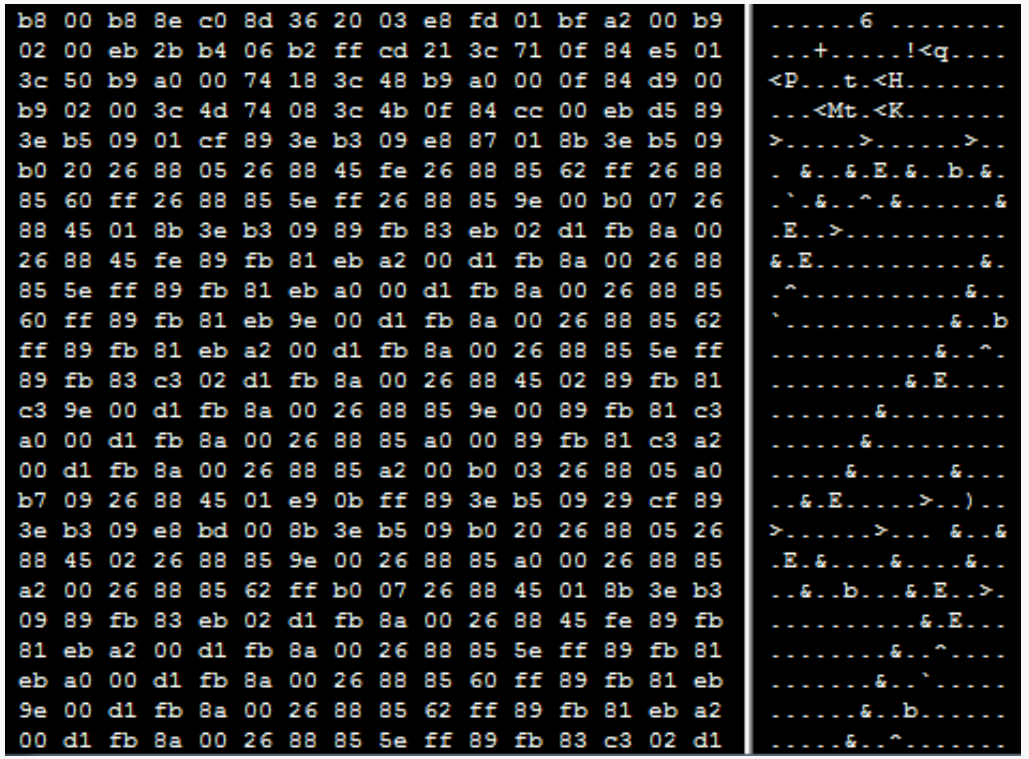
LENGUAJES DE PROGRAMACIÓN

Lenguajes de Programación

Código de Máquina

- Las computadoras ejecutan realmente **código de máquina.**
- Secuencia de bytes que representan instrucciones y e información.



Lenguajes de Programación

- Sin embargo, nosotros programamos en los llamados **lenguajes de alto nivel** (C, Java, Python, etc).
- ¿Cómo pasamos un programa de un lenguaje de alto nivel a código de máquina?

Lenguajes de Programación

Generando código de máquina

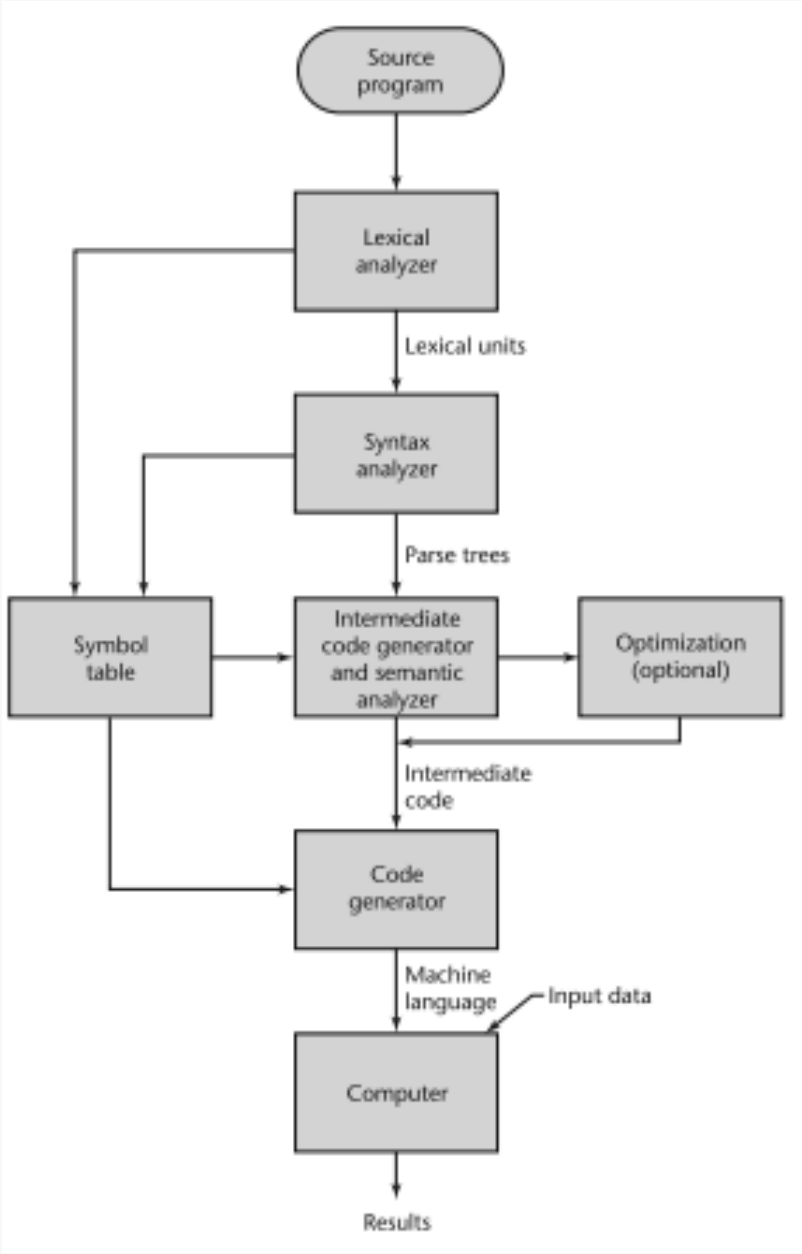
1. Compilación
2. Interpretación pura
3. Sistemas Híbridos

Lenguajes de Programación

Compilación

En los lenguajes compilados,

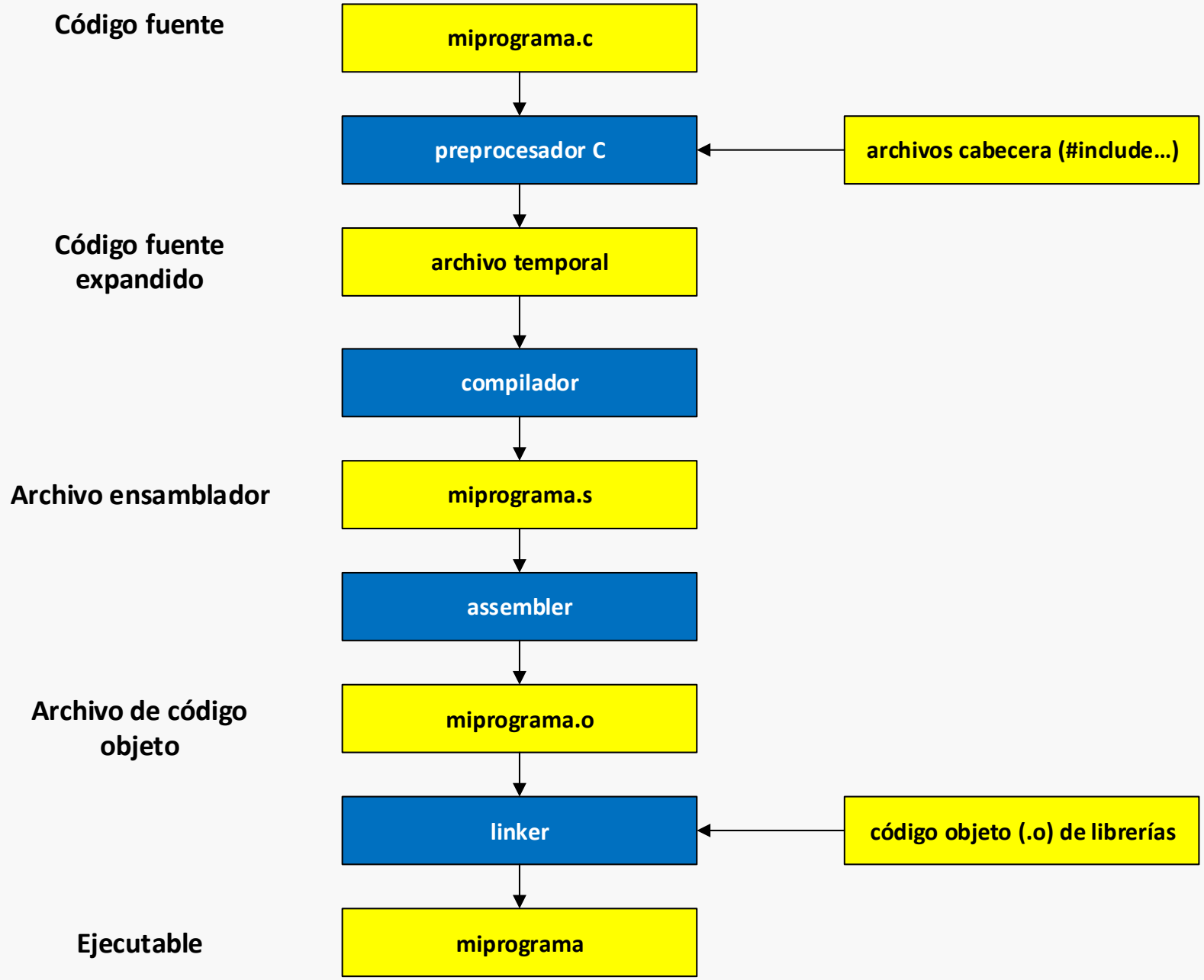
el código fuente es directamente traducido a código de máquina.



Lenguajes de Programación

Generación de código (en gcc):

1. Pre-procesador (salida: código fuente expandido)
2. Compilación (salida: código ensamblador)
3. Assembler (salida: lenguaje de máquina – archivo objeto)
4. Linker (salida: ejecutable)


Código fuente
miprograma.c
preprocesador C
archivos cabecera (#include...)
Código fuente expandido
archivo temporal
compilador
Archivo ensamblador
miprograma.s
assembler
Archivo de código objeto
miprograma.o
linker
código objeto (.o) de librerías
Ejecutable
miprograma

Lenguajes de Programación

Ejemplo:

```
1    int accum = 0;
2
3    int sum(int x, int y)
4    {
5        int t = x + y;
6        accum += t;
7        return t;
8    }
```

Lenguajes de Programación

Linking

Es el proceso de recolectar y combinar varias pedazos de código y unirlos en un solo archivo, que puede ser cargado en memoria y ejectuado.

Lenguajes de Programación

Linking

Puede hacerse durante:

- Compilación
- Carga (en memoria)
- Ejecución.

Util cuando escribimos programas muchos más grandes. Permiten la **compilación por separado.**

Lenguajes de Programación

Linking estático

Toma archivos objetos reubicables, y genera un ejectuable completo.

Dos tareas:

- Resolución de símbolos
- Re-ubicación

Lenguajes de Programación

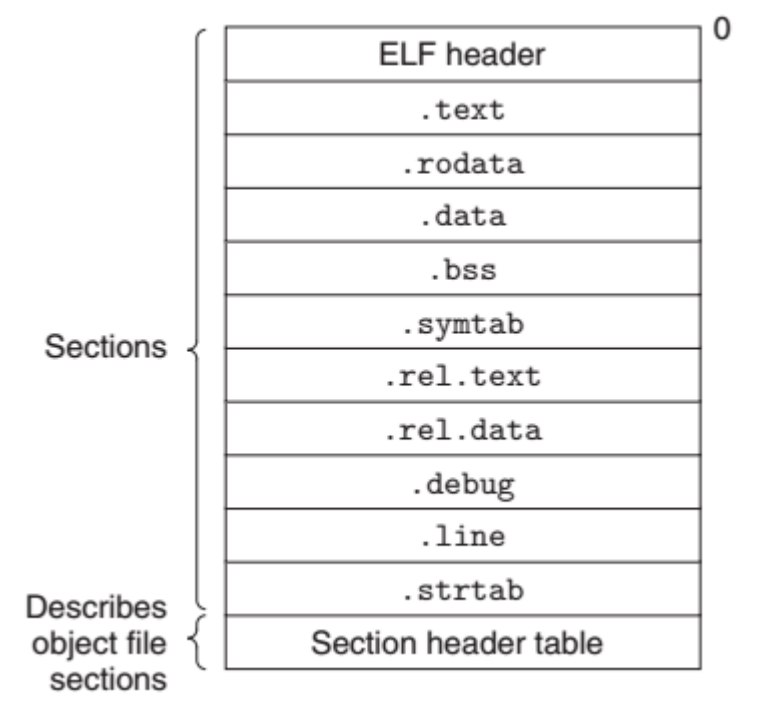
Archivos Objeto

Tres tipos:

1. Re-ubicables (ELF)
2. Ejecutables
3. Compartidos

Lenguajes de Programación

Archivos Objeto Re-ubicables



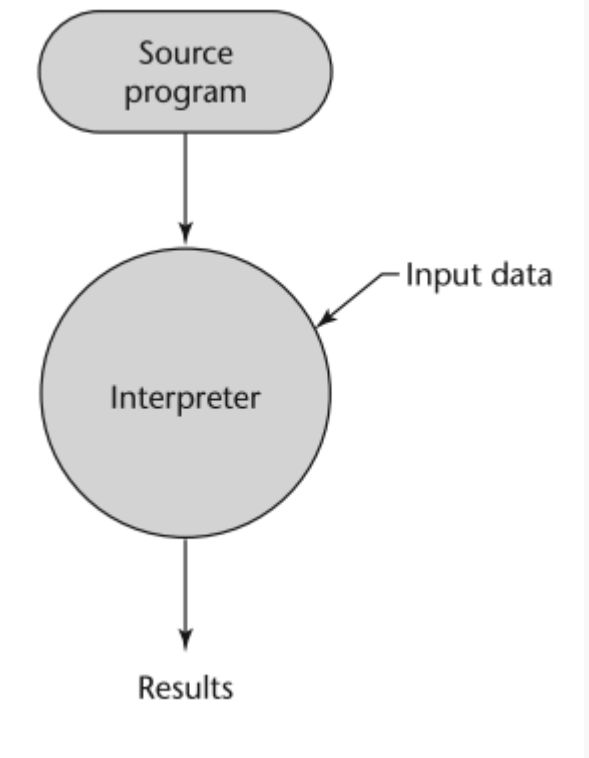
Lenguajes de Programación

Interpretación

Código fuente es traducido por el

Intérprete. El intérprete es una especie

de máquina virtual.



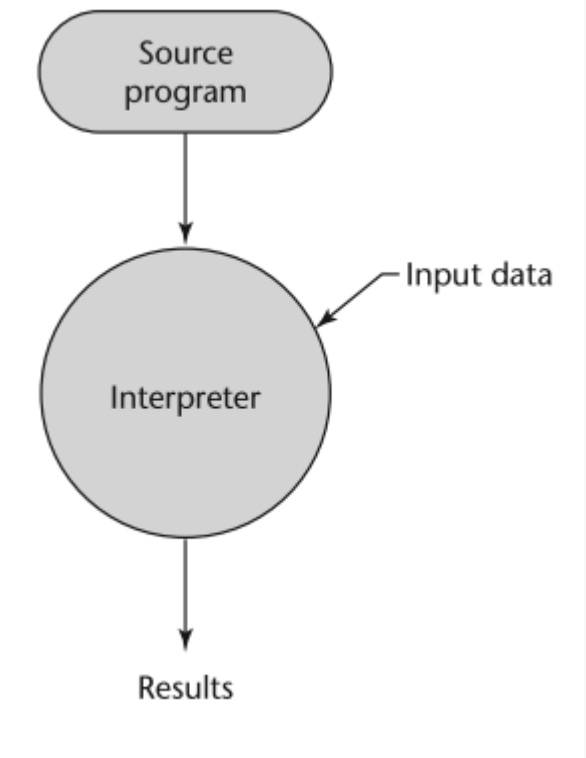
Lenguajes de Programación

Interpretación

Código fuente es traducido por el

Intérprete. El intérprete es una especie

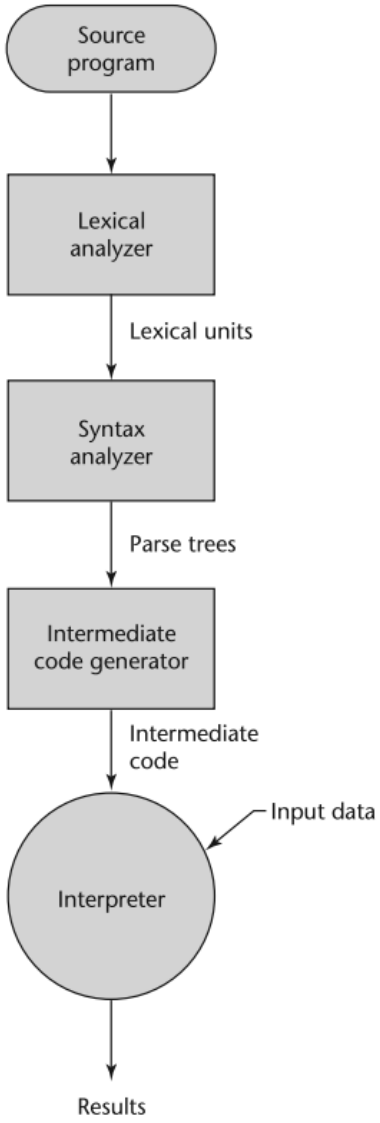
de máquina virtual. Ej Python.



Lenguajes de Programación

Híbridos

Código fuente es traducido a un lenguaje Intermedio, y luego este es ejecutado por el intérprete. Ej Java.



GCC

GCC (GNU C Compiler)

- GCC fue creado como una alternativa gratuita a compiladores creados por Intel, Borland, etc.

- El compilador *de-facto* de toda distribucion Linux. Además tiene la capacidad de llamar a:
  - *El **assembler** (as)*
  - *El **linker** (ld)*
- GCC es un **compilador** y un ***cross-compiler*** *(¿?)*. *GCC* es **modular** (*language front-end)*. Finalmente, GCC es **portable** (¿?). GCC es **gratuito**.

GCC

- Compilando un programa en C. Usemos el clásico programa “Hello World!”

```c
#include <stdio.h>

int main (void) {
        printf ("Hello, world!\n");
        return 0;
}
```

- Para compilarlo, escribimos:

```
$ gcc -Wall hello.c -o hello
```

GCC

- Analicemos el commando:
  - *gcc tiene muchísimas opciones (guíon seguido del comando); en el ejemplo:*
    - **-Wall** → muestra TODAS las advertencias
    - **-o** → Nos permite de definir el nombre del archivo generado.
    - Si omitimos –o , el archivo de salida por defecto es **a.out**
- Para correr el programa:

```
./hello
```

GCC

Errores

```
#include <stdio.h>

int main (void) {
        printf ("Two plus two is %f\n", 4);
        return 0;
}
```

- Si compilamos este programa con el comando de la diapositiva anterior, obtenemos:

```
$ gcc -Wall bad.c -o bad
bad.c: In function ‘main’:
bad.c:6: warning: double format, different type arg (arg 2)
```

GCC

- Si no usabamos –Wall, y ejecutabamos el programa, esta hubiera sido la salida:

```
$ gcc bad.c -o bad
$ ./bad
Two plus two is 2.585495
```

- Esto es un resultado incorrecto. Si este resultado hubiera sido usado para controlar un equipo crítico, el resultado pudiera ser catastrófico o fatal.

GCC

- **Compilación Múltiple Archivos**

```c
#include "hello.h"

int main (void) {
        hello("world");
        return 0;
}
```

main.c

```c
#include <stdio.h>
#include "hello.h"

void hello (const char * name) {
        printf ("Hello, %s!\n", name);
}
```

hello_fn.c

  - *¿Diferencia entre “” y < >?*

GCC

- Para compilar:

```
gcc -Wall main.c hello_fn.c -o newhello
```

- Usar gcc de esta forma llama en un solo comando a:
  - *El compilador*
  - *El assembler*
  - *El linker*

GCC

Compilando archivos de manera independiente

- ¿Por qué?
- Primero los archivos son compilados, luego **enlazados** (*linked)*.
  - *En una **primera etapa**, archivos son compilados y luego assembled (¿?)*
    - Se crean **archivos objetos (.o o .so)**
  - *En una segunda etapa, el **linker** combina los archivos objeto y crea el **ejecutable***

GCC

Creando archivos objeto

- Para `main.c`

```
gcc -Wall -c main.c
```

- Para `hello_fn.c`

```
gcc -Wall -c hello_fn.c
```

- Estos commandos producirán `main.o` y `hello_fn.o`
- La directiva **–c** produce un archivo `.o` con el mismo nombre del archivo `.c`

GCC

Creando el ejecutable

```
gcc main.o hello_fn.o -o hello
```

- Orden de los archivos objeto es importante
  - *Archivo objeto que contiene la definición de una función debe estar **después** del archivo objeto que la usa.*

```
$ gcc hello_fn.o main.o -o hello
main.o: In function ‘main’:
main.o(.text+0xf): undefined reference to ‘hello’
```

GCC

Recompilando y re-enlazando

- Asumamos que hacemos un cambio (en rojo) en el archivo `main.c`

```
#include "hello.h"

int main (void) {
        hello("everyone");
        return 0;
}
```

- Para recompilar **solo** este archivo:

```
gcc -Wall -c main.c
```

- Esto produce un nuevo archivo `main.o`. Para enlazarlo nuevamente:

```
gcc main.o hello_fn.o -o hello
```

GCC

Enlazando con librerías externas

- Estáticas (.a) y dinámicas (.so)
- Usualmente se encuentran en las carpetas `/lib` y `/usr/lib`
- Las definiciones `(.h)` se encuentran en `/include` y `/usr/include`
- Librería estándar de C está en `/usr/lib/libc.a`

```
#include <stdio.h>
#include <math.h>

int main (void) {
        double x = sqrt (2.0);
        printf ("La raiz cuadrada de 2.0 is
%f\n", x);
return 0;
 }
```

GCC

- Al compilar, tenemos un error del linker:

```
$ gcc -Wall calc.c -o calc
/tmp/ccbR6Ojm.o: In function ‘main’:
/tmp/ccbR6Ojm.o(.text+0x19): undefined reference
to ‘sqrt’
```

- Problema: función `sqrt` se encuentra en la libería `libm.a`¿Cómo la incluimos?

```
$ gcc -Wall calc.c -lm -o calc
```

- La directiva `-l` es una atajo para evitar escribir toda la ruta de la librería.

GCC

- La sintáxis de esta directiva es:
  - *Si quiero enlazar con una librería `libNOMBRE.a`, entonces la directiva correspondiente es `-lNOMBRE`*
  - *Por estándar, todas los nombres de librería empiezan con la palabra lib*
- Orden de la liberías: deben aparacer **después de la archivo que las usa**.

```
$ gcc -Wall calc.c -lm -o calc     → correcto

$ gcc -Wall -lm calc.c -o calc     → incorrecto
```

- Si usamos varias librerías, la misma convención se usa. En este caso, `libglpk.a` usa funciones de `libm.a`:

```
$ gcc -Wall data.c -lglpk -lm
```

GCC

Archivos Cabecera (.h)

- ***SIEMPRE*** *incluir el archivo cabecera de definiciones*
- *Si no lo hacemos, funciones pueden pasar argumentos con tipo equivocado, produciendo errores.*

```
#include <stdio.h>

 int main (void) {
        double x = pow (2.0, 3.0);
        printf ("Two cubed is %f\n", x);
        return 0;
}
```

- Si compilamos y corremos, obtenemos un error.

```
$ gcc badpow.c -lm
$ ./a.out
Two cubed is 2.851120
```

GCC

- ¿Qué pasó?
  - *No incluimos* `<math.h>`
  - *Por lo tanto, el prototipo de la función no será visto por el compilador y los argumentos pasaran con el tipo de dato equivocado.*

```
double pow(double x, double y)
```

  - *¡Por eso es importante habilitar* `-Wall`*!*

GCC

Buscando archivos cabecera

- Un error común al compilar es el siguiente:

```
FILE.h: No such file or directory
```

- Sucede por que la librería no se encuentra en ninguno de los directorios estandares que busca `gcc`
- Lo mismo puede pasar con las liberías

```
/usr/bin/ld: cannot find library
```

- Como vimos anteriormente, el compilador buscar en directorios por defecto (`/lib, /usr/lib, /usr/include, /include`)

GCC

Directivas `-I` y `-L`

- *La directiva `-I` nos permite especificar rutas para buscar archivos `.h`*
- *La directiva `-L` nos permite especificar rutas para buscar archivos `.a` y `.so`*

```c
#include <stdio.h>
#include <gdbm.h>

 int main (void) {
            GDBM_FILE dbf;
            datum key = { "testkey", 7 };       /* key, length */

            datum value = { "testvalue", 9 };   /* value, length */

            printf ("Storing key-value pair... ");
            dbf = gdbm_open ("test", 0, GDBM_NEWDB, 0644, 0);

            gdbm_store (dbf, key, value, GDBM_INSERT);
            gdbm_close (dbf);
            printf ("done.\n");
            return 0;
}
```

- El programa hace uso de la librería GDBM (GNU Database Management Library).
  - *Usar la cabecera* `gdbm.h` *y* `libgdbm.a`

- Si la librería **no** se instaló en una ubicación estándar pasará lo siguiente:

```
$ gcc -Wall dbmain.c -lgdbm
dbmain.c:1: gdbm.h: No such file or directory
```

- Asumamos que la librería gdbm `/opt/gdbm-1.8.3/lib/libgdbm.a`, y el archivo cabecera en `/opt/gdbm-1.8.3/include/gdbm.h`

- Compilamos de nuevo:

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include dbmain.c -
lgdbm /usr/bin/ld: cannot find -lgdbm collect2:
ld returned 1 exit status
```

- ¿Qué pasó?
  - *Falta la ubicación de la librería libgdbm.a*
- El comando correcto sería:

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include -L/opt/gdbm-1.8.3/lib dbmain.c
-lgdbm
```

GCC

Librerías compartidas (.so)

- *¿Por qué?*
    1. El **usar librerías estáticas agranda el tamaño del ejecutable** (ya que el código de la librería es anexado al ejectuble).
    2. Si queremos usar una version más actual de la librería debemos **recompilar** el programa (!)
    3. Librerías compartidas tienen la extensión `.so` (shared object)

- GCC preferirá por defecto usar **librerías compartidas**.
  - *Si se especifica* `-lgdbm`, *gcc preferirá* `libgdbm.so` *a* `libgdbm.a`

- Al correr el programa, el linker (*ld*) busca la librería.
- ¿Cómo especificar nuevas rutas de librerias compartidas? → con la variable de entorno `LD_LIBRAR_PATH`

```
$ LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib
$ export LD_LIBRARY_PATH
$ ./a.out Storing key-value pair... done.
```

- Para evitar hacer esto, tambien podemos editar el archivo

/home/<usuario>/.bash_profile

- ¿Y para agregar multiples rutas?
  - *Usamos los dos puntos (:)*

```
$ LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib:/opt/gtk-1.4/lib
$ export LD_LIBRARY_PATH
```

  - *Sintaxis → LD_LIBRARY_PATH=DIR1:DIR2:DIR3*
  - *Para extender los directorios ya existentes:*

```
$ LD_LIBRARY_PATH=/opt/gsl-1.5/lib:$LD_LIBRARY_PATH
```

- Finalmente, si queremos enlazar estaticamente, tenemos dos opciones:
  - *Especificar la librería (.a) directamente:*

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include dbmain.c
/opt/gdbm-1.8.3/lib/libgdbm.a
```

```
$ gcc -Wall -static -I/opt/gdbm-1.8.3/include/
-L/opt/gdbm-1.8.3/lib/ dbmain.c -lgdbm
```

  - *Usar la directiva* `-static`:

GCC

Generando librerías compartidas

Desde un archivo .c:

```
gcc -shared -o libhello_fn.so -fPIC hello_fn.c
```

GCC

Generando liberería estáticas

Desde un archivo .c:

```
gcc -c -o hello_fn.o hello_fn.c

ar rcs libhello_fn.a hello_fn.o
```