Algoritmos y Estructuras de Datos
Ingeniería en Informática, Curso 2º

SEMINARIO DE C
Sesión 4

Contenidos:

1. El preprocesador de C
2. Programación modular
3. El programa make

Ejercicios

1. El preprocesador de C

- El **preprocesador** es una parte del compilador que procesa los ficheros **.c** antes que el compilador propiamente dicho. Admite varios comandos, todos los cuales empiezan por **#**.

- **Comando: `#define`**. Permite definir una constante, con o sin parámetros. Su valor será sustituido textualmente en los sitios donde se encuentre en el código. Estas constantes se llaman también **macros**.

    - **Macro sin parámetros**. Normalmente se ponen en mayúsculas, para diferenciarlas de las variables.

      `#define NOMBRE  VALOR`

      ```
      #define PI 3.1415926
      #define MAX_INT  ((unsigned)(~0)>>1)
      #define MENSAJE "Mensaje predefinido"
      ```

    - **Macro con parámetros**. Es parecido a una función, pero sigue siendo una constante:
        - Los parámetros no tienen tipo.
        - La expresión se sustituye textualmente en tiempo de compilación.

      `#define NOMBRE(P1,P2,...,Pk) EXPRESION`

      ```
      #define CUADRADO(N)  (N)*(N)
      #define MAX(A,B)  ((A)>(B)?(A):(B))
      ...
      int i, j, k;
      k= CUADRADO(i+1);
      j= MAX(E*PI, k);
      ...
      ```

        - Es recomendable usar paréntesis para evitar efectos indeseados. ¿Qué pasaría si pusiéramos: `#define CUADRADO(N) N*N` ?

- **Comando: `#include`**. Incluye el contenido de un fichero en el punto del programa donde se encuentra el comando. Normalmente será un fichero de cabecera de una librería (extensión **.h**), pero puede ser cualquier fichero.

  `#include <NOMBRE_FICHERO>` → Para librerías de sistema

  `#include "NOMBRE_FICHERO"` → Para librerías y ficheros propios

    - **`stdio.h`** → Librería estándar de entrada/salida
    - **`stdlib.h`** → Librería de funciones y tipos estándar
    - **`time.h`** → Funciones y tipos para manejo del tiempo
    - **`math.h`** → Funciones matemáticas
    - **`mem.h`** → Manejo de memoria
    - **`...`**

- **Comandos de compilación condicional.** Permiten añadir o quitar trozos de código, en tiempo de compilación, según cierta condición. La condición es siempre del tipo "macro definido" o "macro no definido".

```
#define DEBUG
      ...
#ifdef DEBUG
      printf("Pasa por aquí.");
#endif
      ...

      ...
#ifndef OPTIMIZAR
      i= i + 1;
      j= j*(i+1);
      i= j;
#else
      i= j*= (++i + 1);
#endif
```

- **Indefinir un macro. `#undef` NOMBRE**

- **Ejemplo de uso.** Evitar que se incluya un fichero varias veces.

```
#ifndef _LIBRERIA_PILAS
#define _LIBRERIA_PILAS
.....
.....
#endif
```

- **Ejemplo.** Probar compilando con "`gcc fichero.c`" y "`gcc -DDEBUG fichero.c`".

```
#include <stdio.h>

#define MAX_INT  ((unsigned)(~0)>>1)
#define MENSAJE "Mensaje predefinido"
#define MAX(A,B)  ((A)>(B)?(A):(B))

int main (void) {
  int i, j;
#ifdef DEBUG
  printf("Maximo entero: %d\n", MAX_INT);
  printf("%s\n", MENSAJE);
#endif
  printf("Introduce dos enteros:");
  scanf("%d %d", &i, &j);
  printf("Maximo: %d\n", MAX(i, j));
  return 0;
}
```

- Significado de la opción -D: "`man gcc`".

2. Programación modular

- La **programación modular** permite descomponer la complejidad de una aplicación en distintos trozos. Un **módulo** o **paquete** contiene un conjunto de funcionalidades relacionadas.

- **Ejemplo.** Aplicación de edición de textos. El código no va en un solo fichero, sino que se descompone en módulos:
  - **Tipos**. Incluye la implementación de los tipos de datos básicos necesarios en el programa (pilas, listas, diccionarios, etc.).
  - **Interface**. Funciones del interface de usuario (entrada y salida).
  - **Ficheros**. Funciones para el manejo de los documentos (lectura y escritura).
  - **Diccionario**. Código relacionado con el corrector ortográfico.
  - **Principal**. Contiene la estructura final de la aplicación, haciendo uso de todos los demás módulos.

- En C no existe, de forma explícita, el concepto de módulo o paquete. Un programa C puede estar compuesto por varios **ficheros**. En uno de ellos tiene que estar definida la función `main`.

- **Programación modular en C.** Para cada módulo tendremos dos ficheros:

  - **Fichero de cabecera (header)**. Extensión **.h**. Contiene la definición de los tipos de datos y la declaración de las funciones (cabecera).

  - **Fichero de implementación.** Extensión **.c**. Contiene la implementación de las funciones declaradas en el fichero de cabecera.

- **Ejemplo.** Módulo Tipos, del editor de textos.

tipos.h

```
#ifndef _LIB_TIPOS
#define _LIB_TIPOS

#define MAX_CAPACIDAD 100

extern int estado_error;

typedef struct {
   int tope;
   int datos[MAX_CAPACIDAD];
} pila;

void push (pila *p, int v);
void pop (pila *p);
int top (pila *p);

...

#endif
```

tipos.c

```
#include "tipos.h"

int estado_error= 0;

void push (pila *p, int v)
{
  if (p->tope>=MAX_CAPACIDAD) {
    estado_error= 1;
    return;
  }
  p->datos[p->tope++]= v;
}

void pop (pila *p)
{
  p->tope--;
  estado_error|= p->tope<0;
}
...
```

principal.c

```
/* Módulo principal */

#include <stdio.h>
#include <stdlib.h>

#include "tipos.h"
#include "interface.h"

int main (int na, char *va[])
{
  pila *p1;
  ...
}
```

- **Consideraciones de interés.**

1. Todas las macros y tipos de datos públicos (los que queremos que utilicen los usuarios) deben definirse en el fichero de cabecera del módulo.

2. Las funciones públicas también aparecerán en el fichero cabecera, pero sólo debe aparecer la declaración, es decir, la cabecera. La implementación de las funciones viene en el fichero de implementación.

3. En el fichero de implementación se hará normalmente un `#include` del fichero de cabecera del mismo módulo (para usar las macros y tipos definidos).

4. En el caso de las variables públicas, la variable debe estar definida (y, en su caso, inicializada) en el fichero de implementación. En el fichero de cabecera se pondrá: `extern tipo nombre;` (variable externa).

5. Es aconsejable documentar bien ambos ficheros. Documentación del fichero cabecera: de cara al usuario. Documentación del fichero de implementación: de cara a posibles modificaciones posteriores.

6. Si un módulo usa otro módulo, hará un `#include` del fichero correspondiente. Este `#include` puede estar en el fichero cabecera del primero.

7. Conviene usar `#ifndef ... #endif` en el fichero cabecera, para evitar que un fichero se incluya varias veces.

- **Compilación de la aplicación.** Supongamos el editor de textos. Al compilar se deben indicar todos los ficheros de implementación. No es necesario incluir los ficheros cabecera.

```
>> gcc tipos.c interface.c ficheros.c principal.c -o myword
```

3. El programa make

- La generación de un programa implica **dos pasos**:
  - **Compilación:** Dado un fichero de **código fuente** (**.c**) generar un fichero de **código objeto** (**.o**).
  - **Linkado o enlace:** Dado uno o varios ficheros de **código objeto** (**.o**) producir un **fichero ejecutable**.

- Por defecto, **gcc** hace los dos pasos sin generar el fichero de código objeto.

- Es posible hacer que genere el fichero **.o**. **Opción -c**: compilar sólo, sin enlazar.

```
>> gcc -c tipos.c          → Genera tipos.o
```

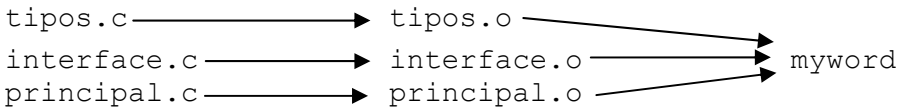
- Al compilar y enlazar un programa se pueden utilizar, y mezclar, ficheros fuente y objeto.

```
>> gcc tipos.o interface.c ficheros.o principal.c -o myword
```

- **Compilación en programas de gran tamaño:** muchos módulos, código fuente largo, mucho tiempo de compilación, modificaciones frecuentes, ...

- **Solución:** generar el fichero objeto de cada módulo por separado. Cuando se cambia un módulo, recompilar ese módulo y lincar el programa principal.

```
>> gcc -c tipos.c
>> gcc -c interface.c
>> gcc -c ficheros.c
>> gcc -c principal.c
>> gcc tipos.o interface.o ficheros.o principal.o -o myword
```



- **Problema:** si se modifica un fichero cabecera (.h) o fuente (.c) otros módulos se pueden ver afectados y hace falta recompilarlos. ¿Cuáles?

- **Programa make**. Ayuda a automatizar el proceso de compilación de un programa con muchos módulos. Busca y procesa un fichero llamado `makefile`. Estructura del fichero `makefile`:

```
OBJETIVO: FICHERO1 FICHERO2  ... FICHEROP
<Tabulador>COMANDO
```

- **Significado:** El fichero `OBJETIVO` debe recompilarse cuando se modifiquen `FICHERO1` o `FICHERO2`, ..., o `FICHEROP` (estos se denominan dependencias o

prerrequisitos). La forma de obtener este fichero `OBJETIVO` es ejecutando `COMANDO`.

```
myword: tipos.o interface.o ficheros.o principal.o
<Tabulador>gcc tipos.o interface.o ficheros.o principal.o -o myword

tipos.o: tipos.c
<Tabulador>gcc -c tipos.c

interface.o: tipos.h interface.c
<Tabulador>gcc -c interface.c

...
```

- Al ejecutar "`make`", busca el fichero `makefile`. De todos los objetivos, intenta generar el primero que aparezca. Si necesita otros objetivos, los procesa también.

- **Ejemplo.** Editar los siguientes ficheros y compilar usando `make`.

arit.h

```
#ifndef _LIB_ARIT
#define _LIB_ARIT

int suma (int a, int b);
int mult (int a, int b);

#endif
```

arit.c

```
#include "arit.h"

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

int mult (int a, int b)
{
   return a*b;
}
```

calc.c

```
#include <stdio.h>
#include "arit.h"

int main(void)
{
  int a, b;
  printf("Operando 1: ");
  scanf("%d",&a);
  printf("Operando 2: ");
  scanf("%d",&b);
  printf("%d+%d=%d\n",a,b,suma(a,b));
  printf("%d*%d=%d\n",a,b,mult(a,b));
  return 0;
}
```

makefile

```
#
# Programa de calculadora
#

calc: calc.o arit.o
<TABULADOR>gcc -o calc calc.o arit.o

calc.o: calc.c
<TABULADOR>gcc -c calc.c

arit.o: arit.c
<TABULADOR>gcc -c arit.c
```

Ejercicios

1. Encuentra y corrige el error existente en el siguiente programa. Antes de ejecutarlo deduce el resultado que tendrá.

```
#include <stdio.h>

#define CUADRADO1 (A) A*A
#define CUADRADO2(A) (A)*(A)
#define CUADRADO3(A) ((A)= (A)*(A))

int main (void) {
  int i= 3, j= 4;
  printf("CUADRADO1(%d+1)= %d\n", i, CUADRADO1(i+1));
  printf("CUADRADO2(%d+1)= %d\n", i, CUADRADO2(i+1));
  CUADRADO3 (j);
  printf("CUADRADO3(%d)= %d\n", j, CUADRADO3(j));
  return 0;
}
```

2. Dado el programa escrito para el ejercicio 3 de la 3º sesión del seminario (cálculo del producto de dos matrices) dividir el código en tres módulos: **Ficheros** (funciones de entrada salida), **Matematico** (tipo de datos matriz y producto de matrices) y **Principal** (programa principal). Crea un fichero `makefile` para generar automáticamente el programa ejecutable.

3. Escribe un programa que dado un nombre de fichero, que se pasará como parámetro en la línea de comandos, haga que la primera letra de cada palabra esté en mayúsculas. Se deberá poder compilar en "modo debug", mostrando en ese caso los principales pasos de ejecución. Crea un fichero `makefile` para generar automáticamente el programa ejecutable.