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

Metodología de la Programación
Grado en Ingeniería Informática

Prácticas

Curso 2012/2013

Francisco J. Cortijo Bon

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

Grado en Ingeniería Informática. Metodología de la Programación.

Práctica 1. El modelo de compilación

Francisco J. Cortijo Bon

Curso 2012-2013

Objetivos

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

Índice

1. El modelo de compilación en C++

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

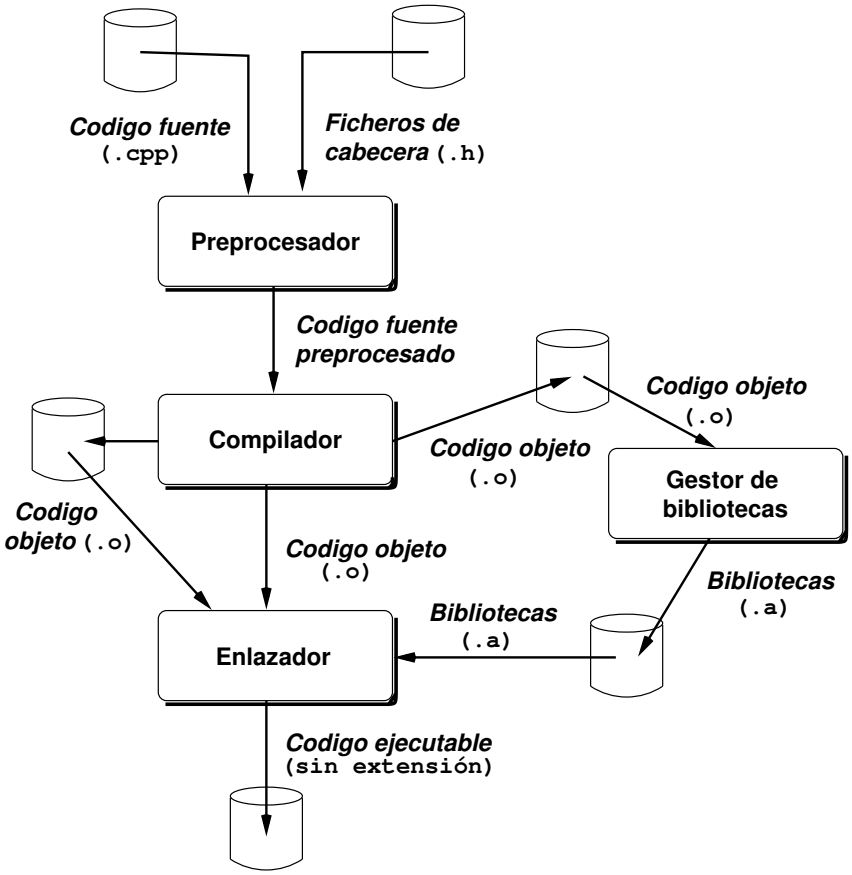


Figura 1: Esquema del proceso de compilación (generación de programas ejecutables) en C++

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

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

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

1. Enlazarlos con otros y generar un ejecutable.

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

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

2. Incorporarlos a una biblioteca.

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

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

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

Ejercicio

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

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

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

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

1.1. El preprocesador

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

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

Dos de las directivas más comúnmente empleadas en C++ son `#include` y `#define`. En la sección B tratamos con más profundidad estas directivas y algunas otras más complejas. En cualquier caso, retomando el esquema mostrado en la figura 1 destacaremos que el preprocesador **no** genera un fichero de salida (en el sentido de que no se guarda el código fuente preprocesado). El código resultante se pasa directamente al compilador. Así, aunque formalmente 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.

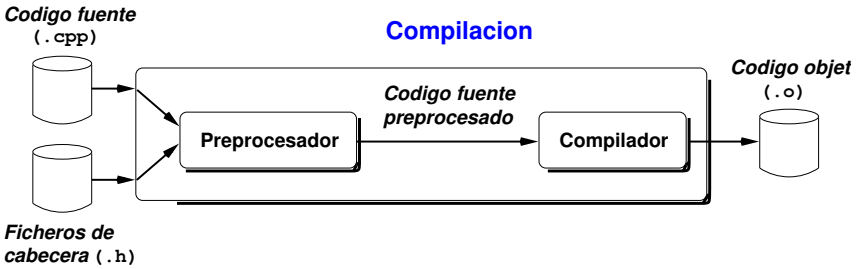


Figura 2: Fase de compilación

1.2. El compilador

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

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

1.3. El enlazador

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

1.4. Bibliotecas. El gestor de bibliotecas

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

No obstante, y afortunadamente, algunas bibliotecas *se enlazan automáticamente* al generar un programa ejecutable, lo que induce al error de pensar que las funciones presentes en esas bibliotecas son propias del lenguaje C++. Otra cuestión es que se ha definido y estandarizado la llamada **biblioteca estándar de C++** (en realidad, bibliotecas) de forma que cualquier compilador que quiera tener el "marchamo" de *compatible con el 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++.

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

Cualquier programador puede desarrollar sus propias bibliotecas de funciones y enriquecer de esta manera el lenguaje de una forma completamente estandarizada. En la figura 1 se muestra el proceso de creación y uso de bibliotecas propias. En esta figura se ilustra que una biblioteca es, en realidad, una "objetoteca", si se nos permite el término. De esta forma nos referimos a una biblioteca como a

una *colección de módulos objeto*. Estos módulos objeto contendrán el código objeto correspondiente a variables, constantes y funciones que pueden usarse por otros módulos si se enlazan de forma adecuada.

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

2.1. Un poco de historia

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

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

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

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

2.2. Sintaxis

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

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

donde:

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

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

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

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

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

2.3. Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

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

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

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

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

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

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

Ejecutar ahora la misma orden pero sin enlazar con la biblioteca matemática (`libm.a`). Observará que funciona correctamente ¿porqué? Ejecute la siguiente orden para interpretarlo:
`g++ -v -o bin/potencias src/potencias.cpp -lm`

`-D` *nombre*[=*cadena*] define una constante simbólica llamada *nombre* con el valor *cadena*. Si no se especifica el valor, *nombre* simplemente queda definida. *cadena* no puede contener blancos ni tabuladores. Equivale a una 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).

A. Introducción al depurador DDD

A.1. Conceptos básicos

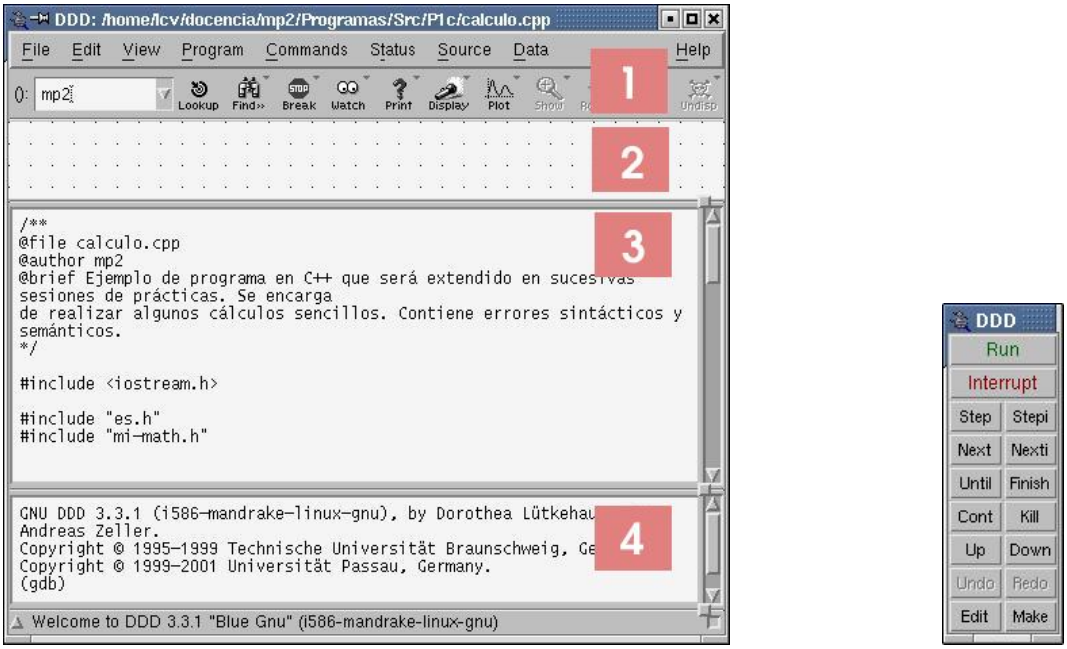
El programa `ddd` es, básicamente, una interfaz (*front-end*) separada que se puede utilizar con un depurador en línea de órdenes. En el caso que concierne a este documento, `ddd` será la interfaz de alto nivel del depurador `gdb`.

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

`ddd` *`fichero-binario`*

A.2. Pantalla principal

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



(a) (b)

Figura 5: Pantalla principal de `ddd`

En ella se pueden apreciar las siguientes partes.

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

Sobre la 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.

A.3. Ejecución de un programa paso a paso

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

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

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

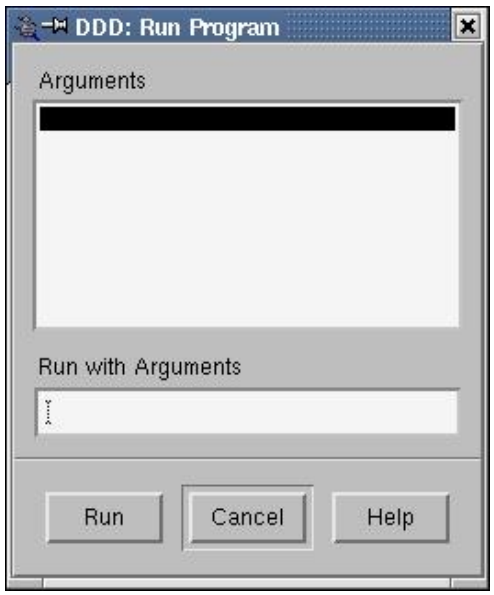


Figura 6: Ventana para pasar argumentos a `main`

Para comenzar a ejecutar un programa bajo control del depurador es conveniente colocar un punto de 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 estuviese visible, entonces puede mostrarse pulsando en menú Program - Run in execution window.

A.4. Inspección y modificación de datos

`ddd`, como cualquier depurador, permite inspeccionar los valores asociados a cualquier variable y modificar sus valores. Se puede visualizar datos temporalmente, de forma que sólo se visualizan sus

[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  .

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.

A.5. Inspección de la pila

Durante el proceso de ejecución de un programa se suceden llamadas a módulos que se van almacenando en la pila. `ddd` ofrece la posibilidad de inspeccionar el estado de esta pila y analizar qué llamadas se están resolviendo en un momendo dado de la ejecución de un programa (ver cuadro 1).

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

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

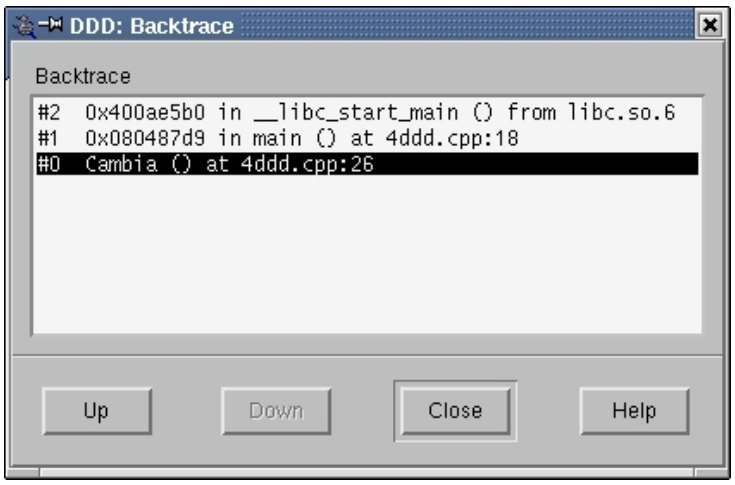


Figura 7: Ventana que muestra el estado de la pilla de llamadas a módulos

A.6. Mantenimiento de sesiones de depuración

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

A.7. Reparación del código

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

B. El preprocesador de C++

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

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

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

B.1. Constantes simbólicas y macros funcionales

B.1.1. Constantes simbólicas

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

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

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

```
#define TAMMAX 256
```

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

```
#define UTIL_VEC
```

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

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

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

B.1.2. Macros funcionales (con argumentos)

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

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

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

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

y tenemos la sentencia

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

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

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

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

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

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

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

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

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

Calcula la diferencia absoluta entre dos operandos.

B.1.3. Eliminación de constantes simbólicas

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

B.2. Inclusión de ficheros

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

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

La única diferencia es que <*fichero*> indica que el fichero se encuentra en alguno de los directorios de ficheros de cabecera del sistema o entre los especificados como directorios de ficheros de cabecera (opción `-I` del compilador: ver sección 2.3), mientras que "*fichero*" indica que se encuentra en el directorio donde se 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.

B.2.1. Inclusión condicional de código

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

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

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

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

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

#include CABECERA
```

De esta forma, estamos seguros de que incluiremos el fichero de cabecera apropiado al sistema en el que estemos compilando. Por supuesto, debemos especificar de alguna forma el valor de la constante `SISTEMA` (por ejemplo, usando macros en la llamada al compilador, como indicamos en la sección 2.3).

Podemos emplear el predicado: `defined(`*identificador*`)` para comprobar la existencia del *identificador* especificado. Éste existirá si previamente se ha utilizado en una macrodefinición (siguiendo a la 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.