Guion de prácticas 2

Acceso a memoria a nivel de bit

Marzo de 2017

DECSAI

Metodología de la Programación

Curso 2016/2017

Índice

1. Definición del problema

En esta práctica aprenderemos a acceder a cada byte de memoria, bit a bit. Para ello, debemos implementar ciertas operaciones básicas con los bits como activar/desactivar bits determinados, o activar/desactivar todos los bits simultáneamente.

Supondremos que cada bit puede estar sólo en dos estados, encendido (1) y apagado (0), y los 8 bits se representan como un byte. En C++, las variables de tipo `unsigned char` ocupan exactamente un byte, por lo cual, el bloque de bit se representará como una variable de dicho tipo. Para facilitar su uso y abstraernos de la representación interna, podemos utilizar un "alias" para `unsigned char` y hacer:

```
typedef unsigned char Byte;
```

y de esta manera definir variables de tipo `Byte`.

2. Objetivos

El desarrollo de esta práctica pretende servir a los siguientes objetivos:

- repasar conceptos básicos de funciones,
- practicar el paso de parámetros por referencia,
- practicar el paso de parámetros de tipo array,
- entender el uso de operaciones a nivel de bit,
- reforzar la comprensión de los conceptos de compilación separada y utilización de makefile.

3. Operadores a nivel de bit

Las funciones de manejo de bits concretos (encendido, apagado, consulta de estado, etc.) necesitarán acceder y modificar posiciones específicas de la variable de tipo `Byte`. Es decir, tendremos que manipular los bits de la variable de manera independiente.

El lenguaje C++ ofrece un conjunto de operadores lógicos **a nivel de bit** para operar con diversos tipos de datos, en particular con **caracteres** y **enteros**. Los operadores son:

- operadores binarios, donde la operación afecta a los bits de dos operandos (de tipo `unsigned char`, en nuestro caso):
  - and $(op1 \ \& \ op2)$: devuelve 1 si los dos bits valen 1
  - or $(op1 \mid op2)$: devuelve 1 cuando al menos 1 de los bits vale 1
  - or-exclusivo $(op1 \ \hat{} \ op2)$: produce un valor 1 en aquellos bits en que sólo 1 de los bits de los operandos es 1

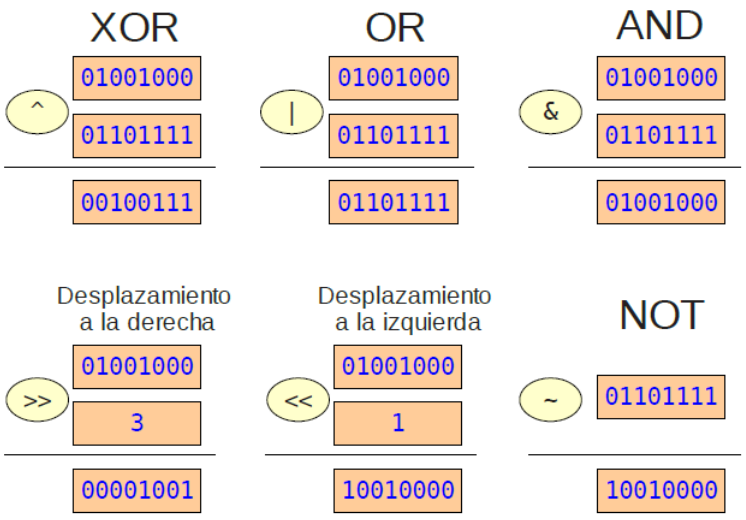


Figura 1: Ejemplos de operaciones a nivel de bit

- desplazamiento a la derecha un determinado número de bits (posiciones) $(op1 >> desp)$. Los $desp$ bits más a la derecha se perderán, al tiempo que se insertan bits con valor 0 por la izquierda
- desplazamiento a la izquierda $(op1 << desp)$: ahora los bits que se pierden son los de la izquierda y se introducen bits a 0 por la derecha

- operador unario de negación ($\sim op1$): cambia los bits de $op1$ de forma que aparecerá un 0 donde había un 1 y viceversa.

La Fig. 1 muestra un ejemplo de dichas operaciones.

4. Módulo Byte

El módulo a implementar contendrá las siguientes funciones (puedes ver una explicación con más detalle en el fichero `Byte.h` incluído en la carpeta `include`).

```
typedef unsigned char Byte;

// enciende el bit pos del Byte b
void onBit(Byte &b, int pos);

// apaga el bit pos del Byte b
void offBit(Byte &b, int pos);

// devuelve el estado del bit pos
//(encendido = true, apagado = false)
bool getBit(Byte b, int pos);

// enciende todos los bits
```

```
void onByte(Byte &b);

// apaga todos los bits
void offByte(Byte &b);

// enciende los bits según la configuracion de v.
// el tamaño de v debe ser 8
void asignarByte(Byte &b, const bool v[]);

// asigna en v el estado de cada bit
void volcarByte(Byte b, bool v[]);

// devuelve en posic un vector con las posiciones
// de los bits que están encendidos.
// En cuantos se devuelve el número de bits encendidos
//(número de elementos ocupados en el vector posic).
void encendidosByte(Byte b, int posic[], int &cuantos);

// Imprime los bits del byte b, a la izquierda los más
   significativos
// a la derecha los menos significativos
void imprimirByte(Byte b);
```

Como consultar y modificar el estado de un bit

Las operaciones de consulta, apagado y encendido de bits se traducen a operaciones a nivel de bits. La consulta se corresponde con una lectura y el encendido/apagado con una operación de asignación.

Las operaciones de asignación y lectura de bits se pueden dividir en dos pasos: a) generar una "máscara" (un byte con una secuencia determinada de 0's y 1's) y b) aplicar un operador lógico.

Por convención, asumiremos que el bit más a la derecha representa el primer bit y tiene asignada la posición 0, mientras que el bit de más a la izquierda ocupa la posición 7.

Consulta del estado de un bit

Supongamos que queremos averiguar si el bit en la posición 5 se encuentra encendido. Esto es equivalente a averiguar si el bit en la posición 5 del bloque de bits `b` está en 1. Para ello, debemos seguir los siguientes pasos.

1. Crear una máscara (un byte específico) que contenga sólo un 1 en la posición de interés. En este caso: **0010 0000**. Para ello:

   a) Partimos del valor decimal **1** (su codificación en binario es **0000 0001**) [1]

[1] En C++ se pueden escribir literales en hexadecimal precediéndolos por `0x`, p. ej.

b) lo desplazamos a la izquierda el número de posiciones deseadas (en este caso 5). Utilizando el operador de desplazamiento a izquierda hacemos: `unsigned char mask = 0x1 <<` 5.

2. hacer una operación **AND** entre `b` y `mask`.
3. Si el resultado es distinto de cero, entonces el bit 5 es un 1 y por tanto, el bit está encendido. Caso contrario es un cero y el bit está apagado.

Apagar y Encender de un bit

Apagar un bit implica poner a 0 el bit correspondiente. Supongamos que deseamos apagar el bit de la posición 2. Para poner el bit `k`=2 del bloque de bits `b` a cero, se deben seguir los siguientes pasos.

1. generar la máscara `mask` con valor 1111 1011 Para ello:
   a) generar primero una máscara 0000 0100 como hemos explicado antes.
   b) aplicarle el operador de negación NOT.
2. hacer un AND entre `mask` y `b` y guardar el resultado en `b`.

Encender un bit implica poner a 1 el bit correspondiente. Por ejemplo, para poner el bit `k`=2 a 1 del bloque de bits `b` haremos:

1. generar la máscara `mask` con valor 0000 0100.
2. aplicar el operador OR entre `mask` y la variable `b`. El resultado se guarda en `b`.

Secuencias de Animación

El bloque de bits puede mostrar una “animación” si se encienden y apagan los bits en un orden determinado y se muestran sus valores. A continuación se muestran dos ejemplos.

Ejemplo 1	Ejemplo 2
11111111	11111111
01111111	01111110
10111111	00111100
11011111	00011000
11101111	00000000
11110111	00011000
11111011	00111100
11111101	01111110
11111110	11111111

32 decimal es `0x20`, o en octal precediéndolos por `0`, p. ej. `040`. Desde C++14, los literales binarios pueden representarse precediéndolos con `0b`, p. ej. `0b00100000`

5. Tareas a realizar

- Implementar las funciones indicadas en el fichero **byte.h**
- El módulo **main.cpp** ya incluye instrucciones para probar la funcionalidad básica del Byte. Extienda el módulo mostrando como utilizaría las instrucciones `on`, `off` y `imprimirByte` para generar las secuencias de animación indicadas en la sección anterior.
- Crear el archivo **makefile** con las órdenes necesarias para generar el ejecutable (**byte**) a partir de los archivos fuentes. Debe crear también los objetivos **clean** y **zip**.
- Escribir un informe donde consten los nombres y DNI de los integrantes del grupo, los problemas que hayan podido surgir durante el desarrollo de la práctica, capturas de pantalla, etc. Este informe, en formato pdf, se guardará en la carpeta `doc`.

Al completar la implementación debería obtener una salida como la mostrada en la Fig. 2.

6. Material a entregar

Cuando esté todo listo y probado el alumno empaquetará la estructura de directorios anterior en un archivo con el nombre **practica2.zip** y lo entregará en la plataforma **decsai** en el plazo indicado. No deben entregarse archivos objeto (**.o**) ni ejecutables. Para asegurarse de esto último conviene ejecutar **make zip** antes de proceder al empaquetado.

El fichero **practica2.zip** debe contener la siguiente estructura:

```
./
├── makefile
├── include
│   └── byte.h
├── src
│   ├── byte.cpp
│   └── main.cpp
├── bin
├── obj
├── lib
├── data
└── zip
```

El alumno debe asegurarse de que ejecutando las siguientes órdenes se compila y ejecuta correctamente su proyecto:

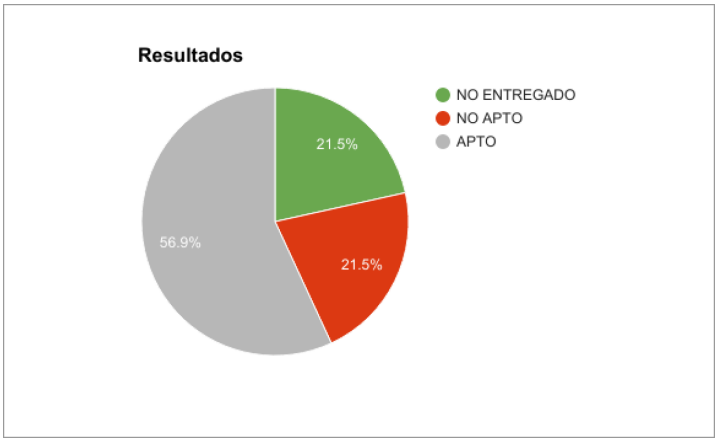
```
unzip practica2.zip
make
bin/byte
```

```
ubuntu:$ make
g++ -Wall -g -c ./src/main.cpp -o ./obj/main.o -I./include
g++ -Wall -g -c ./src/byte.cpp -o ./obj/byte.o -I./include
ar -rvs ./lib/libbyte.a ./obj/byte.o
ar: creating ./lib/libbyte.a
a - ./obj/byte.o
g++ -o ./bin/byte ./obj/main.o -lbyte -L./lib
ubuntu:$ bin/byte

byte apagado bits: 00000000

Inicializo el byte a partir de un vector de bool 10100000

Ahora enciendo los bits 0, 1 y 2 con la funcion on
10100001
10100011
10100111

Los bits encendidos estan en las posiciones: 0,1,2,5,7,

Todos encendidos: 11111111
Todos apagados: 00000000


Ejemplo 1
11111111
01111111
10111111
11011111
11101111
11110111
11111011
11111101
11111110

Ahora la animacion
Ejemplo 2
11111111
01111110
00111100
00011000
00000000
00011000
00111100
01111110
11111111
ubuntu:$
```

Figura 2: Salida esperada del programa.

Gradebook Practicas Grupo F MP1516 - Práctica 2

	Tipo de error	Frecuen
E0	#include "" detrás de #include< >	0
E1	#include <> mal colocados dentro de un .h	0
E3	#includes innecesarios (no se usan)	0
EC	Error makefile	4
EE	Errores de compilación	2
EI	Faltan #ifndef en .h	0
EM	Faltan los .h en el cuerpo de algunas de las reglas de makefile	4
ES	Reglas mal formadas en makefile	6
EU	ZIP mal	2
EV	Faltan macros en makefile	20
EW	Errores de ejecución	4
EX	main() manipulado	2
EY	Copia o deja copiar	0
EZ	Falta resumen	7
E7	Error de acceso a memoria (SEGFAULT)	2
E6	Entregado fuera de plazo	2



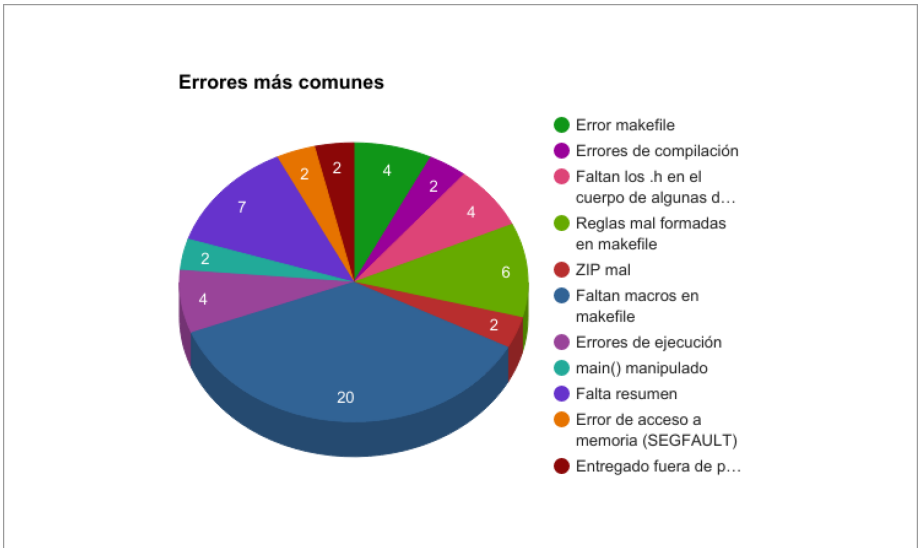


Figura 3: Estadísticas y principales errores del curso 2015-2016