Herramientas
MAKE

Que es make?

- Herramienta que permite gestionar grandes programas o grupos de programas
- El tiempo de (re)compilación de grandes programas es considerable
- No es una buena política “recompilar todo” cuando sólo se estuvo trabajando en un número específico de funciones o módulos
- El propósito de **make** es determinar automáticamente cuales piezas de código deben re-compilarse
  make instrumenta los comandos para realizar dicha re-compilación
- Software creado por Richard Stallman y Roland McGrath (1er release 1977)

http://en.wikipedia.org/wiki/Make_(software)

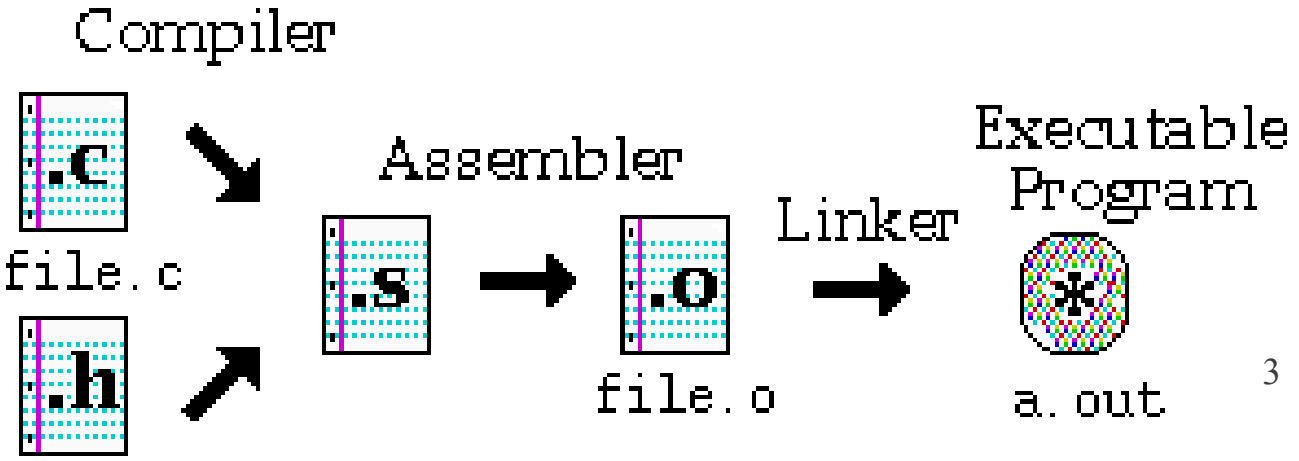
Compilación de un solo programa

- Se requieren tres pasos para obtener el programa ejecutable

- **Etapa de (Pre)compilación**
  Los programas en C se convierten en assembler

- **Etapa de Assembler**
  El codigo assembler de la etapa previa es convertido a código objeto (fragmentos) comprendido directamente por el hardware

- **Etapa de Linkeo**
  Se vincula el código objeto con las librerías que contienen funciones definidas. Aquí se genera el programa ejecutable, llamado a.out por defecto

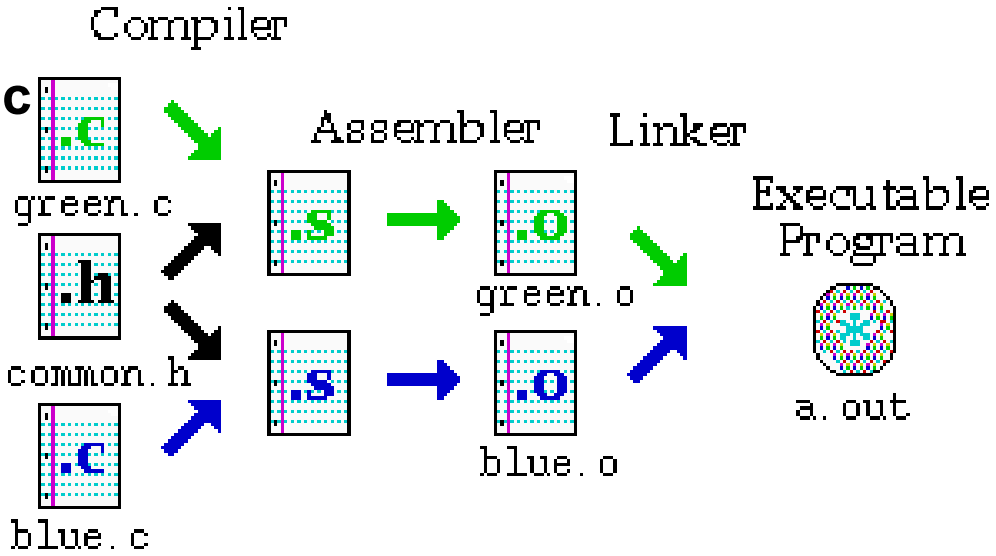


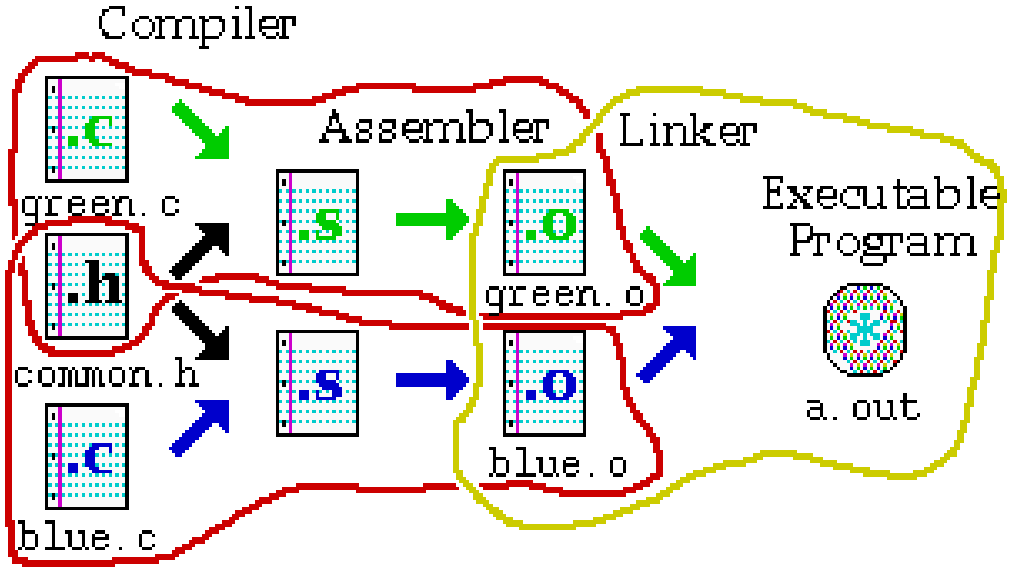
Instituto Politécnico Superior

Compilación de múltiples programas

- Es natural separar la codificación de los sistemas en módulos para simplificar su desarrollo y mantenimiento
- Ejemplo con dos módulos: **green.c** + **blue.c**

La fase de linkeo compone los archivos **green.o** y **blue.o** para generar el programa ejecutable **a.out**





- Dos pasos compilación/assembler
  Un paso de linkeo

```
$ gcc -c green.c
$ gcc -c blue.c
$ gcc green.o blue.o
```

Programación modular

- Al separar el programa C en múltiples archivos se debe considerar:

  - No tener funciones homónimas en diferentes archivos

  - No definir variables globales con igual nombre en diferentes archivos

  - Si se emplean variables globales definirlas en un único archivo y declararla en un archivo .h como

```
extern variable_global;
```

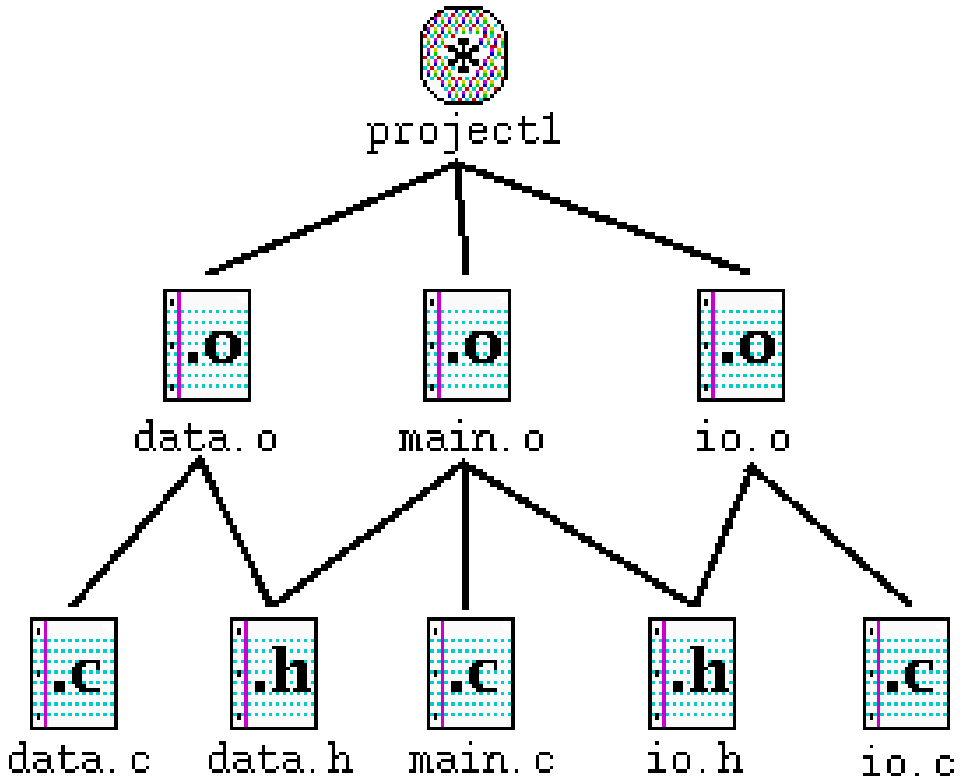
  - Para invocar funciones de otro archivo crear un archivo .h con los prototipos de las funciones y utilizar directivas #include para incluir estos header files en los archivos .c

  - Al menos un archivo debe contener la función main()

Dependencias

- Make crea programas de acuerdo a la dependencias existentes entre los archivos
  . **program.o** depende de **programa.c**
  . **programa.c** pude depender de varios **header files** (.h)

- Grafo de dependencia



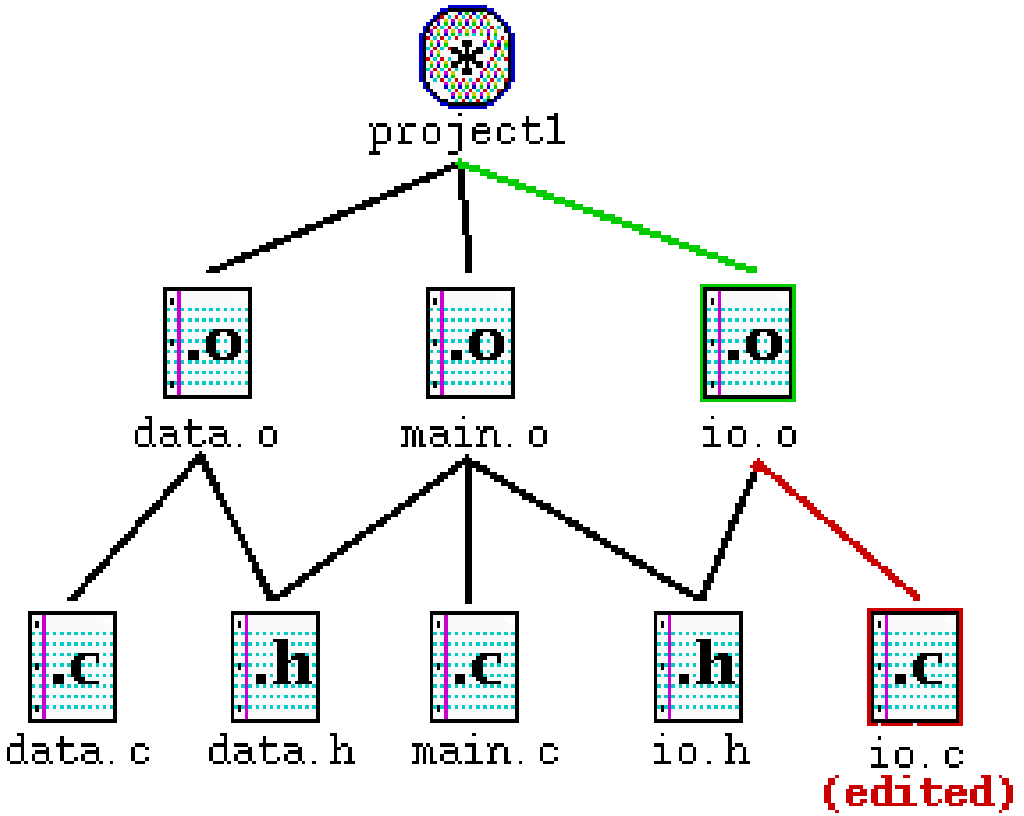
- 5 archivos fuente
  3 archivos .c
  2 archivos .h

Los archivos de un nivel dependen del nivel inferior

main.o depende de:
. data.h
. main.c
. io.h

Dependencias (y compilación)

- De aplicarse un cambio de funcionalidades o debugging el o los cambios impactan en io.o y project1, estos dos archivos deben actualizarse



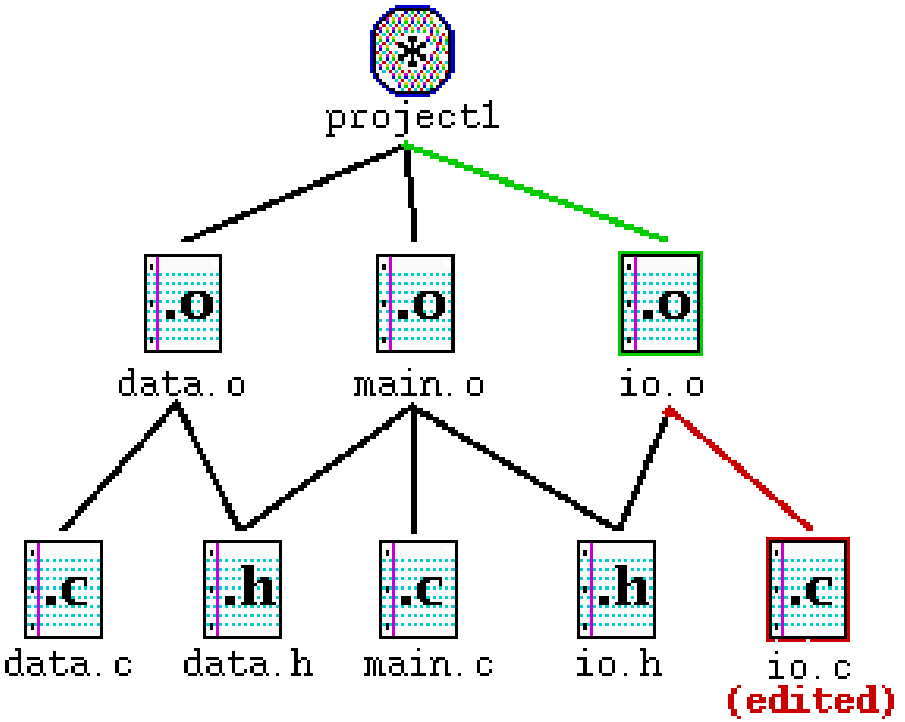
Makefile

- **make** representa a dicho grafo de dependencia mediante un archivo llamado Makefile (makefile o GNUmakefile) ubicado en el mismo directorio que los fuentes

- **make** controla los tiempos de modificación de cada archivo, cuando alguno de Ellos se torna “más actual” que algo que depende de éste se desencadenan las compilaciones respetando las dependencias

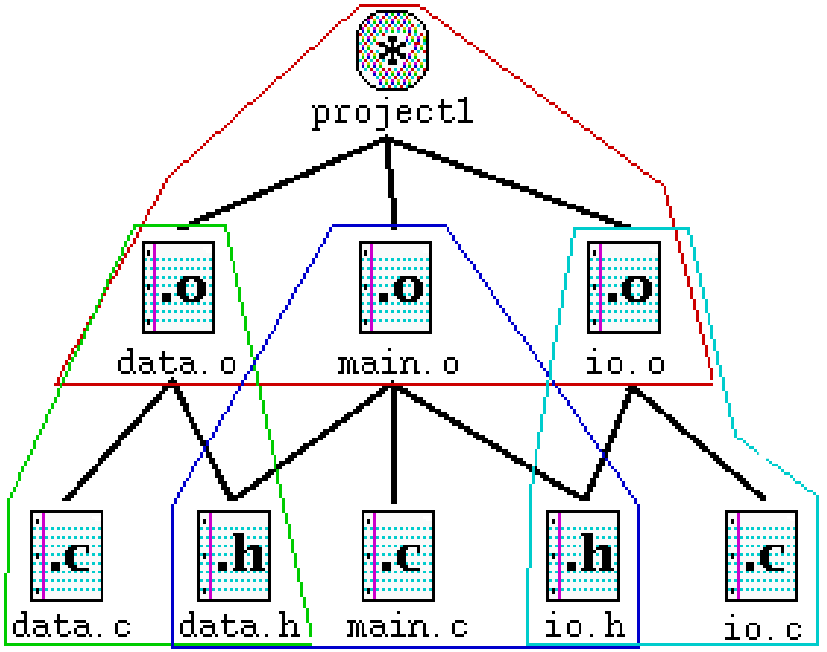
- Ejemplo
  Cambio en `io.c`

```
gcc -c io.c
gcc data.o main.o io.o -o project1
```



Makefile

- Grafo de dependencia y archivo Makefile




```
#Primer ejemplo (comentario)
project1: data.o main.o io.o
    gcc data.o main.o io.o -o project1

data.o: data.c data.h
    gcc -c data.c

main.o: data.h io.h main.c
    gcc -c main.c

io.o: io.h io.c
    gcc -c io.c
```

- Cada dependencia tiene la forma

  `target: source file(s)`

  `(tab)command`

  Cada **`target`** deberá ser creado o modificado en caso de que algunos de los **`source files`** de los que depende cambien, para ello se ejecuta el **`command`**

Make

- Ejemplo de uso

```
$ make
gcc -c data.c
gcc -c main.c
gcc -c io.c
gcc data.o main.o io.o -o project1
```

```
#Primer ejemplo (comentario)
project1: data.o main.o io.o
    gcc data.o main.o io.o -o project1

data.o: data.c data.h
    gcc -c data.c

main.o: data.h io.h main.c
    gcc -c main.c

io.o: io.h io.c
    gcc -c io.c
```

Nota: observar el orden de compilación

```
$ make -f otro_makefile
```
//permite especificar el makefile a utilizar

Make - Macros

- **make** permite el uso de macros para especificar nombres o archivos

```
OBJECTS = data.o io.o main.o
```

Para obtener el valor (expandir la macro), utilizar `$(OBJECTS)`.

- Usos típicos de macros

```
CFLAGS=-c -Wall -O2
```

y luego en los comandos

```
gcc $(CFLAGS) source file(s)
```

```
OBJECTS = data.o main.o io.o

project1: $(OBJECTS)
    gcc $(OBJECTS) -o project1

data.o: data.c data.h
    gcc -c data.c

main.o: data.h io.h main.c
    gcc -c main.c

io.o: io.h io.c
    gcc -c io.c

borrar:
    rm *.o
```

Ejercicio

- Crear un programa que consista en los siguientes archivos
  - **sort.c** // Implementación de los algoritmos de ordenamiento de un array de enteros // enteros: bubble sort, insertion sort, merge sort, quick sort
  - **sort.h** // prototipos de las funciones sort.c
  - **frontend.c** // interfaz de interacción con el usuario para carga e impresión del array // array

  [Para más adelante]
  - **io.c** // funciones para el almacenamiento y captura de valores del array desde archivos // desde archivos
  - **io.h** // prototipos de lsa funciones de io.c

- Crear un archivo **Makefile** de este proyecto el ejecutable debe llamarse “orden”

- **Recursos**:
  . epaperpress.com/sortsearch/downloads/sortsearch.pdf
  . Youtube: Algorithms Lesson 1: Bubblesort
  Algorithms Lesson 2: Insertion Sort
  Algorithms Lesson 3: Merge Sort
  Algorithms Lesson 4: Quick Sort