Tema 1: Paso de Parámetros, Compilación, Linkado, Proyectos Complejos y Proceso de Depuración

Introducción a la programación II

Marcos Novalbos
Maximiliano Fernández
Alfonso Castro

Índice

Índice

Tema 1 – Apartado 1

1. Paso de Parámetros a través de la línea de comandos: argc y argv

1. Paso de parámetros a través de la línea de comandos

- Algunas veces resulta útil pasar información al programa cuando se ejecuta.
- El método general es pasar información a la función main() mediante el uso de argumentos de línea de comandos
- Un argumento de línea de comandos es la información que sigue al nombre del programa cuando lo ejecutamos.
- El fichero media_alturas.c invoca al main utilizando los siguientes argumentos:

```
int main (int argc, char * argv[ ])
```

- Estos argumentos deben pasarse cuando se ejecuta el fichero media_alturas.exe:

```
$ ./media_alturas.exe 157 180
```

1. Paso de parámetros a través de la línea de comandos

- Hay dos argumentos, **argc y argv**, que se utilizan para recibir los argumentos de línea de órdenes.

- El parámetro **argc**
  - Contiene el número de argumentos de línea de comandos y es un entero.
  - Al menos, o como mínimo, siempre vale 1, ya que el nombre del programa cuenta como primer argumento

- El parámetro **argv**
  - Es un puntero a un array de punteros a caracteres.
  - Cada elemento del array apunta a un argumento de la línea de órdenes

- Todos los argumentos de línea de órdenes son cadenas. Cualquier número tendrá que ser convertido manualmente por el programa al formato deseado

Ejemplo 1: argc y argv

Crear un programa que muestre todos los argumentos introducidos a través de la línea de órdenes. Véase a continuación el resultado de ejecutar el programa

Ejemplo 2: argc y argv

Escribir un programa que calcule la media de una lista de enteros. El programa recibirá el número de elementos de los que consta la lista como argumento desde la línea de comandos, seguido de los números que se quieren calcular.

Usar la función `strtol` que convierte un string en long int

```
long int strtol(const char *str, char **endptr, int base)
```

Ejemplo:

```
long int valor=strtol("5",NULL,10);
```

$./media 4 2 3 5 4

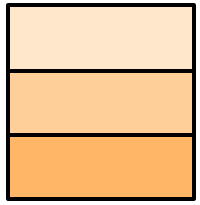
Repaso puntero a puntero

```
#include <stdio.h>
int main()
{
    int i;
    int *ptrToi;            /* Puntero a entero */
    int **ptrToPtrToi;      /* Puntero a puntero a entero */

    ptrToPtrToi = &ptrToi; /* Puntero contiene dirección de puntero */
    ptrToi = &i;           /* Puntero contiene dirección de entero */

    i = 10;                /* Asignación directa */
    *ptrToi = 20;          /* Asignación indirecta */
    **ptrToPtrToi = 30;    /* Asignación con doble indirección */

    return 0;
}
```



Ejemplo 3: argc y argv

Escribir un programa que reciba una matriz 3x3 de números en float a través de la línea de comandos y nos la presente por pantalla

Usar strtof() (string to float)

Ejemplo:

```
float valor=strtof("3.5",NULL);
```

Ejemplo 3: argc y argv

Escribir un programa que reciba una matriz 3x3 de números en float a través de la línea de comandos y nos la presente por pantalla

Ejercicio

- Implementar una programa tipo "calculadora" que reciba por parámetros "argc/argv" los operandos y la operación a realizar. **Está prohibido** usar funciones de lectura de parámetros por STDIN (getchar, scanf, etc...)

- Modo de paso de parámetros:
  - El programa recibirá 3 parámetros especificados de la siguiente manera:

    - Se usarán 3 palabras para identificar los operandos y la operación : OP1, OP2 , OPERACION
    - Estas palabras estarán seguidas (sin espacios) por el separador "=",seguido a su vez por el valor que se quiera dar a esa variable.
      - Ej: OP1=5
    - Los valores válidos para OP1 y OP2 son números enteros
    - Los valores válidos para OPERACION son las palabras "suma", "resta" y "multiplicación", sin comillas

      - Ej: OPERACION=suma

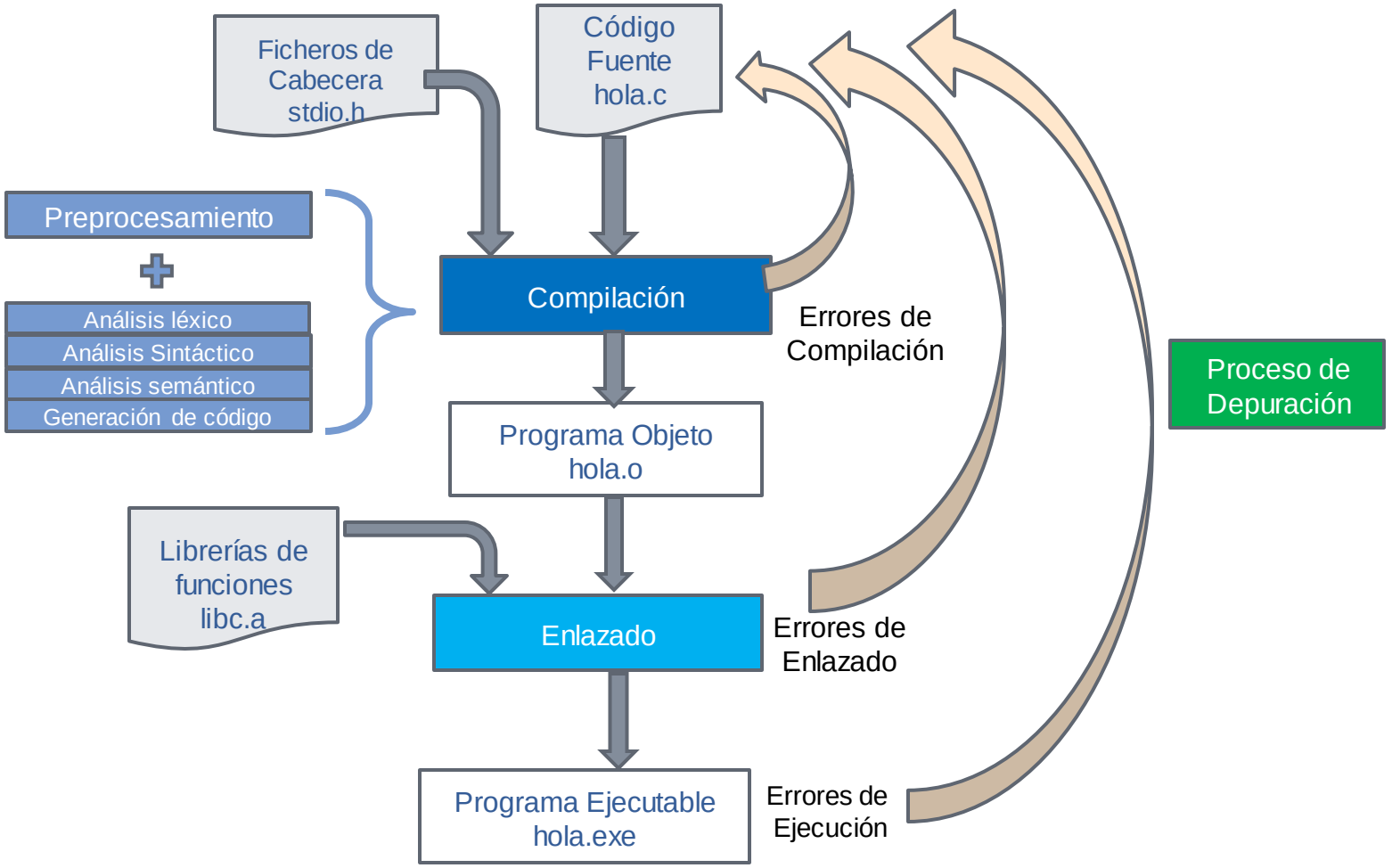
    - Un ejemplo de uso sería el siguiente:
      - ./calculadora OP1=5 OP2=7 OPERACION=suma
      - El resultado obtenido será: 12

Índice

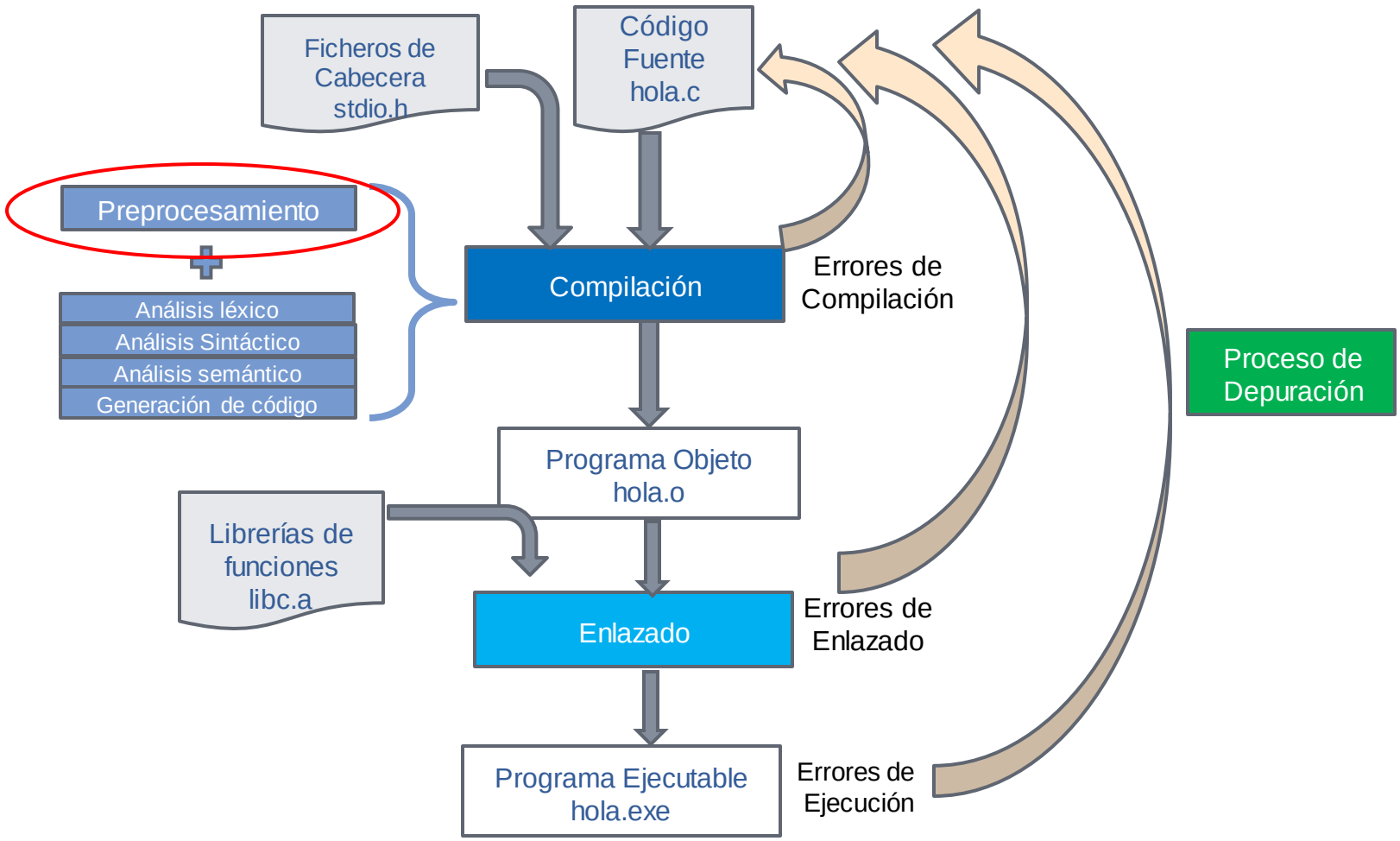
Tema 1 – Apartado 2

2. Proceso de compilación y enlazado

2. Etapas del Proceso de generación de código



2.1 Proceso de Compilación: Preprocesamiento



2.1 Proceso de Compilación: Preprocesamiento

- La primera etapa del proceso de compilación se conoce como preprocesamiento.
- Es una traducción previa y básica que tiene como finalidad “acomodar” el código fuente antes de que éste sea procesado por el compilador en sí.
- El preprocesador modifica el código fuente según las directivas que haya escrito el programador.
- En el caso del lenguaje C, estas directivas se reconocen en el código fuente porque comienzan con el carácter #,
  - #define, #undef, #include, #error, #if, #ifdef, #ifndef, #else, #endef, #pragma.

2.1 Proceso de Compilación: Preprocesamiento

Directiva **#include**

#include <nombre de archivo> #include "nombre de archivo"

- Indica al preprocesador que incluya el archivo de cabecera indicado.
- Los archivos incluidos, a su vez, pueden tener directivas #include.
- Si el nombre del archivo está entre < y > significa que se encuentra en alguno de los directorios que almacenan archivos de cabecera (los estándares del compilador y los especificados con la opción -I cuando se ejecuta la compilación).
- En el otro caso, significa que el archivo que se ha de incluir se encuentra en el directorio actual de trabajo (el directorio a partir del cual se ejecuta el compilador).
- Ejemplo
  - #include <stdio.h>
  - #include "lista.h"

2.1 Proceso Compilación: Preprocesamiento

Directiva #define

#define <nombre macro> <expansión de la macro>

- Crea un identificador con nombre <nombre macro> y una cadena de sustitución de ese identificador.
- Cada vez que el preprocesador encuentre el identificador de la macro, realizará la sustitución de éste por la cadena de sustitución. Esta última termina con un salto de línea.
- Cuando la cadena ocupa más de una línea, se utiliza la barra invertida para indicarle al compilador que omita el salto de línea
- Ejemplo
  - #define MAX_PILA 100
  - #define CUBO (x) ((x)*(x)*(x))

2.1 Proceso de Compilación: Preprocesamiento

Directiva #undef

#undef <nombre identificador>

- Elimina valores definidos por #define.
- Su propósito es asignar los macronombres solo a aquellas secciones de código que los necesiten permitiendo eliminar una definición previa.
- Ejemplo
  - #undef MAX_PILA

2.1 Proceso de Compilación: Preprocesamiento

macros vs. funciones

- Cuando se utiliza una función
  - El código fuente no se encuentra consecutivo
  - Cada vez que se invoca a la función se produce un cambio de contexto y todos los valores de los registros se almacenan en la pila

- Cuando se utiliza un define de una macro dentro de un programa,
  - En tiempo de compilación, el identificador se reemplaza por el código fuente asociado a la macro.
  - No verifica el tipo
  - De esta manera no se rompe la secuencia de ejecución.
  - Se optimiza el tiempo de ejecución

2.1 Proceso de Compilación: Preprocesamiento

Uso de macros como funciones

- Una de las características más potentes de las macros es la de poder acelerar la ejecución de código, ahorrando llamadas a funciones externas:
  - Dos formas de implementar una suma:

```
Int suma(int a,int b)
{
    return a+b;
}
```

```
#define suma(a,b) (a+b)
```

2.1 Proceso de Compilación: Preprocesamiento

macros vs. funciones

- Ventajas:
  - Las llamadas a funciones tipo macro son mucho más rápidas:
    - Código incrustado en la llamada, no se generan instrucciones extra para copia de datos a registros, pila, retorno, etc...
  - Más flexible, es código parseado en tiempo de compilación, los tipos de datos se resuelven en ese momento.

- Inconvenientes:
  - Código menos legible
  - Código más difícil de depurar
  - No son tan potentes como una función
    - No permiten recursividad (no se pueden apilar llamadas a macros por defecto, no existe el concepto de "stack")
    - No existe "return", hay que adaptar el código pasando variables extra para guardar resultados en operaciones complejas.

2.1 Proceso de Compilación: Preprocesamiento

- Ejemplo 1:
  - Funciones y macros con retorno de resultados (return):

```
vector3f suma (vector3f v1,
               vector3f v2)
{
   vector3f result;

   result.x=v1.x +v2.x;
   result.y=v1.y +v2.y;
   result.z=v1.z +v2.z;

   return result;
}
```

```
#define suma(v1, v2, result)\
{\
   result.x=v1.x +v2.x;\
   result.y=v1.y +v2.y;\
   result.z=v1.z +v2.z;\
}
```

2.1 Proceso de Compilación: Preprocesamiento

Directiva #error

#error <mensaje de error>

- Fuerza al compilador a detener la compilación del programa y que el compilador emita el mensaje de error que acompaña a esta directiva
- El mensaje de error no va entre comillas.
- Ejemplo
  - Some floating-point code requires at least 12 digits of resolution to return the correct results. Accordingly, the various variables are defined as type long double. But ISO C only requires that a long double have 10 digits of resolution. Thus on certain machines, a long double may be inadequate to do the job. To protect against this, I would include the following:

```
#include <float.h>
#if (LDBL_DIG < 12)
        #error *** long doubles need 12 digit resolution.
        Do not use this compiler! ***
#endif
```

https://barrgroup.com/embedded-systems/how-to/c-preprocessor-error-directive

2.1 Proceso de Compilación: Preprocesamiento

Directiva #pragma

#pragma <Nombre directiva>

- Un pragma es una directiva que permite proveer información adicional al compilador para que haga algo específico que depende de la arquitectura para la que se compila y sobre la que se ejecutará la aplicación compilada.
- Permite activar funciones extra del compilador, por ejemplo, OpenPM para permitir concurrencia en nuestros programas mediante la creación de threads.
- Se utiliza para hacer algo específico de implementación en C, es decir, "algo pragmático"

https://barrgroup.com/embedded-systems/how-to/c-preprocessor-error-directive

2.1 Proceso de Compilación: Preprocesamiento

Directivas #if #ifdef #ifndef #else #endif

```
#if <expresión constante>
    <secuencia de sentencias>
#endif

#ifdef <identificador>
    <secuencia de sentencias>
#endif

#ifndef <identificador>
    <secuencia de sentencias>
#endif
```

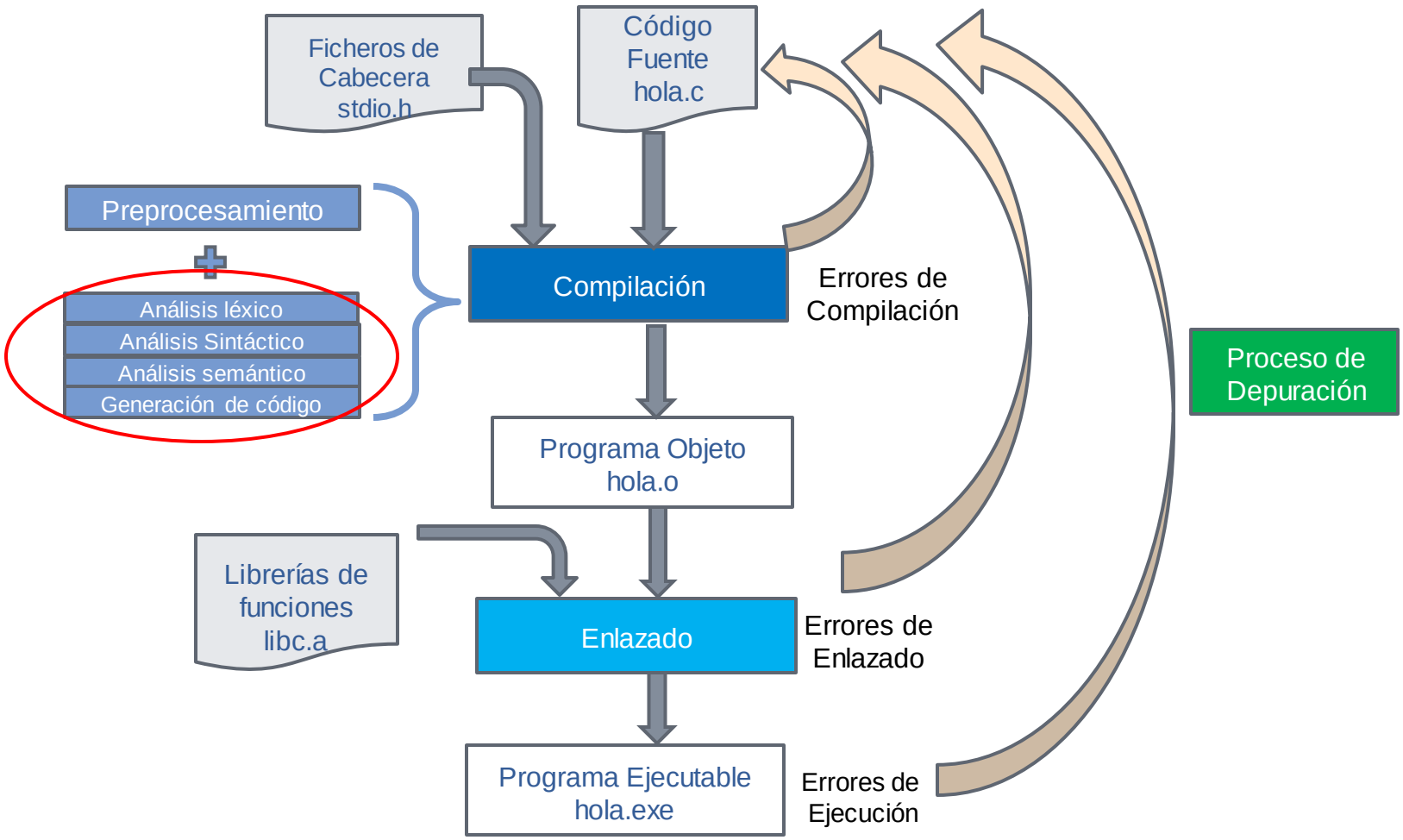
- Estas directivas permiten incluir o excluir condicionalmente partes del código durante la compilación. Si <expresión constante> es verdadera, se compila el código encerrado entre #if y #endif; en caso contrario, se ignora ese código.
- En el segundo caso, si <identificador> está definido (por ejemplo, usando la directiva #define), entonces el código encerrado entre #ifdef y #endif es compilado.
- En el tercer caso, si <identificador>no está definido, entonces el código encerrado entre #ifndef y #endif es compilado.

2.1 Proceso de Compilación: Preprocesamiento

La directiva #else establece una alternativa:

```
#if <expresión constante>
    <secuencia de sentencias 1>
#else
    <secuencia de sentencias 2>
#endif
```

2.2 Proceso de Compilación: fases de análisis y generación de código



2.2 Proceso de Compilación (fases).

- **Análisis Léxico**
  - Extrae del archivo fuente todas las cadenas de caracteres que reconoce como parte del vocabulario y genera un conjunto de tokens como salida. En caso de que parte del archivo de entrada no pueda reconocerse como lenguaje válido, se generarán los mensajes de error correspondiente

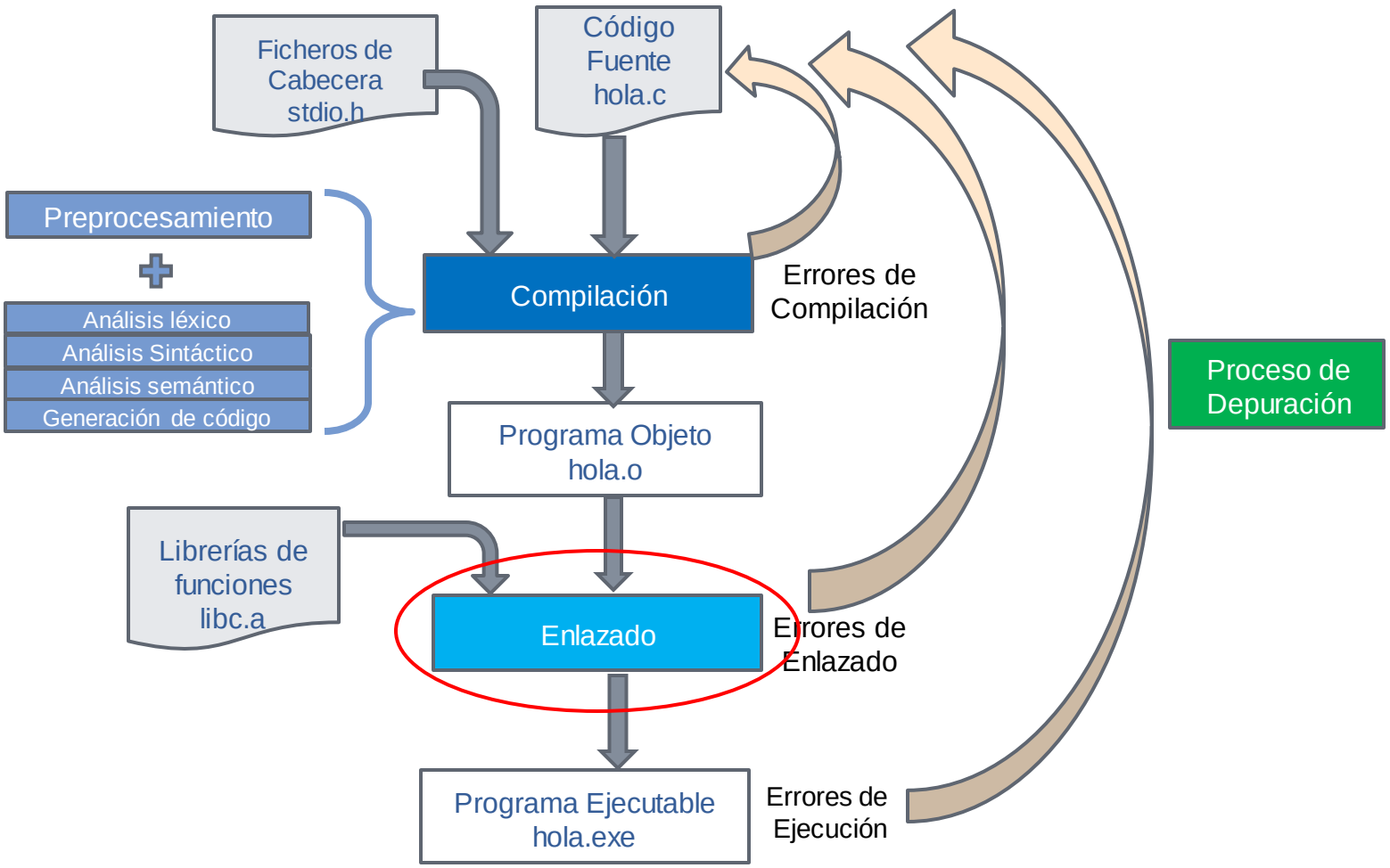
- **Análisis Sintáctico**
  - Se procesa la secuencia de tokens generada con anterioridad, y se construye una representación intermedia, que aún no es lenguaje de máquina, pero que le permitirá al compilador realizar su labor con más facilidad en las fases sucesivas.
  - Esta representación suele ser en forma de árbol

2.2 Proceso de Compilación (fases).

- **Análisis Semántico**
  - Durante el análisis semántico se utiliza el árbol generado en la fase previa para detectar posibles violaciones a la semántica del lenguaje de programación, como podría ser la declaración y el uso consistente de identificadores (por ejemplo, que el tipo de dato en el lado derecho de una asignación sea acorde con el tipo de dato de la variable destino, en el lado izquierdo de la asignación).

- **Generación de código**
  - Se transforma la representación intermedia en lenguaje de máquina (código objeto). En los casos típicos esta fase involucra mucho trabajo relacionado con la optimización del código, antes de generarse el lenguaje de máquina

2.3 Proceso de Enlazado (linkado)



2.3 Proceso de Enlazado

- No siempre las aplicaciones se construyen a partir de un solo archivo fuente.
- En la práctica sólo se escribe una parte, y lo demás se toma de bibliotecas externas que, en la última etapa de la generación de código, se enlazarán unas con otras para generar la aplicación ejecutable final.
- Ésta es, básicamente, la tarea del enlazador: busca en el código objeto las referencias a funciones que usa el programa y las localiza o bien en el propio programa o en las librerías de funciones externas (que tienen extensión .lib o .a).
- Los archivos que se enlazan con nuestros archivos “objeto” para formar el programa se denominan bibliotecas externas que, por su parte, pueden haber sido construidas por nosotros mismos o pueden provenir de terceras partes (por ejemplo, las bibliotecas estándares del compilador). Una biblioteca o librería, en este contexto, es una colección de funciones.
- Este tipo de archivos almacena el nombre de cada función, los códigos objeto de las funciones y la información de reubicación necesaria para el proceso de enlace.
- En el proceso de enlace sólo se añade al código objeto el código de la función a la que se hizo referencia.

Índice

Tema 1 – Apartado 3

3. Proyectos Complejos de más de un fichero

3. Proyectos Complejos de más de un fichero

- En la construcción de programas complejos es muy comun separar el código que se escribe en diferentes archivos y esto tiene muchas ventajas.
- Permite modularizar la funcionalidad y hacer agrupaciones de funciones relacionadas desde un punto de vista lógico.
- Se pueden crear módulos específicos de librerías de funciones que se pueden utilizar en múltiples programas. El uso de los ficheros de cabecera proporcionan la interfaz necesara para poder usar dichas funciones desde cualquier modulo.
- La compilación de módulos se puede hacer por separado lo que facilita la correción de los errores en ficheros de muchas líneas de código.
- En caso tener que modificar un fichero o en caso de encontrar errores de compilación, sólo es necesario volver a compliar dicho fichero con lo que el proceso de compilación es mucho más rápido y eficiente (no necesario compilar el resto de ficheros)

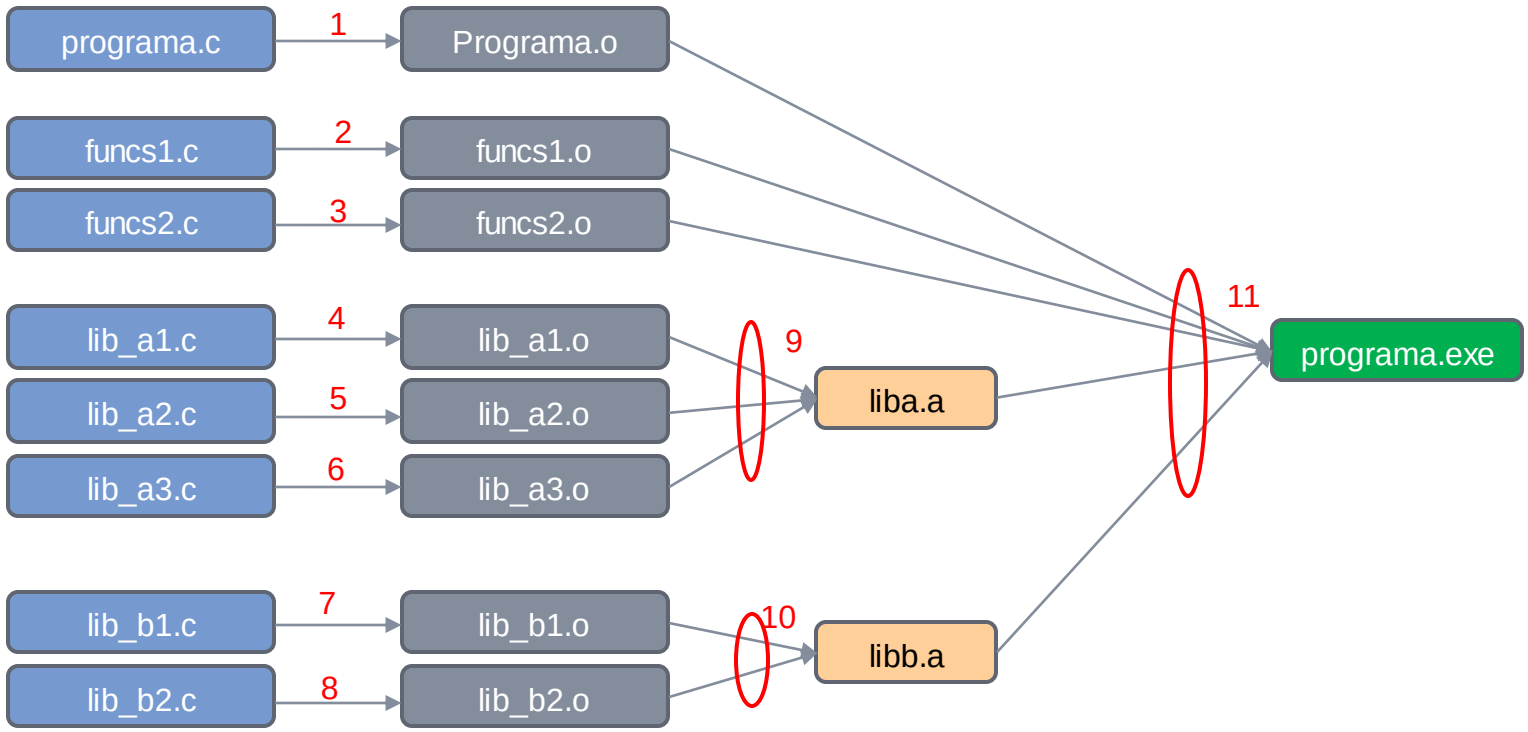
3. Proyectos Complejos de más de un fichero

Mostraremos con un ejemplo cómo construir un programa ejecutable a partir de ficheros objeto y bibliotecas, y cómo construir éstos a partir de una serie de ficheros con código fuente. Nuestro programa se construirá a partir de los siguientes ficheros:

- **programa.c**: Fichero que contiene la función main() y, posiblemente, otras funciones del programa.
- **funcs1.c y funcs2.c**: Ficheros que contienen funciones y/o tipos requeridos en el programa, sin interés en que formen parte de ninguna biblioteca, es decir, son funciones específicas de mi programa y sin posibilidad de uso desde otros programas.
- **lib_a1.c, lib_a2.c, lib_a3.c** y **lib_b1.c y lib_b2.c**: son ficheros que contienen funciones genéricas y de uso generalizado por otros programas. En este ejemplo, las que llevan sufijo “lib_a” formarán una librería denominada liba.a mientras que las que llevan sufijo “lib_b” formarán la librería libb.a

3. Proyectos Complejos de más de un fichero

Esquema completo del proceso de generación del ejecutable de un proyecto complejo



3. Proyectos Complejos de más de un fichero

Generaremos los módulos objeto correspondientes a cada uno de los módulos fuente. El orden de estas ocho acciones no es importante:

1. $gcc -c programa.c -o programa.o

2. $gcc -c funcs1.c -o funcs1.o
3. $gcc -c funcs2.c -o funcs2.o

4. $gcc -c lib_a1.c -o lib_a1.o
5. $gcc -c lib_a2.c - o lib_a2.o
6. $gcc -c lib_a3.c -o lib_a3.o

7. $gcc -c lib_b1.c -o lib_b1.o
8. $gcc -c lib_b2.c -o lib_b2.o

3. Proyectos Complejos de más de un fichero

Antes de generar el ejecutable deben estar creadas las dos bibliotecas, por lo que debemos crearlas ahora que los módulos objeto que las componen ya se han creado. :

```
9. ar -rvs liba.a lib_a1.o lib_a2.o lib_a3.o
10. ar -rvs libb.a lib_b1.o lib_b2.oc
```

Las bibliotecas se crean con el programa **ar**, el gestor de bibliotecas de GNU.

3. Proyectos Complejos de más de un fichero

Una vez creadas las bibliotecas ya puede crearse el ejecutable, enlazando el archivo “programa.o” (que contiene la función main()) con los módulos objeto funcs1.o y funcs2.o y con las bibliotecas liba.a y libb.a.

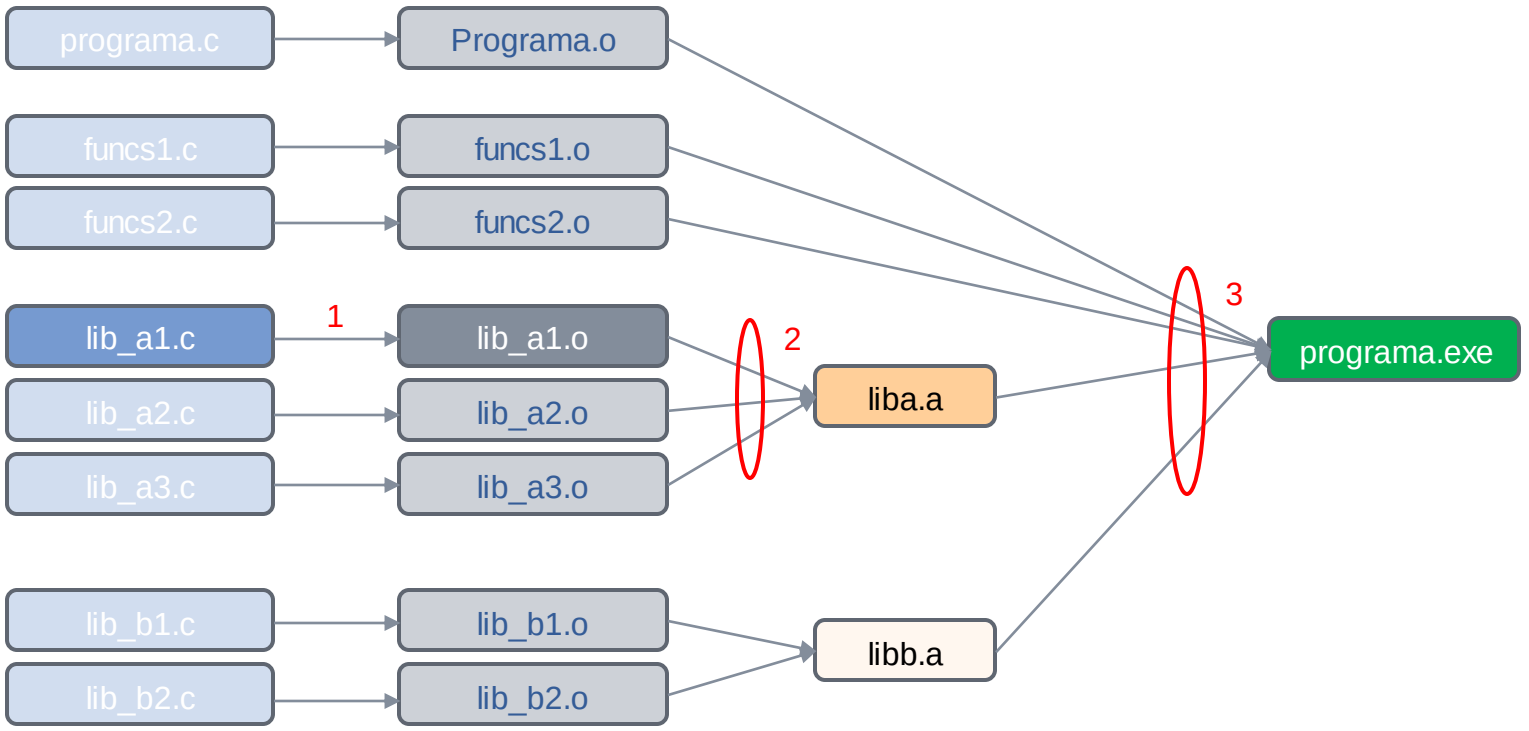
```
11. gcc -o programa programa.o funcs1.o funcs2.o liba.a libb.a
```

Equivalente a:

```
11. gcc  programa.o funcs1.o funcs2.o liba.a libb.a -o programa
```

3. Proyectos Complejos de más de un fichero

Acciones a realizar si se modifica *lib_a1.c*



3. Proyectos Complejos de más de un fichero

Las acciones a realizar en este caso serían:

1. gcc -c lib_a1.c -o lib_a1.o
2. ar -rvs liba.a lib_a1.o lib_a2.o lib_a3.o
3. gcc programa.o funcs1.o funcs2.o liba.a libb.a -o programa.exe

3. Proyectos Complejos de más de un fichero

Utilidad "make"

- En un caso real es bastante complicado gestionar adecuadamente las acciones a realizar y el orden en que se deben acometer. Para resolver estas dificultades disponemos de la utilidad (make).

- **make** se encarga de comprobar qué ficheros se han modificado y de realizar las acciones oportunas para que todo lo que depende de estos ficheros se reconstruya ordenadamente de forma que se incorporen las modificaciones introducidas.

- El programador tan sólo debe especificar, en un formato adecuado, las dependencias entre ficheros y las acciones que deben ejecutarse si se realiza alguna modificación. Esta especificación se realiza mediante un fichero especial denominado genéricamente makefile.

3. Proyectos complejos de más de un fichero

- Archivos Makefile
  - El comando "make" buscará un archivo de compilación llamado "Makefile", que contiene las secuencias de comandos para poder compilar el proyecto.
    - Contendrá una lista de archivos, librerías y compiladores (o los comandos necesarios para buscarlos), que se usarán
    - Por cada archivo "objeto" que se generará, habrá un comando para indicar cómo compilarlo
    - Ej básico:

```
all:
    gcc archivo.c -o archivo.exe
```

3. Proyectos complejos de más de un fichero

- Archivos Makefile
  - Reglas similares a los scripts de linux:
    - Importante la indentación y separación con espacios
    - Declaración de variables:
    - Acceso a contenido de las variables usando "$(variable)"
    - Declaración de "secciones/funciones"

```
NombreVariable=lista valores
```

```
NombreFuncion: "precondiciones"
    Comando a ejecutar
```

    - Norma general:
      - Empieza a ejecutar por defecto en la sección "all"
      - Antes de invocar una "función", deben existir los archivos de las "precondiciones"
      - Por cada archivo de la precondición que no exista, debe existir una "función" que sirva para crearlo

```
COMPILER= gcc
OBJFILE= archivo.o
EXENAME= archivo.exe

all: $(OBJFILE)
    $(COMPILER) $(OBJFILE) -o $(EXECNAME)

archivo.o: archivo.c
    $(COMPILER) -c archivo.c -o archivo.o
```

3. Proyectos complejos de más de un fichero

- Makefile genérico con comodines:

```
SRC= $(wildcard *.c)
OBJFILES= $(SRC:.c=.o)
COMPILER= gcc
PROGRAMNAME= prog.exe

all:  $(OBJFILES)
    $(COMPILER) -g $(OBJFILES) -o $(PROGRAMNAME)

%.o: %.c
    $(COMPILER) -g -c $< -o $@

clean:
    rm $(OBJFILES)
```

3. Proceso de compilación y enlazado (gcc)

Ejemplo 1 :

- Requisistos:
  - Desarrollar una librería que realice el producto escalar y producto vectorial de dos vectores de 3 elementos.
  - Desarrollar un programa que calcule el producto escalar y el producto vectorial de los vectores "<1,2,3>" y "<3,2,1>“
- Pasos a seguir:
  - Identificar los tipos de datos
  - Crear estructuras que puedan albergarlos
  - Crear implementaciones de las operaciones pedidas
  - Crear el programa que use las operaciones con datos dados

3. Proceso de compilación y enlazado (gcc)

Ejemplo 1:

Librería "multVectors":

MultVectors.h

```
#ifndef _MULTVECTOR_H
#define _MULTVECTOR_H

typedef struct vector3f{
   float x,y,z;
}vector3f;

float dotProdV(vector3f v1,vector3f
v2);
vector3f crossProdV(vector3f
v1,vector3f v2);

#endif
```

multVectors.c

```
#include "multVectors.h"

float dotProdV(vector3f v1,vector3f v2){
   return v1.x*v2.x + v1.y*v2.y +
   v1.z*v2.z;
}
vector3f crossProdV(vector3f v1,vector3f
v2)
{
   vector3f result;

   result.x=v1.y*v2.z- v1.z*v2.y;
   result.y=v1.z*v2.x-v1.x*v2.z;
   result.z=v1.x*v2.z-v1.z*v2.x;

   return result;
}
```

3. Proceso de compilación y enlazado (gcc)

Ejemplo 1:

Programa principal:

```
#include <stdio.h>
#include <stdlib.h>
#include "multVectors.h"

int main(int argc,char** argv)
{
   vector3f v1={.x=1,.y=2,.z=3};
   vector3f v2={.x=3,.y=2,.z=1};
   float dot=0;
   vector3f cross={.x=0,.y=0,.z=0};
   dot=dotProdV(v1,v2);
   cross=crossProdV(v1,v2);
   return 0;
}
```

3. Proceso de compilación y enlazado (gcc)

Ejemplo1:
una vez construida la librería tendríamos que:

- Compilar en varios pasos con gcc:
  - Por cada archivo ".c" de código fuente, generar un archivo de tipo "objeto" (archivos "*.o")
    - De esta manera, se tienen los archivos compilados en varias partes
    - Más eficiente si hay archivos que con alta probabilidad de cambios/modificaciones
    - Menor tiempo en compilación, sólo se recompilan los archivos modificados
  - **Con gcc: flag "-c"**
    - Este flag realizará las operaciones de:
      - Lectura de un archivo de código
      - Parseo de directivas de precompilado
      - Verificación de código
      - **Generación de archivo objeto ".o"**

3. Proceso de compilación y enlazado (gcc)

Ejemplo 1:

En nuestro ejemplo del producto de vectores:

- Generación del código preprocesado:
  $gcc -E multVectors.c > multVectors.pp

- Por cada archivo ".c" del proyecto, generar un archivo de tipo objeto precompilado
  $gcc –c multVectors.pp -o multVectors.o
  $gcc –c main.pp -o main.o

- Tenemos dos objetos por separado. Si sólo se modifica el archivo "main.c", sólo se vuelve a compilar ese archivo.
  $gcc –c main.c -o main.o

3. Proceso de compilación y enlazado (gcc)

Ejemplo 1:

Proceso de "enlazado" (linker):

- Por cada archivo de objeto generado anteriormente, unirlos en un único ejecutable:

```
$gcc main.o multVectors.o -o mulVectors.exe
```

- El comando anterior añade cabeceras de inicialización del ejecutable, dependientes del sistema operativo:
  - LLamadas a inicialización de "stack" (memoria de pila, variables globales/locales)
  - Definiciones del punto de entrada al programa "main"
  - Definición de DLLs que usará el ejecutable
  - En linux, el formato de ejecutable usado es "ELF" ("PE" en windows)

3. Proceso de compilación y enlazado (gcc)

Ejemplo 1:

- En caso de querer usar librerías que existan en nuestro sistema, hemos de especificarlo en el momento de enlazado, con el flag "-l <nombre de librería>"

- Ejemplo, veamos el enlazado con la librería matemática:
  - Si hemos usado en nuestro programa la función de tipo "raiz cuadrada" (sqrt), cuya implementación está en la librería matemática, entonces en gcc temenos que enlazar con el archivo "libm.a".
  - El nombre de la librería en este caso se puede abreviar como "m" (se quita el prefijo "lib" y la extensión)

```
$gcc  main.o  multVectors.o  -lm  -o multVectors.exe
```

Índice

Tema 1 – Apartado 4

4 Linkado de librerías

4. Linkado de librerías y linkado dinámico

- Ya hemos visto en el capítulo anterior como crear un proyecto complejo y utilizar en el proceso de linkado algunas librerías.
- Las librerías ofrecen una capa de modularidad al proyecto:
  - Secciones de código reutilizable entre proyectos
  - Compilación del proyecto por partes
  - Uso de código externo al proyecto

- Existen dos tipos de linkado de código con librerías:
  - Estático: El usado hasta ahora, todo el código generado se mantiene en el mismo ejecutable
  - Dinámico: Uso de archivos DLL(Windows, Dinamic Library Link) ó SO (Unix/Linux: Shared Object/Library)

4. Linkado de librerías y linkado dinámico

- Diferencias entre linkado estático y dinámico:
  - Estático:
    - Genera ejecutables más grandes
    - Todo el código se encuentra en un único archivo
    - Mover/distribuir el ejecutable es más fácil
    - Modificar el ejecutable una vez compilado es dificil:
      - Hay que recompilar todo y volver a distribuirlo
  - Dinámico:
    - Ejecutables más ligeros
    - El código generado se mantiene en varios archivos
      - Más fácil de modificar el código y generar parches
      - Opciones a crear "plugins"
    - Más difícil de distribuir el ejecutable
      - Se necesitan "instaladores", todas las librerías usadas deben estar disponibles en el sistema

4. Linkado de librerías y linkado dinámico

- Ejemplo de creación de librerías estáticas con gcc:
  - Comando "ar" (archiver):

```
gcc –c multVectors.c -o multVectors.o
ar  rcs libMultVectors.a multVectors.o
```

- Ejemplo de creación de librerías dinámicas con gcc:
  - `gcc –c –fPIC multVectors.c -o multVectors.o`
  - `gcc -shared -o libMultVectors.dll  multVectors.o`

- Linkando con librerías: opción "-l<nombre librería>"
  - `gcc main.c -lMultVectors –o main.exe`
  -

Resumen de uso de gcc

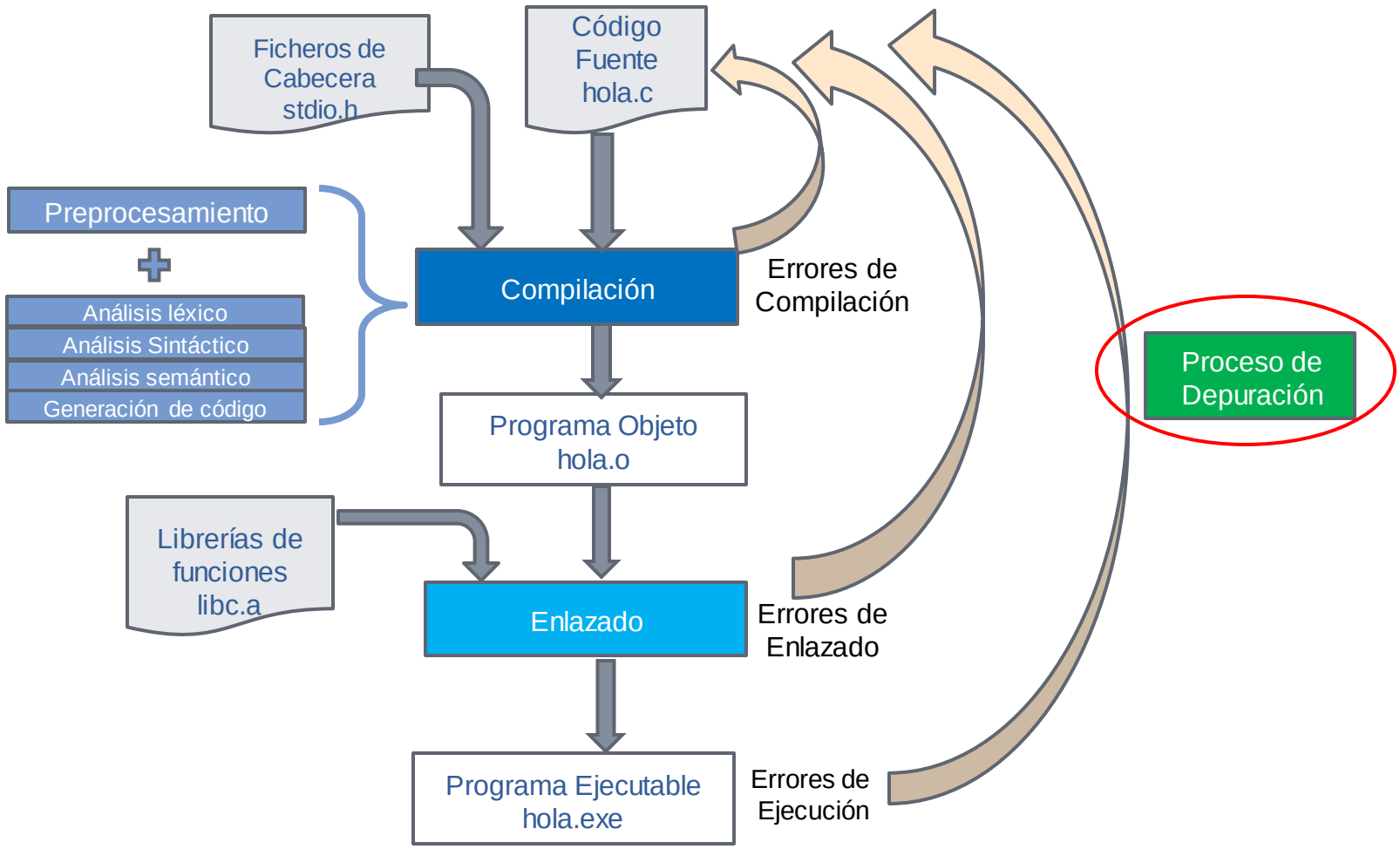
- Para generar un ejecutable con fuente de un solo archivo:
  $ gcc circulo.c -o circulo
- Para crear el fichero preprocesado
  $ gcc –E circulo.c > circulo.PP
- Para crear un módulo objeto con el mismo nombre del fuente y extensión .o:
  $ gcc -c circulo.c
- Para enlazar un módulo objeto en el ejecutable “circulo”:
  $ gcc circulo.o -o circulo
- Para enlazar varios módulos objeto: verde.o, azul.o, rojo.o, ya compilados separadamente, en el archivo ejecutable “colores”:
  $ gcc verde.o azul.o rojo.o -o colores

Índice

Tema 1 – Apartado 5

5. Depuración: GDB

5. Proceso de Depuración y verificación



5. Proceso de Depuración y verificación

- La depuración de un programa es el proceso de encontrar los errores de ejecución de un programa y corregir o eliminar dichos errores.

- Depuración manual:
  - Se proporciona al programa entradas válidas que conducen a una solución conocida.
  - También deben incluirse datos válidos para comprobar la capacidad de detección y generación de errores del programa
  - Se incluyen “trazas” en el programa para ir comprobando los valores intermedios que se van obteniendo son los esperados.

5. Proceso de Depuración y verificación

- Depuración a partir de herramientas:
  - Dada la complejidad y el tamaño de los programas y, para ahorrar tiempo y recursos, existen lo que conocemos como herramientas de depuración (debugger).
  - Todos los IDE tienen asociados su herramienta de depuración.
  - Las acciones más habituales que realizan estas herramientas son:
    - Ejecutar paso a paso un programa (stepping).
    - Marcar puntos de parada (breakpoints).
    - Examinar el contenido de las variables y objetos.
    - Conocer el encadenamiento de llamadas de procedimientos.
    - Retomar la ejecución hasta un nuevo punto de detención.

5. Proceso de Depuración y verificación

- La verificación es la acción de comprobar que el programa está de acuerdo con su especificación o definición de requisitos.
- Es decir, se comprueba que el sistema cumple con los requerimientos funcionales y no funcionales que se han especificado.
- Verificación Software: Asignatura Optativa de INSO4

"La mejor herramienta de depuración es evitar los errores desde el principio"

5. Proceso de Depuración y verificación

- Herramienta de depuración gdb
  - Compilar con opciones de depuración:
    - **gcc –g archivo.c -o archivo.exe**
  - Cargar el ejecutable en el depurador:
    - **gdb archivo.exe**
  - Poner breakpoints en algún punto del programa:
    - Opción 1:
      - **break num_linea**
      - **break fichero:num_linea**
    - Opción 2:
      - **break nombreFunción**
    - Borrar un breakpoint:
      - **delete n (borra el breakpoint n)**
      - **delete (borra todos los breakpoints)**
    - Informacion sobre breakpoint
      - **info break**

5. Proceso de Depuración y verificación

- Ejecutar el programa:
  - **run**
  - **run param1 param2... ParamN**
- Consultar los argumentos del programa:
  - **show args**
- Ejecutar una instrucción (paso a paso):
  - **step**
- Continuar con la ejecución:
  - **continue**
- Inspeccionar variables:
  - **print nombreVariable**
  - **print *nombreVariable**
  - **print *nombreArray@tamanio**
- Salir de la depuración:
  - **quit**

5. Proceso de Depuración y Verificación

- Mostrar el estado de la pila de llamadas:
  - Primero parar el programa: Ctrl+c
  - Mostrar el estado del programa:
    - **backtrace**
  - Moverse a una sección de la pila de llamadas
    - **frame "número"**

Asignatura/Tema