Unidad 8
Utilización Avanzada del Lenguaje C
Desarrollo de la unidad:
Prácticas:
Ejercicios:
Conceptos:
- Depuración y análisis de algoritmos
- Preprocesador (Compilación condicional)
- Modularidad en C, la herramienta make
- Control de versiones
- En C y en ensamblador, acceso a dispositivos,
- Llamadas al sistema y a la BIOS

Otros:
- Gráficos VGA y API de Windows
- Acceso a Base de Datos
- Construcción de Librerías
- Comunicación entre procesos mediante PIPE y Sockets

Fases en la construcción de un programa

Edición – Preprocesado – Compilación – Enlace/ Montado
.c /cpp/h/hpp -> .i -> .obj +. asm + .lib -> .exe/.com /.dll

Fase	**BORLAND**	**GNU**
Prepocesado	cpp	cpp
Compilación	bcc , bcc32	gcc
Enlace /linkado	tlink	ld

Preprocesador

- Expande los includes #include <stdio.h>
- Sustituye los definiciones por su valor # define MAXNOTA 10
- Permite definir expresiones y funciones

```
#define MAX( X,Y) ( X > Y )? X : Y;
#define BYTEALTO (X ) ( X >> 8 )
#define PRIMERBIT ( X) ( X & 0x0001)
#define PI 3.1416
#define AREA ( R ) ( PI * ( R * R ) )
#define INICIO {
#define FIN }
```

Compilación condicional

La compilación condicional permite incluir distinto código fuente en función de al existencias de determinadas definiciones. Aplicación: Versiones para distintas plataformas, hardware, Sistema Operativos, Depuración.

```
// Distinto include en función de sistema operativo
#ifdef  UNIX
#include <curses.h>
#else
#include <conio.h>
#endif

// La versión de depuración imprime una traza
#ifdef DEBUG
printf(" El valor de status es = %d \n", status);
#endif

// Distinta definición de tipos
#ifdef POCAMEMORIA
char buffer[512];
#else
char buffer[1024];
#endif
```

Constantes predefinidas del preporcesador:

- `__FILE__` **Nombre del fichero**
- `__LINE__` **Línea de código**
- `__DATE__` **Fecha de compilación**
- `__TIME__` **Hora de compilación**
- `__STDC__` **Compilación modo estándar ANSI C**

Aplicación: Facilitar las labores de depuración: construcciones de función de Gestión de Error y Avisos, trazas.

```
#define  Merror(X)    Infoerror(X, __FILE__, __LINE__);
```

Muchas aplicaciones llevan un registro automático de incidencias, fallos, errores recuperables, errores fatales, que se pueden consultar para depuración o supervisión de la aplicación. Siempre hay que evitar que un programa falle de forma irrecuperable y no tengamos ninguna información sobre la causa y el lugar donde se ha producido el error.

Funciones de terminación:

void exit (int status)	Provoca la terminación del programa devolviendo al S.O. el valor indicado en status. Normalmente 0 indica terminación correcta y cualquier otro valor un fallo concreto
int aexit(void *funcion ())	Permite registrar una función de usuario que se invocará automáticamente cuando se el programa termina por exit.
void abort()	Provoca la terminación inmediata del programa indicando que se

	ha producido de forma anormal, generando información para depuración.
assert (<condición lógica>)	Provoca la terminación de programa SI NO se cumple la condición lógica. Genera información para depuración.
signal	Permite asignar una función de usuario que trate algunos fallos o interrupciones provocadas por el S.O.o por fallos de ejecución como violación de segmento, división por cero, interrupción de teclado, etc.

Opciones de compilación y depuración

```
Borland C++ 5.2 Copyright (c) 1987, 1997 Borland International
Mar 19 1997 17:29:40
Syntax is: BCC [ options ] file[s]     * = default; -x- = turn switch x off
  -1      80186/286 Instructions    -2    * 80286 Protected Mode Inst.
  -3      80386 Instructions        -Ax     Disable extensions
  -B      Compile via assembly      -C      Allow nested comments
  -Dxxx   Define macro              -Exxx   Alternate Assembler name
  -Hxxx   Use pre-compiled headers  -Ixxx   Include files directory
  -K      Default char is unsigned  -Lxxx   Libraries directory
  -M      Generate link map         -N      Check stack overflow
  -Ox     Optimizations             -P      Force C++ compile
  -R      Produce browser info      -RT   * Generate RTTI
  -S      Produce assembly output   -Txxx   Set assembler option
  -Uxxx   Undefine macro            -Vx     Virtual table control
  -X      Suppress autodep. output  -Yx     Overlay control
  -Z      Suppress register reloads -aN     Align on N bytes
  -b    * Treat enums as integers   -c      Compile only
  -d      Merge duplicate strings   -exxx   Executable file name
  -fxx    Floating point options    -gN     Stop after N warnings
  -iN     Max. identifier length    -jN     Stop after N errors
  -k    * Standard stack frame      -lx     Set linker option
  -mx     Set Memory Model          -nxxx   Output file directory
  -oxxx   Object file name          -p      Pascal calls
  -po     fastthis calls            -tWxxx  Create Windows app
  -u    * Underscores on externs    -v      Source level debugging
  -wxxx   Warning control           -xxxx   Exception handling
  -y      Produce line number info  -zxxx   Set segment names
```

Opciones más interesantes

-S Produce código ensamblador
-D Define una macro –U elimina la definición de una macro
-c Sólo compila
-Ox Optimiza
-jN Para después de N errores
-gN Para después de N avisos
-v Genera información para depuración: Incluye información de instrucciones y variables
-I Directorios de incluyes adicionales
-L Directorios de Librerías adicionales

MODULARIDAD

Concepto:

Las aplicaciones reales implican la resolución de múltiples problemas con cientos de algoritmos más o menos complejos, por lo que es habitual que contenga miles de instrucciones.

La posibilidad de descomponer un programa en varios partes o módulos constituye una necesidad básica para afrontar con más facilidad el desarrollo de aplicaciones complejas.
Por otra parte es frecuente que un mismo algoritmo tenga que ser utilizados en varias partes de un mismo programa o en distintas aplicaciones. La descomposición de un programa en varios módulos nos facilita la reutilización del código así como al creación de librerías para evitar reinventar la rueda cada vez que nos enfrentemos a un nuevo programa. La posibilidad de probar cada módulo de forma independiente permite una primera corrección de errores que se traduce una mayor fiabilidad de la aplicación.

La descomposición de una aplicación en distintos módulos es los que se conoce con el nombre de diseño arquitectónico o de alto nivel. El objetivo es lograr módulos con una gran independencia funcional, es decir que no dependan de los algoritmos y definiciones de datos de otros módulos. Esto se consigue con diseñando módulos con alta cohesión interna (funcional, secuencial, comunicación, datos, procedimental, etc) y bajo acoplamiento (control, datos, llamadas, formatos, etc).

Módulo: Conjunto de funciones y definiciones de datos que realizan unas operaciones especificas
Ej.- Módulo para la gestión de una Cola, Pila, Árbol, La impresora, el puerto serie, Indexación de ficheros, etc.

Librería: Conjunto de módulos que trabajan sobre un tema común
Gráficos, Sonidos, Ordenación de ficheros, Acceso a bases de datos, etc

Las librerías pueden ser estáticas, si se incluyen completamente dentro del código del programa o dinámicas si sólo se cargan a memoria cuando se llaman. Generalmente las librerías dinámicas suelen ser compartidas. Esto permite que una misma librería sólo se cargue una vez en memoria aunque sea utilizada por varios programas en ejecución. Esto supone un ahorro de memoria RAM y de tiempo de carga de los programas

Estructura conceptual de un Módulo:

Definición o Interfaz / Externo : Define lo que hace y como usarlo
Implementación / Interno : Se codifica los algoritmos que realizan la operaciones

El objetivo es poder usar un módulo sin conocer su implementación

Interfaz: *Definición Estructuras de datos y funciones que realiza en módulo.* (PARTE PÚBLICA)
Implementación: *Estructuras de datos internas y codificación de algoritmos que realizan las funciones definidas en el interfaz.* (PARTE PRIVADA)

Los lenguajes modernos que permiten la programación orientada a objeto POO (C++, Delphi, Java, C#, etc.) están diseñados para facilitar el diseño modular mediante la definición de jerarquías de clases

MODULARIDAD EN LENGUAJE C.

Definición de un módulo / Interfaz : **fichero.h**

En un fichero de cabecera se define:
- Constantes
- Tipos de datos
- Funciones públicas que realiza el módulo

Implementación de un módulo **fichero.c**

En un fichero C que implementa un módulo
- Se incluye el fichero interfaz #include
- Codificación de las funciones públicas
- Codificación de funciones privadas o auxiliares
- Definición de datos internos

Dentro de un módulo (fichero.c) las funciones y variables que utilizamos pueden ser:
- **externas** (Implementadas y definidas en otro módulo o fichero)
  Generalmente estas funciones se incluyen su definición en los módulos que usamos
  Ej.- printf (En *stdio.h*) o extern int cosa

- **locales** (Definidas e implementadas en el propio fichero)
  - públicas : Otros módulos o ficheros pueden hacer uso de esa variable o función
  - privadas : Sólo pueden ser usadas dentro del mismo fichero o módulo

(¡Ojo!: No confundir con variables locales y globales)

Si no indicamos nada, por omisión cualquier variable global definida en un módulo (fichero.c) es pública al igual que cualquier función. Es decir, puede ser utilizada por cualquier otro módulo. Para evitar esto debemos incluir el prefijo **static,** para definirla como privada por lo que sólo podrá ser utilizada dentro del módulo. Es habitual incluir las siguientes definiciones para caracterizar todas las funciones que se implementan en un módulo:

```
#define PRIVATE  static
#define PUBLIC

PUBLIC   void ConsultarSaldo( int & valor ) {…
PRIVATE void  RectificarCuenta ( int cantidad ) {….
```

Para utilizar funciones de otros módulos lo normal es incluir la su fichero .h correspondiente, o bien definir la variable o función externa con el prefijo **extern,** con lo que indicamos al compilador que la variable o función se encuentra definida en otro módulo.

Ejemplos
- Utilización de un módulo de ventanas de texto para mostrar el contenido de un fichero.
- Descomposición modular de programa de mantenimiento de artículos.

- Compilación y linkado de proyectos:

La mayor parte de los entorno de desarrollo integrados (IDE) como Borland C++ o Visual C++, tienen su propias herramientas para gestión de proyectos más o menos flexibles de tal forma que el compilador conoce que ficheros componen el proyecto y como deben ser compilarlos.

Para proyectos más complejos lo más habitual es compilar la aplicación mediante la herramienta **make.** El programa make es una aplicación muy utilizada, originaria de mundo UNIX, que permite definir detalladamente como compilar, instalar o configurar un proyecto complejo a partir de un fichero donde se describe los elementos y relaciones entre los archivos a generar y las acciones a realizar. Normalmente este fichero tiene el nombre de *makefile*. En la Internet la mayor parte de software de código abierto se compila mediante esta utilidad.

La sintaxis básica de los archivos makefiles es la siguiente:

```
Objetivos: ficheros necesarios / objetivos previos necesarios
      Acciones a tomar
```

Los objetivos son normalmente:

- La compilación y montado de proyecto
- La instalación de la aplicación
- El borrado de ficheros intermedios
- La invocación a la herramienta de control de versiones
- La compilación de un modulo o librería concreto

El programa make realiza la acción si no existe el objetivo o si el objetivo tiene una fecha anterior que los ficheros de que depende.

Ejemplo de fichero makefile

```
#OPCIONES DE COMPILACION
CFLAGS= -O2 -w -DDEBUG -I/usr/proyectoxx/includes
#OPCIONES DE MONTADO
LFLAGS= -L/usr/proyectoxx/mislib -lcrupto

progcliente: progServidor.o  moduloc.o
gcc $LFLAGS progServidor.o moduloc.o -llibTCP -o progS

progservidor: moduloCliente.o moduloc.o
gcc  moduloServidor.o moduloc.o -llibTCP -o progC

modulo1.o: moduloc.c comun.h
gcc $CFLAG -c moduloc.o

progcliente.o: progcliente.o comun.h
gcc $CFLAG -c progcliente.o

progservidor.o: progservidor.o comun.h
gcc $CFLAG -c progservidor.o

all: progC progS
```

```
clear:
	rm *.o

install:
	medir /usr/bin/proyecto
	cp progS  /usr/bin/proyecto
	cp progC  /usr/bin/proyecto
	mkdir /var/log/miprog
	touch /var/log/miprog/info.log
```

- **Control de versiones**

Normalmente en la programación de una aplicación informática participan varias personar que son coautores de los distintos módulos que componen la aplicación. Teniendo presente que el código fuente es el documento más importantes de una aplicación y que toda aplicación se modifica tarde o temprano, debemos controlar quien ha realizado cada módulo, que cambios se han introducidos, en que fecha, que errores corregían o que mejorar incorporaban.

El control de versiones es fundamental en cualquier proyecto informático, en especial cuando una misma aplicación puede estar instalada en múltiples usuarios y plataformas.

Información de cada módulo:

- Autores
- Fecha
- Versión
- Revisión
- Cambio introducido

Generalmente casi todos los productos software mantienen versiones de desarrollo y versiones comerciales. Cuando una versión de desarrollo ha sido suficientemente probada, a veces se distribuye como *versión beta* para que la prueben determinados clientes antes liberar una versión completa y distribuirla libremente al mercado.

En los sistemas UNIX las herramientas más utilizadas son los sistemas **RCS** (Sistema de Control de Revisión) y **CVS** (Sistema de versiones concurrentes).

En sistema RCS es el más sencillo y se basa en una serie de comandos que pueden ser ejecutados directamente o mediante una herramienta de desarrollo que los invoca.

$ci archivo -- Incluye una nueva versión en el repositorio
$co archivo -- Extrae una versión concreta del repositorio
$rcsdiff archivo -- Para comparar entre distintas versiones
$rlog archivo -- Muestra información sobre todas las versiones almacenadas del archivo

El sistema CVS está basado en el anterior pero permite que una aplicación este distribuida en varios directorios en una misma máquina o en máquinas distintas conectadas en red y donde varios usuarios puedan trabajar simultáneamente sobre una misma versión de un archivo.

Gracias a Internet, actualmente existen muchos proyectos de software abierto donde participan múltiples programadores desde distintos lugares del mundo que trabajan conjuntamente. La mayor parte del software libre (kernel de Linux, proyecto GNU, Debian, sourceforge, apache) se realiza gracias a herramientas de este tipo.

La mayor parte de las empresas de desarrollo de software suelen utilizar sus propias herramientas de gestión de versiones aunque en la mayoría de las ocasiones están basadas en los sistemas anteriores.

- **Herramientas de depuración y optimización**

Depuradores:

Permite controlar la ejecución de un programa para detectar los posibles errores. Para trabajar con estos programas generalmente se precisa un modo de compilación especial para que incluya información sobre código fuente en el ejecutable, en otro caso sólo tendremos el código ensamblador sin ningún nombre de identificador de los datos o funciones que usemos. Muchos depuradores están integrados dentro del entorno de desarrollo, pero también pueden funcionar de forma independiente.

Operaciones:
- Fijar punto de parada (breakpoint)
- Controlar la ejecución: paso a paso, mientras no se cumpla una condición, hasta el siguiente punto de parada, etc.
- Consulta de variables y expresiones
- Modificación de valor de variables
- Consulta el estado de la CPU, memoria, registros, pila

Modos de trabajo

- Arrancando directamente un programa para controla su ejecución, la forma más habitual.
- Capturando y controlando un programa que ya estaba en ejecución y queremos depurar.
- Depurando un programa que ha terminado anormalmente a partir un archivo con la imagen de programa en ejecución. (core dump)

Analizadores o profiling

Son herramientas que ejecutan el programa realizando un registro exhaustivo de las operaciones que realiza: llamadas a rutinas, librerías, sistema operativo, tiempo de ejecución, tiempo en espera, etc. Con esta información estadística podemos detectar y localizan las partes del código que provocar un mayor retardo de nuestra aplicación con objeto de rediseñar los algoritmos para una mayor optimización. Este tipo de herramienta es muy utilizada cuando trabajamos con aplicaciones de tiempo real, que son muy dependientes del tiempo de ejecución.

El C y el ensamblador

El leguaje C a pesar de ser un lenguaje de alto nivel permite realizar operaciones típicas de bajo nivel como los operadores de bits. Es común que en la mayoría de los compiladores permitir insertar dentro de un programa en C código directamente escrito en ensamblador.

```
// Entero sin signo de 16 bits en procesadores Intel
unsigned short int dato = 10;
//Incremento dato utilizando el registro de procesador AX
asm {
    mov ax, dato;
    inc ax
    mov dato, ax;
    }
```

El ensamblador es a veces necesario cuando queremos obtener el máximo de eficiencia en recursos (memoria, tiempo de CPU, etc.) o cuando tenemos que interactuar directamente con el hardware, por ejemplo cuando realizamos el software de control de un dispositivo (driver). Un proyecto puede combinar módulos escritos en C y módulos escritos en ensamblador. Lo normal es que el ensamblador sólo se utilice cuando es imprescindible, para facilitar la depuración y la portabilidad. Desde un módulo de C podemos llamar a una función escrita en ensamblador o viceversa.

Acceso directo a llamadas a sistemas.

Desde C es habitual tener una funciones que permitan invocar directamente una interrupción software, ya sea una llamada al sistema operativo (DOS, Windows, GNU/linux, etc) o a la funciones de la propia ROM-BIOS en ordenadores con arquitectura PC bajo MS-DOS. Lo normal es que sistema operativo ofrezca un interfaz mediante su correspondiente fichero de cabecera (Ej.- dos.h, fcntl.h etc.) En algunos casos podemos rellenar directamente una estructura equivalente a los registros de procesador y invocar la interrupción software especifica.

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

/* deletes file name; returns 0 on success, nonzero on failure */
int delete_file(char near *filename)
{
   union REGS regs;
   int ret;
   regs.h.ah = 0x41;
   regs.x.dx = (unsigned) filename;
   ret = intdos(&regs, &regs);  /* delete file */
   /* if carry flag is set, there was an error */
   return(regs.x.cflag ? ret : 0);
}
int main(void)
{
   int err;
   err = delete_file("NOTEXIST.$$$");

   if (!err)
      printf("Able to delete NOTEXIST.$$$\n");
   else
      printf("Not Able to delete NOTEXIST.$$$\n");
   return 0;
}
```