y `max`. La funcionalidad sería similar a la de un *constructor*.

    b) Escriba un programa que reciba de la línea de órdenes el número de valores que formará la lista y el mínimo y máximo valor aleatorio (todos los argumentos son opcionales).
    El programa creará la lista, mostrará su contenido y calculará su *media* y su *varianza*.

    c) Añada una nueva función a la biblioteca, a la que llamará `RellenaAleatoriamente` para cambiar todos los valores de una lista **existente** con números aleatorios comprendidos entre dos extremos dados. Decida su cabecera.

    d) Añada al programa la llamada a esta función para que modifique el contenido de la lista anterior por nuevos valores aleatorios. Muestre su contenido y calcule su *media* y su *varianza*.

18. Añada las siguientes funciones a la biblioteca. Las funciones modifican el tamaño de la lista trabajando en una posición dada (siempre que la posición indicada sea correcta).

    Las posiciones (funciones `Elimina` e `Inserta`) deben cumplir la precondición:
    $1 \leq$ `pos` $\leq$ `NumElementos(l)`. El valor 1 indica la posición del primer nodo, y `NumElementos(l)` indica la posición del último nodo.

    ```
    void Elimina (Lista & l, int pos);
    ```

    Eliminar. Elimina el nodo que ocupa la posición `pos`.

    Para la función `Inserta` las posiciones deben cumplir la precondición: $1 \leq$ `pos`.
    Si `pos` $>$ `NumElementos(l)` la operación de *inserción* se convierte en *adición*.

    ```
    void Inserta (Lista & l, TipoBase valor, int pos);
    ```

    Insertar. Inserta en la posición `pos` un nodo con el valor indicado en `valor`. El nuevo nodo ocupará la posición `pos`.

    Ejemplos: Antes de insertar en la lista, $l =$ <6, 8, 4, 3, 2, 9 >

    Después de `Inserta(l, 5, 2)` $l =$ <6, 5, 8, 4, 3, 2, 9 >
    Después de `Inserta(l, 5, 6)` $l =$ <6, 8, 4, 3, 2, 5, 9 >
    Después de `Inserta(l, 5, 7)` $l =$ <6, 8, 4, 3, 2, 9, 5 >
    Después de `Inserta(l, 5, 1)` $l =$ <5, 6, 8, 4, 3, 2, 9 >

    Para probar estas funciones escriba un programa que rellene una lista aleatoriamente y la invierta. La inversión debe realizarse sobre la propia lista, no puede usar otra lista ni vector, ni array, ...

19. 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.).

    Los métodos de ordenación basados en selección e intercambio podrán resolverse sin necesidad de intercambiar *nodos*, esto es, bastará con el intercambio de los *valores*. El método de ordenación por inserción, no obstante, tendrá que resolverse *reorganizando los nodos*.

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

```
bool EstaOrdenada (const Lista l);
void OrdenaSeleccion (Lista &l);
void OrdenaInsercion (Lista &l);
void OrdenaIntercambio (Lista &l);
void OrdenaIntercambioMejorado (Lista &l);
```

20. 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.

```
void InsertaValorListaOrdenada (Lista &l, TipoBase v);
```

    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.

```
void EliminaValorListaOrdenada (Lista &l, TipoBase v);
```

    Para probar estas funciones escriba un programa que:

    1) Cree una lista con 10 números aleatorios (valores entre 1 y 100) y la ordene.
    2) Cree otra lista con 5 números aleatorios (valores entre 1 y 100) y la ordene.
    3) Inserte ordenadamente los valores de la segunda lista en la primera.
    4) Elimine de la lista obtenida en el paso 3 todos los valores que estén en la segunda.

    Finalmente, compruebe que la lista resultante sea igual que la primera (ordenada).

    Nota: La aleatoriedad puede hacer que no se prueben todos los casos posibles. Asegúrese de hacer todas las pruebas que garanticen que se comprueban todos los casos posibles de inserción y borrado.

21. Considere dos secuencias de datos **ordenadas** almacenadas en sendas *listas*. Implemente una función para *mezclar ordenadamente* las dos secuencias en una nueva. Implemente dos versiones:

    a) La versión “clásica” consiste en adaptar el algoritmo de mezcla conocido. Las listas originales se mantienen si modificaciones.

```
void MezclaListasClasico (Lista &l,
                          const Lista &l1, const Lista &l2);
```

b) La nueva versión consiste en "mover" los nodos de las lista originales hacia la lista resultante en lugar de copiarlos. De esta forma las dos listas originales quedan *vacías* tras realizar la mezcla y la lista resultante contiene todos los datos.

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

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

22. (*Este ejercicio está basado en el ejercicio 5 de esta Relación de Problemas*)

Modifique el ejercicio 5 de esta Relación de Problemas para que las cadenas leidas se guarden en una **lista** en lugar de un vector dinámico. Realice los mismos cálculos (número de líneas, número de líneas vacías y número de párrafos).

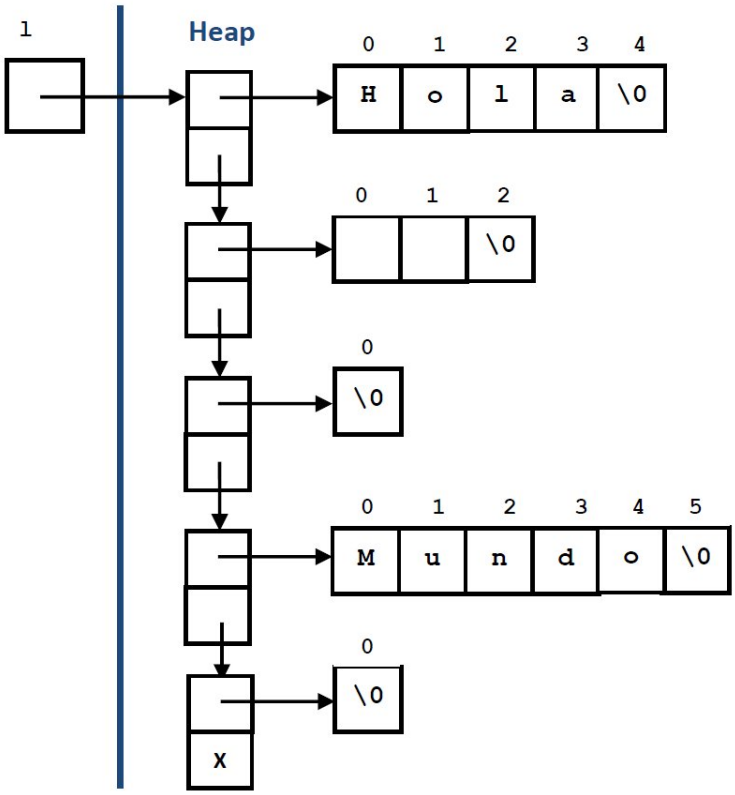


Figura 30: Cada línea se guarda en una cadena clásica, accesible desde un nodo de `l`

No es preciso que implemente (de nuevo) la clase `Lista` para este caso particular (*lista de cadenas*) con todas las funciones ya escritas para la lista de datos `TipoBase`. Bastará con implementar las funciones que requiera para la solución del ejercicio.

23. (*Este ejercicio está basado en el ejercicio 14 de esta Relación de Problemas*)

    En el ejercicio 14 de esta Relación de Problemas proponíamos un vector dinámico de cadenas clásicas (una matriz dinámica de datos `char`) para guardar una colección de secuencias de ADN. Esta representación permitía reservar la memoria estrictamente necesaria para los datos que se iban a procesar ya que tanto el array dinámico de punteros como las cadenas clásicas se almacenaban en el Heap. Además, podíamos acceder a cada una de las casillas de las secuencias de ADN con notación de doble corchete, lo que simplifica el código.

    Reescriba la solución al mismo problema empleando una lista enlazada de cadenas clásicas de manera que cada nodo nos permita acceder a una cadena clásica que guarda una secuencia de ADN (en el nodo se guarda, realmente, el puntero que permite acceder a la cadena clásica). En la figura 31 mostramos un ejemplo de esta estructura de datos.

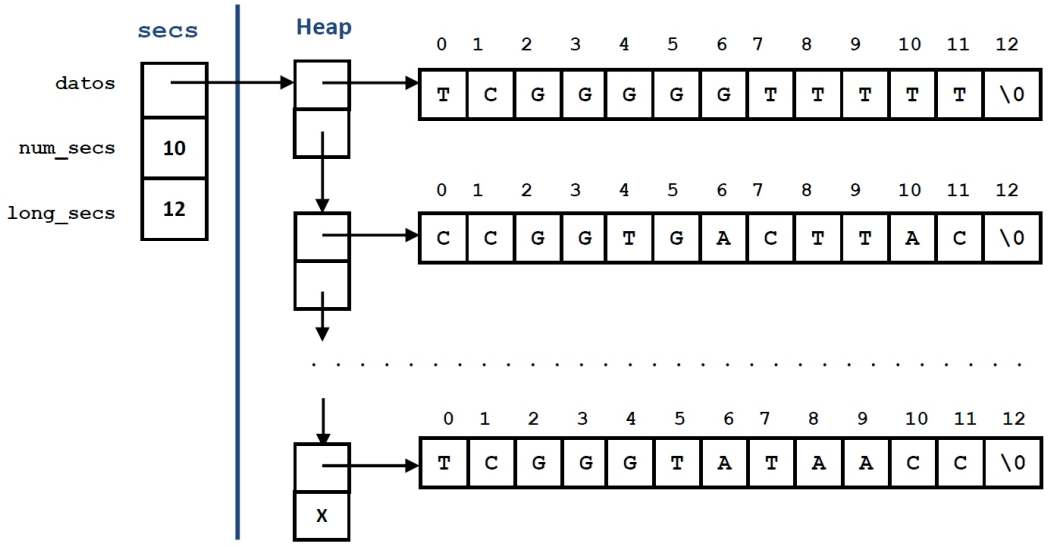


Figura 31: `secs` es un dato de tipo `ColeccionSecuenciaADN`

Reescriba la solución del ejercicio 14 declarando el **nuevo** tipo de datos `ColeccionSecuenciaADN` (en la figura 31 `secs` es un dato de ese tipo).

Al igual que el ejercicioo 14 el programa pedirá la longitud de las secuencias, creará y rellenará **dos** colecciones de secuencias. Después calculará y mostrará la secuencia de consenso de cada colección e informará si son iguales. Finalmente calculará y mostrará las secuencias inconexas. El número de secuencias de una colección no está predeterminada: se conoce cuando finaliza la lectura de las secuencias de una colección[5]. Todas las secuencias deberán tener la misma longitud[6].

[5] La lectura de una colección de secuencias de ADN finalizará cuando se introduce `FIN`

[6] Deberá comprobarlo tras leer cada secuencia. Se descartarán las secuencias que tengan una longitud diferente

24. 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.

    ...

Figura 32: Un sistema real de fila única

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

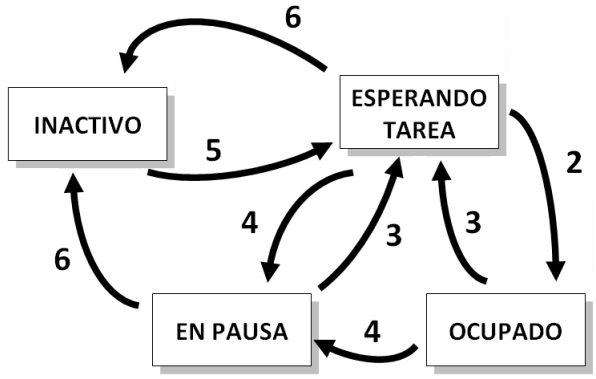


Figura 33: Estados posibles de los recursos y transiciones permitidas

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 y permita evaluar la funcionalidad de la clase.

---

1. Implementar la **clase** `Secuencia` para trabajar con secuencias -colecciones- de datos de tipo `TipoBaseSecuencia`. La funcionalidad de esta clase será idéntica a la que hemos utilizado en la asignatura *Fundamentos de Programación* pero en este caso los datos se alojan en el Heap en un vector dinámico y no hay una capacidad máxima predeterminada y el espacio asignado a los datos crece/decrece de acuerdo a las especificaciones que se indican en el enunciado.

   Proponer una **representación** para la clase (*propiedades* ó *datos privados* y cómo se almacenan en memoria) e implementar los siguientes **métodos**:

   a) **Constructor** sin argumentos, que crea una secuencia *vacía*. El número máximo de casillas disponibles (la capacidad) está establecido en la constante `TAMANIO`. Una secuencia vacía no tiene datos útiles, independientemente del número de casillas reservadas.

   b) **Constructor** con un argumento, que crea una secuencia *vacía* con una capacidad indicada en el argumento.

   c) **Constructor de copia**.

   d) **Destructor**.

   e) Método que *consulta* si la secuencia está vacía.

   ```
   bool EstaVacia(void) const;
   ```

   f) Métodos para *consultar* el número de casillas ocupadas y reservadas, respectivamente.

   ```
   int TotalUtilizados (void) const;
   int Capacidad (void) const;
   ```

   g) Método para *eliminar todos los valores* de la secuencia.

   ```
   void EliminaTodos (void);
   ```

   La secuencia quedará vacía y la capacidad de la secuencia se reajusta al valor indicado en la constante `TAMANIO`.

h) Método para "*serializar*" una secuencia.

```
string ToString (void);
```

i) Método para *acceder* a un elemento de la secuencia, dada una posición. El método puede actuar como *lvalue* ó *rvalue*.

```
TipoBaseSecuencia & Valor (int indice);
```

j) Método para hacer una **copia profunda** desde la secuencia explícita `otra` en la secuencia implícita.

```
void Clona (const Secuencia & otra);
```

k) Método para *comparar* dos secuencias. Devuelve `true` si las secuencias implícita y `otra` son exactamente iguales (dimensiones y contenidos).

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

l) Método para *añadir* un valor (siempre al final).

```
void Aniade (TipoBaseSecuencia nuevo);
```

El método se encargará de redimensionar el vector dinámico de datos si no tuviera espacio disponible. Realmente pedirá memoria adicional si el número de casillas usadas es superior al `PORC_PETICION` (p.e. 75 %) de la capacidad actual.

Si tuviera que redimensionarse por crecimiento, el número de casillas adicionales no será fijo, sino el `PORC_CRECIMIENTO` (p.e. 10 %) de la capacidad actual. En cualquier caso, se establece un valor mínimo de `MIN_CRECIMIENTO` (p.e. 5) casillas para la redimensión.

m) Método para *insertar* un valor en una posición dada.

```
void Inserta (int indice, TipoBaseSecuencia valor);
```

El método se encargará de redimensionar el vector dinámico de datos si no tuviera espacio disponible (exactamente como se hace en el método `Aniade`).

n) Método para *eliminar un valor*, el valor que ocupa la posición `indice` de la secuencia.

```
void Elimina (int indice);
```

Una consecuencia de la eliminación podría ser la reducción de la capacidad de la secuencia. Esto ocurrirá cuando el número de casillas usadas es inferior al `PROC_REDUCCION` (p.e. 50 %) de la capacidad actual.

Si tuviera que reajustarse por reducción, el número de casillas finales no será fijo, sino que será un `PORC_EXTRA` (p.e. 20 %) superior a la ocupación de la secuencia después de la eliminación.

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

Sugerencia: Podría usar la solución del ejercicio 6 de la *Relación de Problemas II* cambiando la clase `SecuenciaEnteros` por la nueva clase `Secuencia` donde `TipoBaseSecuencia` sea `int`.

2. Implementar la **clase** `Matriz2D` para elementos de tipo `TipoBaseMatriz2D` y escribir una función `main` que permita probar todos los métodos de la clase.

   Proponga una **representación** para la clase (puede basarse en la idea del `struct` propuesto en la *Relación de Problemas II*) e implemente 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 que *consulta* si la matriz está vacía.

   ```
   bool EstaVacia(void) const;
   ```

   h) Métodos de *consulta* de las dimensiones de la matriz.

   ```
   int NumFilas (void) const;
   int NumColumnas (void) const;
   ```

   i) Método para *eliminar todos los valores* de la matriz.

   ```
   void EliminaTodos (void);
   ```

   La matriz quedará vacía.

   j) 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 `VALOR_DEF`.

   ```
   void Ecualiza (TipoBaseMatriz2D valor=VALOR_DEF);
   ```

   k) Método para "*serializar*" una matriz.

   ```
   string ToString (void);
   ```

   l) Método para *acceder* a un elemento de la matriz, dadas sus coordenadas (índices). El método puede actuar como *lvalue* ó *rvalue*.

   ```
   TipoBaseMatriz2D & Valor (int fila, int col);
   ```

   m) Método para hacer una **copia profunda** desde la matriz explícita `otra` en la matriz implícita.

   ```
   void Clona (const Matriz2D & otra);
   ```

   n) Método para *comparar* dos matrices. Devuelve `true` si las matriz implícita y `otra` son exactamente iguales (dimensiones y contenidos).

   ```
   bool EsIgualA (const Matriz2D & otra) const;
   ```

ñ) Método para *añadir una fila*. La fila nueva (una dato `Secuencia`) debe tener el mismo número de casillas que columnas tenga la matriz.

```
void Aniade (const Secuencia & fila_nueva);
```

o) Método para *insertar una fila* en una posición dada. La fila nueva (una dato `Secuencia`) debe tener el mismo número de casillas que columnas tenga la matriz. La posición indicada será la posición que tendrá la fila *después* de la inserción.

```
void Inserta (int indice, const Secuencia & fila_nueva);
```

p) Métodos para *eliminar una fila/columna*. El número de la fila/columna que se va a a eliminar está indicada por el argumento *indice*:

```
void EliminaFila (int indice);
void EliminaColumna (int indice);
```

q) Métodos que *devuelven un objeto de otra clase*. Se trata de obtener una fila/columna completa. El objeto devuelto será de la clase `Secuencia`:

```
Secuencia Fila (int indice);
Secuencia Columna (int indice)
```

r) Método que *devuelve un objeto de la clase*. Se trata de obtener una submatriz de la matriz implícita y devolverla. La submatriz es una zona rectangular de la matriz implícita que empieza en la casilla de coordenadas (`fila_inic`, `col_inic`) y que tiene (un máximo de) `num_filas` filas y `num_cols` columnas.

```
Matriz2D SubMatriz (int fila_inic, int col_inic,
                    int num_filas, int num_cols);
```

Considere todos los posibles casos entre `fila_inic`, `col_inic`, `num_filas`, `num_cols` y el número de filas y columnas de `original` y resuélvalos de la mejor manera posible (consulte el problema 7*l* de la *Relación de Problemas II*)

3. Implementar la **clase** `Lista` para trabajar con listas enlazadas (de tamaño arbitrario no definido a priori cuyos nodos residen en el *heap*) de datos de tipo `TipoBaseLista`. Escribir una función `main` que permita probar todos los métodos de la clase.

   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 que *consulta* si la lista está *vacía*.

g) Método que *consulta* el número de nodos de la lista.

h) Método para *eliminar todos los valores* de la lista. La lista quedará vacía.

i) Método que *inicializa* todos los nodos de la lista al valor indicado como argumento. Si no se especifica ninguno, inician todos los nodos al valor `VALOR_DEF`.

j) Método para "*serializar*" una lista.

k) Método para *acceder* a un elemento de la lista, dada su posición. El método puede actuar como *lvalue* ó *rvalue*.

El criterio para referirnos a la posición de un nodo es el mismo que seguimos en el ejercicio 15*g* de la *Relación de Problemas II*: se usa un número entero de tal manera que 1 indica el primer nodo, 2 el segundo, etc.

l) Método para hacer una **copia profunda**.

m) Método para *comparar* dos listas. Devuelve `true` si las dos listas son exactamente iguales (dimensiones y contenidos).

n) Método para *insertar* un valor en la lista. Responderá al siguiente prototipo:

```
void Inserta (TipoBaseLista 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.

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

Antes: $< 6, 8, 4, 3, 2, 9 >$ `Inserta (5, 2)` Después: $< 6, 5, 8, 4, 3, 2, 9 >$
Antes: $< 6, 8, 4, 3, 2, 9 >$ `Inserta (5, 6)` Después: $< 6, 8, 4, 3, 2, 5, 9 >$
Antes: $< 6, 8, 4, 3, 2, 9 >$ `Inserta (5, 7)` Después: $< 6, 8, 4, 3, 2, 9, 5 >$
Antes: $< 6, 8, 4, 3, 2, 9 >$ `Inserta (5, 1)` Después: $< 5, 6, 8, 4, 3, 2, 9 >$

ñ) Método para *eliminar un nodo* en la lista. Responderá al siguiente prototipo:

```
void Elimina (int pos);
```

El criterio para las posiciones es el mismo que en el método `Inserta`.

o) Método para *añadir un valor* en la lista. La adición siempre se hace al final de la lista.

p) Método que *devuelve un objeto de la clase*. Se trata de obtener una sublista de la lista implícita y devolverla.

```
Lista SubLista (int pos_inic, int num_nodos);
```

Considere todos los posibles casos entre `num_nodos`, `pos_inic` y el número de nodos de `original` y resuélvalos de la mejor manera posible (consulte el problema 15*k* de la *Relación de Problemas II*)

4. 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** (**l**ast **i**n **f**irst **o**ut).

   La *representación* para la clase estará basada en el almacenamiento de los nodos en memoria dinámica (lista). 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*.

   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. Se trabajará sobre un menú que proporciona las opciones: a) Añadir un valor, b) Extraer un valor, c) Consultar el elemento del *tope* y d) Mostrar el estado de la pila.

5. 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).

   La *representación* para la clase estará basada en el almacenamiento de los nodos en memoria dinámica (lista). 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. Se trabajará sobre un menú que proporciona las opciones: a) Añadir un valor, b) Extraer un valor, c) Consultar el elemento de la *cabecera* y d) Mostrar el estado de la cola.

6. Implementaciones alternativas de las clases `Pila` y `Cola`.

   La implementación de las clases `Pila` y `Cola` de los ejercicios 4 y 5 se han realizado usando una lista *ad-hoc*, implementando los métodos usando punteros adaptando (a veces copiando) métodos que se habían empleado para la clase `Lista`.

   Deberá reescribir las clases `Pila` y `Cola` usando la clase `Lista` como base, es decir:

a) Un objeto `Pila` o `Cola` tendrá un único campo -privado- de tipo `Lista`. Los valores de la pila se guardan realmente en una lista.

b) Todos los métodos de las clases deben reescribirse usando los métodos de la clase `Lista`.

Por ejemplo, si decidimos que el elemento que está en el tope de la lista siem,pre será el primero de la lista, el método que devuelve el elemento del *tope* de la lista (apartado 4*g* de esta Relación de Problemas) se podría escribir:

```
TipoBasePila Pila :: Tope (void) const
{
   if (datos.Tamanio() > 0)
      return (datos.Valor(1));
}
```

suponiendo que `datos` sea un campo -privado- de tipo `Lista`.

7. *(Versión adaptada del Examen Ordinario MP 2018/2019)*

   Se desea construir la clase `RedMetro` para poder realizar simulaciones sobre una red de transporte metropolitano. La **red** está compuesta por **líneas**, y cada línea es una sucesión de **paradas**. Ni el número de líneas ni el número de paradas está predeterminado. Algunas paradas pueden ser comunes a distintas líneas. Una parada se identifica por un número (*índice*) y puede estar activa -admite el tráfico- para una(s) determinada(s) línea(s) e inactiva (no admite el tráfico) para la(s) otra(s).

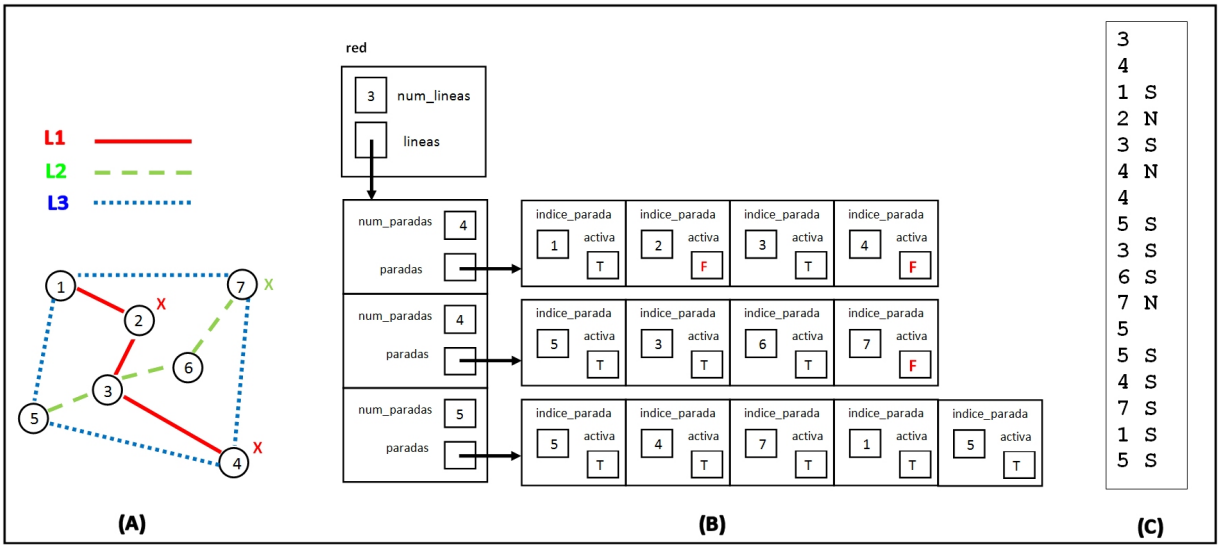
   Se propone la siguiente representación para las clases:

```
class RedMetro {
  private:
    int num_lineas;
    Linea * lineas;
  public:
    // ... interfaz
};

class  Linea {
  private:
    int num_paradas;
    InfoParada * paradas;
  public:
    // ... interfaz
  };

class  InfoParada {
   private:
      bool activa;
      int indice_parada;
   public:
     // ... interfaz
};
```

   En la figura A mostramos una red formada por tres líneas y siete paradas. Esta red se muestra en la figura B representada en el objeto `red` (clase: `RedMetro`). Las líneas 1 y 2 tienen 4 paradas mientras que la línea 3 tiene 5 (una línea circular *duplica* la parada de cabecera). Las líneas 1 y 2, por ejemplo, comparten la parada 3. En la línea 1 la parada 4 está fuera de servicio, mientras que en la línea 3 esa misma parada está activa.



Para la realización de los ejercicios propuestos deberá escribir `RedMetro.h`, `Linea.h`, e `InfoParada.h`. Deberá implementar:

a) Métodos básicos de las clases.

- Clase `RedMetro`: Constructor sin argumentos (crea una *red vacía*), constructor de copia y destructor. Escriba también el método `EstaVacia` que indique si una red está vacía.
- Clase `Linea`: Constructor sin argumentos (crea una *línea vacía*), constructor de copia y destructor. Escriba también el método `EstaVacia` que indique si una línea está vacía.
- Clase `InfoParada`: Constructor sin argumentos y constructor con argumentos. ¿Por qué no es preciso un constructor de copia ni un destructor?
- Implemente la **copia profunda** (método `Clona`) para las clases `RedMetro` y `Linea`. ¿Por qué no es preciso para la clase `InfoParada`?

b) Métodos de consulta de las clases.

- Clase `RedMetro`: a) `GetNumLineas`, que devuelve el número de líneas, b) `GetNumeroTotalParadas`, que devuelve el número **total** de paradas (en el ejemplo, 7) y c) `GetLinea`, que devuelve una línea.
- Clase `Linea`: a) `GetNumParadas`, que devuelve el número de paradas, b) `ContieneParada` para calcular si la línea contiene una parada dada y c) `GetInfoParada`, que devuelve una parada.
- Clase `InfoParada`: a) `GetIndice`, que devuelve el índice de la parada y b) `EstaActiva`, que indica si la parada está activa o no para la línea en la que se encuentra.

c) Métodos que modifican las clases:

- Escriba el método `AniadeLinea` para la clase `RedMetro`. El objetivo es añadir una nueva línea (un objeto `Linea`) a la red. El nuevo objeto `Linea` se añade **al final** del objeto `RedMetro`.
- Escriba el método `AniadeInfoParada` para la clase `Linea`. El objetivo es incorporar una nueva parada (un objeto `InfoParada`) a la línea. El nuevo objeto `InfoParada` se añade **al final** del objeto `Linea`.

d) Métodos para "serializar" y enviar a un flujo de salida el contenido de las clases.

- Escriba los métodos `ToString` y `ToText` para poder serializar el contenido de una red en formato texto (`ToString`) y así poder insertar su contenido en un flujo de salida como `cout` (`ToText`). En el caso de la clase `RedMetro` deberá aparecer (ver figura C):
  - Un número entero (número de líneas de la red) y un salto de línea,
  - Para cada línea:
    - Un número entero (número de paradas de la línea) y un salto de línea,
    - Tantas líneas de texto como paradas haya en la línea. En cada una aparecerá: un número entero (el *índice* de la parada), un carácter (`S` si está operativa ó `N` si no lo está) y un salto de línea.
- Escriba los métodos `ToString` y `ToText` para las otras clases. Para la implementación siga las pautas indicadas y modularice adecuadamente para evitar repetir código.

e) Métodos para "leer" objetos.

- Escriba el método `FromText` para poder leer de un flujo de entrada (`cin`) el contenido de una red. Los datos se proporcionan como se indica en la figura C.
- Escriba los métodos `FromText` para las otras clases. Para la implementación siga las pautas indicadas y modularice adecuadamente para evitar repetir código.

f) Métodos de cálculo.

- Implemente el método `MejorConectada` que permita obtener la parada (su *índice*) en la que confluyen el mayor número de líneas, independientemente de si las paradas están activas o no.
- Implemente el método `EstaTotalmenteOperativa` que indique si una línea está totalmente operativa, o sea, si todas sus paradas están activas.

Escriba una programa que lea una red de metro (recomendamos usar la redirección de entrada) y calcule:

a) la parada mejor conectada.

b) si todas las líneas están totalmente conectadas . Si alguna línea no lo estuviera, deberá indicar las paradas no activas.

8. *(Versión adaptada del Examen Ordinario MP Junio 2012. Primera parte)*

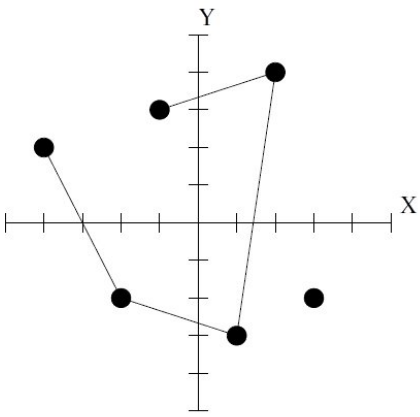
   Para poder gestionar figuras planas en una aplicación de gráficos 2D deseamos crear una estructura de datos capaz de representar adecuadamente esas figuras. La estructura escogida será la de una línea poligonal construida en base a una serie de puntos. Cada punto es un objeto de la clase `Punto2D`.

   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 Punto2D
{
   // La pareja (x,y) son las coordenadas de
   // un punto en un espacio 2D
   double x;
   double y;
   ......
};


class PoliLinea
{
   Punto2D * datos;  // Acceso al vector de datos Punto2D.
   int       usados; // Número de casillas usadas.

   // PRE: 0 <= usados
   ......
};
```



   Deberá implementar las bibliotecas `libPunto2D.a` y `libPoliLinea.a` para trabajar con objetos de las clases `Punto2D` y `PoliLinea`.

1. Métodos básicos

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

a) El **constructor** sin parámetros (crea una línea poligonal vacía).

b) El **constructor de copia**.

c) El **destructor**.

d) Método para realizar una **copia profunda**.

2. Métodos varios

a) Método `Valor`: Accede (tanto para lectura como para escritura) a un dato de tipo `Punto2D` en una `PoliLinea`.

b) Método `EsIgualA`: Compara dos objetos `PoliLinea` y nos informa si son iguales. 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).

c) Método `Aniade`: Recibe un objeto de la clase `Punto2D` y lo añade al final de la `PoliLinea`. El método devuelve un objeto *nuevo*. La `PoliLinea` que actúa como objeto implícito **no** se modifica.

d) Método `Concatena`: Recibe otro objeto de la clase `PoliLinea` y las concatena. El método devuelve un objeto *nuevo* y los datos `PoliLinea` usados para la concatenación no se modifican.

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

3. Métodos de E/S

a) Escriba el método `ToString` para poder *serializar* (creando un `string`) el contenido de una `PoliLinea`. Deberá contener:

- Un número entero (número de puntos de forman la `PoliLinea`) y un salto de línea,
- Tantas parejas de valores `int` como puntos tiene la `PoliLinea`. Los valores están separados por caracteres separadores. Cada pareja está en una línea.

b) Escriba el método `ToText` para poder insertar en un flujo de salida (`cout`) el contenido de una polilínea. Los datos se insertan como se ha detallado en el método `ToString`.
El resultado de `ToText` para el objeto `PoliLinea` mostrado en la figura podría ser:

```
5
-1   3
 2   4
 1  -3
-2  -2
-4   2
```

c) Escriba el método `FromText` para poder leer de un flujo de entrada (`cin`) el contenido de una polilínea. Los datos se proporcionan como se ha detallado en la descripción del método `ToString`.

d) Escriba los métodos `ToString`, `ToText` y `FromText` para la clase `Punto2D`. El método `ToString` aplicado al primer punto de la polilínea del ejemplo crearía una dato string con el contenido:

```
-1   3
```

4. Aplicación

Escribir un programa que lea una serie indeterminada de puntos y cree una dato `PoliLinea` a partir de ellos. Probar los operadores y funciones implementadas.

RELACIÓN DE PROBLEMAS IV. Clases (II)

Los ejercicios propuestos tienen como finalidad que el alumno practique con la **sobrecarga de operadores**. Los operadores que se van a sobrecargar son:

- operador de asignación =
- operadores de acceso `[]` y `()`
- operadores aritméticos
- operadores combinados
- operadores relacionales

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. Deberá evitar duplicar código y modularizar las operaciones de reserva y liberación de memoria, así como la **copia** del contenido de objetos con métodos **privados**. En este mismo sentido, deberá también escribir el código de algunos operadores basándose en el código escrito para otros operadores.
Deberá escribir para cada clase:

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

Finalmente, deberá proporcionar un fichero `.cpp` con la función `main` que contenga ejemplos sobre el uso de las clases.

---

1. Ampliar la funcionalidad de la **clase** `Secuencia` para trabajar con secuencias -colecciones- de datos de tipo `TipoBaseSecuencia`. La versión inicial se empezó a implementar en el problema 1 de la *Relación de Problemas III*. Deberá implementar los siguientes métodos:

   a) Sobrecarga del operador de asignación.

   b) Una sobrecarga alternativa del operador de asignación, que recibe como argumento un dato de tipo `TipoBaseSecuencia` e inicia **toda** la secuencia al valor especificado.

   c) Sobrecargar el operador `[]` para que sirva de operador de acceso a los elementos de la secuencia y pueda actuar de *lvalue* y *rvalue*. El índice hace referencia a la posición, de tal manera que 1 indica el primer elemento, 2 el segundo, etc.)

   d) Sobrecargar los operadores relacionales binarios == y `!`= para comparar dos secuencias. Dos secuencias 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 secuencias. Usar un criterio similar al que se sigue en la comparación de dos cadenas de caracteres clásicas.

f) Sobrecargar los operadores binarios + y - de manera que se apliquen: a) a dos secuencias, b) a una secuencia y un dato de tipo `TipoBaseSecuencia` y c) a un dato de tipo `TipoBaseSecuencia` y a una secuencia. En todos los casos se crea una *nueva* secuencia.

- El operador + cuando se aplica a dos secuencias crea una nueva, uniendo el contenido (completo) de la primera y el contenido (completo) de la segunda.
- El operador - cuando se aplica a dos secuencias crea una nueva, 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 (secuencia/secuencia, secuencia/valor y valor/secuencia) para los dos operadores.
- Los operandos no se modifican.

g) Sobrecargar los operadores combinados += y -= en los que el argumento explícito sea de tipo `TipoBaseSecuencia` y añadan o eliminen de la secuencia el valor dado por el argumento explícito.

- El nuevo valor se añade al final.
- 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.

2. Ampliar la funcionalidad de la **clase** `Matriz2D`de datos de tipo `TipoBaseMatriz2D`. La versión inicial se empezó a implementar en el problema 2 de la *Relación de Problemas III*. Deberá implementar los siguientes métodos:

a) Sobrecarga del operador de asignación.

b) Una sobrecarga alternativa del operador de asignación, que recibe como argumento un dato de tipo `TipoBaseMatriz2D` 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` 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 `TipoBaseMatriz2D`. En este caso ese valor indica una matriz con las mismas dimensiones que la otra y con todas sus casillas iguales e igual ese valor.

Importante: ninguno de los operandos se modifica.

f) Sobrecargar los operadores combinados += y -= de manera que el argumento explícito sea de tipo `TipoBaseMatriz2D` 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.

3. Ampliar la clase `Lista` de datos de tipo `TipoBaseLista` con los *métodos*:

   a) Sobrecarga del operador de asignación.

   b) Una sobrecarga alternativa del operador de asignación, que recibe como argumento un dato de tipo `TipoBaseLista` 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 binarios `+` y `-` de manera que se apliquen: a) a dos listas, b) a una lista y un dato de tipo `TipoBaseLista` y c) a un dato de tipo `TipoBaseLista` y a una lista. En todos los casos se crea una *nueva* lista.

   - El operador `+` cuando se aplica a dos listas crea una nueva, uniendo el contenido (completo) de la primera y el contenido (completo) de la segunda.
   - El operador `-` cuando se aplica a dos listas crea una nueva, 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.

   e) Sobrecargar los operadores combinados `+=` y `-=` en los que el argumento explícito sea de tipo `TipoBaseLista` 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.
   - 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.

   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.

4. Ampliar la clase `Pila` de datos de tipo `TipoBasePila` con los siguientes *métodos*:

   a) Sobrecarga del operador de asignación.

   b) Sobrecargar el operador unario `~` para obtener el elemento de tipo `TipoBasePila` que ocupa la cabecera de la pila. La pila no se modifica.

   c) Sobrecargar el operador combinado `+=` para añadir un dato de tipo `TipoBasePila`.

   d) Sobrecargar el operador unario `--` para eliminar un dato de tipo `TipoBasePila`.

5. Ampliar la clase `Cola` de datos de tipo `TipoBaseCola` con los siguientes *métodos*:

   a) Sobrecarga del operador de asignación.

   b) Sobrecargar el operador unario `~` para obtener el elemento de tipo `TipoBaseCola` que ocupa la primera posición de la cola. La cola no se modifica.

   c) Sobrecargar el operador combinado `+=` para añadir un dato de tipo `TipoBaseCola`.

   d) Sobrecargar el operador unario `--` para eliminar un dato de tipo `TipoBaseCola`.

6. *(Versión adaptada del Examen Ordinario MP 2018/2019. Segunda parte)*

   Continuamos trabajando sobre el ejercicio 7 de la *Relación de Problemas III*. Los métodos (operadores) que se solicitan en este ejercicio sustituyen a métodos escritos en la primera versión de la solución al problema indicado. Descarte los métodos antiguos (o al menos **evite su uso público**).

   a) Sobreescribir el operador de asignación para las clases `RedMetro` y `Linea`. ¿Por qué no es preciso para la clase `InfoParada`?

   Una vez implementado, **descarte** el método `Clona` de las clases `RedMetro` y `Linea` (ejercicio 7.a de la *Relación de Problemas III*).

   b) Implemente el operador `()` para las clases `RedMetro` y `Linea`. En el caso de la clase `RedMetro` permite acceder a un objeto `Linea` y en el caso de la clase `Linea` permite acceder a un objeto `InfoParada`.

   Descarte los métodos de acceso `GetLinea` (clase `RedMetro`) y `GetInfoParada` (clase `Linea`) propuestos en el ejercicio 7.b de la *Relación de Problemas III*).

   c) Implemente el operador `+=` para las clases `RedMetro` y `Linea`. En el caso de la clase `RedMetro` permite añadir un objeto `Linea` y en el caso de la clase `Linea` permite añadir un objeto `InfoParada`.

   Descarte los métodos `AniadeLinea` de la clase `RedMetro` y `AniadeInfoParada` de la clase `Linea` (ejercicio 7.c de la *Relación de Problemas III*).

   d) Implemente los operadores relacionales (todos) para las clases `Linea` y `RedMetro`.

   Se usará un *valor de calidad* asociado a cada clase.

   - Un objeto `Linea` tendrá un valor de calidad que se calcula a partir de: 1) el número de paradas activas (80 %) y 2) el número total de paradas (20 %).
   - Un objeto `RedMetro` tendrá un valor de calidad que se calcula a partir de: 1) el número **total** de paradas activas (70 %) y 2) el número de líneas (30 %).
     Nota: Una parada común a dos líneas contará dos veces, si es común para tres líneas contará tres veces, ... Si una parada no está operativa para una línea, no contará. Lo hará para cualquier otra línea para la que estuviera operativa.

   e) Escriba un programa que lea una red de metro de `cin` (use redirección de entrada), la muestre, calcule su valoración, informe sobre la operatividad de sus lineas y nos diga cual es la parada mejor conectada de la red.

   Escriba una programa que lea una red de metro (recomendamos usar la redirección de entrada) la muestre y calcule su valoración. Para cada línea debe informar acerca de su valoración y operatividad, y finalmente qué línea es la “mejor”.

7. *(Versión adaptada del Examen Ordinario MP Junio 2012. Primera parte)*

   Continuamos trabajando sobre el ejercicio 8 de la *Relación de Problemas III*. Los métodos (operadores) que se solicitan en este ejercicio sustituyen a métodos escritos en la primera versión de la solución al problema indicado. Descarte los métodos antiguos (o al menos **evite su uso público**).

   1. Métodos básicos

   Implemente el operador de asignación para la clase `PoliLinea`. ¿Por qué no es preciso para la clase `Punto2D`?

   Una vez implementado, **descarte** el método `Clona` de la clases `PoliLinea` (ejercicio 8.1 de la *Relación de Problemas III*).

   2. Operadores varios

   a) Implemente el operador de acceso `[]`, que permite acceder (tanto para lectura como para escritura) a un dato de tipo `Punto2D` en una `PoliLinea`. Las posiciones se numeran desde el valor 1.

   Una vez implementado, **descarte** el método `Valor` de la clase `PoliLinea` (ejercicio 8.2 de la *Relación de Problemas III*).

   b) Implemente los operadores lógicos de igualdad `==` y desigualdad `!=` para las dos clases. 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).

   Una vez implementado, **descarte** el método `EsIgualA` de la clase `PoliLinea` (ejercicio 8.2 de la *Relación de Problemas III*).

   c) Sobrecargar el operador binarios `+` de manera que se apliquen: a) a dos `PoliLinea`, b) a una `PoliLinea` y un `Punto2D` y c) a un `Punto2D` y a una `PoliLinea`. En todos los casos se crea una *nueva* `PoliLinea`.

   Los operandos no se modifican.

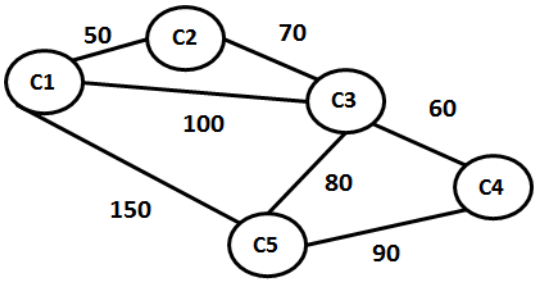
   - El operador `+` cuando se aplica a dos `PoliLinea` se crea una nueva, uniendo el último punto de la primera con el primero de la segunda.
   - Cuando el operador `+` se aplica a una `PoliLinea` y a un `Punto2D` se crea una nueva polilínea, añadiendo el punto al final de la polilínea.
   - Si el operador `+` se aplica a un `Punto2D` y una `PoliLinea` se crea una nueva polilínea, añadiendo el punto al inicio de la polilínea.

   Una vez implementado, **descarte** los métodos `Aniade` y `Concatena` de la clase `PoliLinea` (ejercicio 8.2 de la *Relación de Problemas III*).

   Escribir un programa que lea una serie indeterminada de puntos y cree una dato `PoliLinea` a partir de ellos. Probar los operadores y funciones implementadas.

8. *(Examen de Junio 2017 - parte I)*

   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`:

a) Constructor por defecto y destructor. El constructor por defecto crea una *red vacía* (escriba también el método `EstaVacia` que indique si una red está vacía).

b) Constructor de copia y operador de asignación.

Escriba los siguientes métodos públicos de consulta:

a) `NumCiudades`: devuelve el número de ciudades.

b) `Distancia`: devuelve la distancia entre dos ciudades.

c) `NombreCiudad`: devuelve el nombre de una ciudad (en realidad, la dirección de inicio).

d) `PoblacionCiudad`: devuelve número de habitantes de una ciudad.

2. Métodos de acceso

Usar el operador `()` para acceder a la información de una ciudad de la red. El operador admite un único argumento, n, el número de la ciudad ($n \geq 1$).

Sobrecargar el operador de acceso `()` para que también admita como argumento el *nombre de la ciudad*. Tenga en cuenta que esta operación lleva consigo la necesidad de realizar una *búsqueda*.

3. Métodos de E/S

a) Escriba el método `ToString` para poder *serializar* (creando un `string`) el contenido de una red de ciudades. Deberá contener:

- Un número entero (número de ciudades) y un salto de línea,
- Tantas parejas de líneas de texto como ciudades. En la primera línea de cada pareja aparecerá el nombre de la ciudad y un salto de línea y en la segunda línea su población y un salto de línea.
- Un número indefinido de tripletas en las que los dos primeros valores indican números de ciudades y el tercero, la distancia entre esas dos ciudades.
  Nota: Cada conexión entre ciudades se escribe una sóla vez (no importa cuál es la ciudad de origen o destino).

b) Escriba el método `ToText` para poder insertar en un flujo de salida (`cout`) el contenido de una red de ciudades. Los datos se insertan como se ha detallado en el método `ToString`.

El resultado de `ToText` para la red mostrada en la figura sería:

```
5                    (...sigue)
C1                   C5
50000                250000
C2                   1   2   50
25000                1   3   100
C3                   1   5   150
1500000              2   3   70
C4                   3   4   60
10000                3   5   80
   (sigue...)        4   5   90
```

c) Escriba el método `FromText` para poder leer de un flujo de entrada (`cin`) el contenido de una red. Los datos se proporcionan como se ha detallado en la descripción del método `ToString`.

4. 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 $70 + 80 = 150$.

5. Aplicación

Escriba un programa *completo* que cree una red de ciudades (tomar los datos usando redirección de entrada).

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.

9. Implementa la clase `Conjunto` que permita manipular un conjunto de elementos de tipo `TipoBaseConjunto`. 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 `TipoBaseConjunto`: 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 (TipoBaseConjunto 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 (TipoBaseConjunto 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 `TipoBaseConjunto` 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 `TipoBaseConjunto`, `A+a` será un dato de tipo `Conjunto` y contendrá $A \cup \{a\}$
- Si `A` es un dato de tipo `TipoBaseConjunto` y `a` es un dato de tipo `TipoBaseConjunto`, `a+A` será un dato de tipo `Conjunto` y contendrá $\{a\} \cup A$

m) Sobreescribir el operador binario - para calcular la **diferencia** 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 $A - B$, 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 `TipoBaseConjunto`, `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) Sobreescribir 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 `TipoBaseConjunto`, `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 `TipoBaseConjunto`, `a*A` será un dato de tipo `Conjunto` y contendrá $\{a\} \cap A$

ñ) Añadir un módulo de *interface* entre las clases `Conjunto` y `Secuencia` que ofrezca dos funciones, con los siguientes prototipos:

```
Secuencia ConjuntoToSecuencia (Conjunto & c);
```

Devuelve en un objeto `Secuencia` los datos de un objeto `Conjunto`

```
Conjunto SecuenciaToConjunto (Secuencia & v);
```

Devuelve en un objeto `Conjunto` los datos de un objeto `Secuencia`