UNIVERSIDAD DE GRANADA

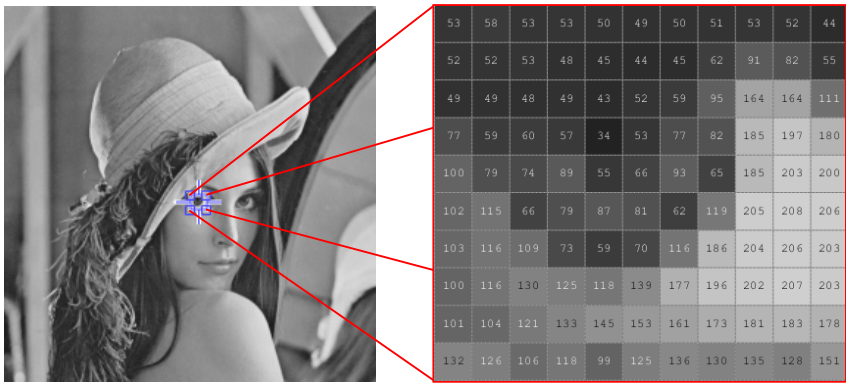
ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

DGIM

Curso 2021/2022



Guion de prácticas

Acceso a memoria a nivel de bit

Febrero de 2022

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

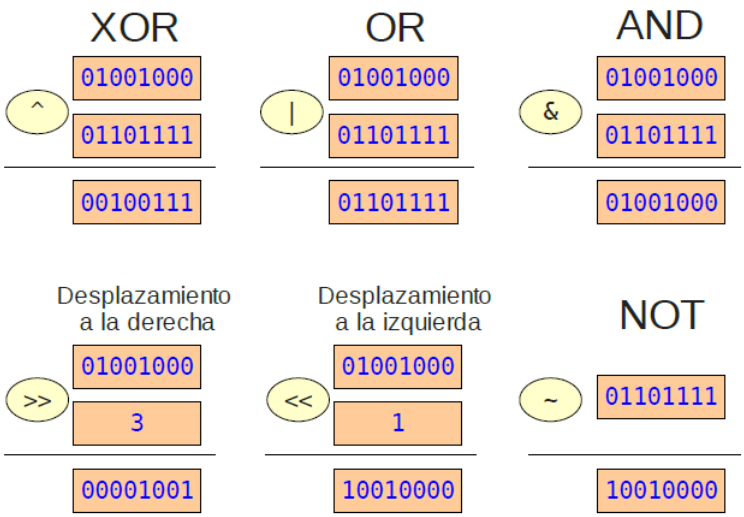


Figura 1: Ejemplos de operaciones a nivel de bit

- 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 (fichero `Byte.h` incluído en la carpeta `include`).

```
1  /**
2  @file Byte.h
3  @brief Operaciones a nivel de bit
4  @author MP-DGIM - Grupo A
5    */
6
7  #ifndef _BYTE_H_
8  #define _BYTE_H_
9
10
11 typedef unsigned char Byte; ///< A @c Byte contains the state of  8 Bits
12
13 #define MAX_BYTE 256 ///< Max number of values allowed
14 #define MAX_BYTE_VALUE 255 ///< Highest value allowed
15 #define MIN_BYTE_VALUE 0 ///< Lowest value allowed
16 #define NUM_BITS 8 ///< Number of bits
17 #define MIN_BIT 0 ///> Rightmost bit
18 #define MAX_BIT 8 ///> Leftmost bit
19 /**
20 @brief Turns on the @p position bit of @c Byte @p b
21 @param b  the @c Byte
22 @param pos   the position @p b of the bit within the Byte (0 means the rightmost position)
23 */
24 void onBit(Byte &b, int pos);
25
26 /**
```

```
27  @brief Turns off the @p position bit of @c Byte @p b
28  @param b  the @c Byte
29  @param pos   the position @p b of the bit within the Byte (0 means the rightmost position)
30  */
31  void offBit(Byte &b, int pos);
32
33  /**
34  @brief It returns the state of a certain bit inside a Byte (on = true, off = false) at the position @p pos
35  @param b  the @c Byte
36  @param pos   the position @p b of the bit within the Byte (0 means the rightmost position)
37  @retval true  when the bit at the @p pos position is 1
38  @retval false when the bit at the @p pos position is 0
39  */
40  bool getBit(Byte b, int pos);
41
42
43  /**
44  @brief It returns a string with a sequence of 0 and 1 corresponding to the state of each bit. For instance,
45   * the value 5 is represented as "00000101" since the bits 0 and 2 are set to 1
46  */
47  std::string printByte(Byte b);
48
49  /**
50  @brief Turns all the bytes on
51  @param b  the @c Byte
52  */
53  void onByte(Byte &b);
54
55  /**
56  @brief Turns all the bytes off
57  @param b  the @c Byte
58  */
59  void offByte(Byte &b);
60
61  /**
62  @brief Turns on just the bits contained in the vector. That is, if vector[i] is true,
63   * then the bit at the position @a i is turned on.
64  @param b  the @c Byte
65  @param v A vector of 8 elements representing which bits are to be turned on
66  */
67  void encodeByte(Byte &b, const bool v[]);
68
69  /**
70  @brief It dumps the bits in the Byte into the vector, such as, if bit @a i is 1, then vector[i] is true
71  @param b  the @c Byte
72  @param v A vector of 8 elements representing which bits are turned on
73   */
74  void decodeByte(Byte b, bool v[]);
75
76  /**
77  @brief It returns a vector of int, such that if vector[i]=k, then the bit at the position @a k is 1
78  @param b  the @c Byte
79  @param posic An array of positions that contains a bit set to 1
80  @param Number of bits 1 in the byte
81  */
82  void decomposeByte(Byte b, int posic[], int &cuantos);
83
84  /**
85   * @brief Shifts the byte @a n bits to the right, overflowing the rightmost bits
86   * @param b The byte
87   * @param n The number of shifts
88   */
89  void shiftRByte(Byte &b, int n);
90
91  /**
92   * @brief Shifts the byte @a n bits to the left, overflowing the leftmost bits
93   * @param b The byte
94   * @param n The number of shifts
95   */
96  void shiftLByte(Byte &b, int n);
97
98  /**
99   * @brief It returns the wheigted average of both bytes, that is
100  * ((b1*(100-merge))+(b2*merge))/100
101  * @param b1 The first Byte
102  * @param b2 The second Byte
103  * @param merge A value in [0,100]
104  * @return The wheigted average
105  */
106 Byte mergeByte(Byte b1, Byte b2, int merge);
107
108 #endif
```

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 deter-

minada 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]
   b) lo desplazamos a la izquierda el número de posiciones deseadas (en este caso 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 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`.

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

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

5. Práctica a entregar

- 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 para generar las secuencias de animación indicadas en la sección anterior.

5.1. Ejemplo de ejecución

```
byte apagado bits:
00000000

Inicializo el byte a partir de un vector de bool
00000101

Ahora enciendo los bits 0, 1 y 2 con la funcion on
10000101
11000101
11100101

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

Todos encendidos:
11111111
Todos apagados:
00000000


Ejemplo 1
11111111
11111110
11111101
11111011
11110111
11101111
11011111
10111111
01111111

Ahora la animacion
Ejemplo 2
11111111
01111110
00111100
00011000
00000000
00011000
00111100
01111110
11111111

RUN FINISHED; exit value 0; real time: 0ms; user: 0ms; system: 0ms
```

6. TESTS DOCUMENTATION FOR PROJECT MP-hotoshop

6.1. _01_Basics

6.1.1. UnitByte_onBit

1. Given a byte 00000000, activating the 0-bit gives 1
2. Given a byte 00000000, activating the 1-bit gives 2
3. Given a byte 00000000, activating the 7-bit gives 2

6.1.2. UnitByte_offBit

1. Given a byte 11111111, deactivating the 0-bit gives 254
2. Given a byte 11111111, deactivating the 1-bit gives 253
3. Given a byte 11111111, deactivating the 7-bit gives 127

6.1.3. UnitByte_getBit

1. Given a byte 11111111, querying any bit always give true
2. Given a byte 00000000, querying any bit gives false

6.1.4. UnitByte_printByte

1. A byte 11111111 prints as it is
2. A byte 00000000 prints as it is

6.1.5. UnitByte_shiftRByte

1. A byte 11111111 shifted to the right gives 127
2. A byte 11111111 shifted twice to the right gives 63
3. A byte 00000001 shifted to the right gives 0

6.1.6. UnitByte_shiftLByte

1. A byte 11111111 shifted to the left gives 254
2. A byte 11111111 shifted twice to the right gives 252
3. A byte 00000001 shifted to the right gives 2

6.1.7. UnitImage_Constructors

1. A newly created instance of Image should have 0 columns, and 0 rows.
2. A newly created instance of a 10x10 Image should have 10 columns, and 10 rows.

6.1.8. UnitImage_Width

1. A newly created instance of Image should have 0 columns, and 0 rows.
2. A newly created instance of a 10x5 Image should have 10 columns, and 5 rows.

6.1.9. UnitImage_Height

1. A newly created instance of Image should have 0 columns, and 0 rows.
2. A newly created instance of a 10x5 Image should have 10 columns, and 5 rows.

6.1.10. INTEGRATION_Byte

1. The output of the main program must match that of the assignment

6.1.11. P1

1. The output of the main program must match that of the assignment

6.2. _02_Intermediate

6.2.1. UnitByte_onByte

1. Activating a Byte gives 255

6.2.2. UnitByte_offByte

1. Deactivating a Byte gives 0

6.3. _03_Advanced

6.3.1. UnitByte_encodeByte

1. Activating bits 0,1 and 7 gives 131

6.3.2. UnitByte_decodeByte

1. A byte 131 gives true only in bits 0,1 and 7
2. A byte 131 gives true only in bits 0,1 and 7
3. A byte 131 gives true only in bits 0,1 and 7
4. A byte 131 gives true only in bits 0,1 and 7
5. A byte 131 gives true only in bits 0,1 and 7

6.3.3. UnitByte_decomposeByte

1. Decomposing byte 131 gives 3 active bits
2. Decomposing byte 131 gives 3 active bits
3. Decomposing byte 131 gives 3 active bits
4. Decomposing byte 131 gives 3 active bits

6.4. Tests run

```
[==========] Running 11 tests from 3 test suites.
[----------] Global test environment set-up.
[----------] 6 tests from _01_Basics
[ RUN      ] _01_Basics.UnitByte_onBit
[       OK ] _01_Basics.UnitByte_onBit (1 ms)
[ RUN      ] _01_Basics.UnitByte_offBit
[       OK ] _01_Basics.UnitByte_offBit (1 ms)
[ RUN      ] _01_Basics.UnitByte_getBit
[       OK ] _01_Basics.UnitByte_getBit (3 ms)
[ RUN      ] _01_Basics.UnitByte_printByte
[       OK ] _01_Basics.UnitByte_printByte (1 ms)
[ RUN      ] _01_Basics.UnitByte_shiftRByte
[       OK ] _01_Basics.UnitByte_shiftRByte (1 ms)
[ RUN      ] _01_Basics.UnitByte_shiftLByte
[       OK ] _01_Basics.UnitByte_shiftLByte (1 ms)
[----------] 6 tests from _01_Basics (8 ms total)

[----------] 2 tests from _02_Intermediate
[ RUN      ] _02_Intermediate.UnitByte_onByte
[       OK ] _02_Intermediate.UnitByte_onByte (1 ms)
[ RUN      ] _02_Intermediate.UnitByte_offByte
[       OK ] _02_Intermediate.UnitByte_offByte (0 ms)
[----------] 2 tests from _02_Intermediate (1 ms total)

[----------] 3 tests from _03_Advanced
[ RUN      ] _03_Advanced.UnitByte_encodeByte
[       OK ] _03_Advanced.UnitByte_encodeByte (0 ms)
[ RUN      ] _03_Advanced.UnitByte_decodeByte
[       OK ] _03_Advanced.UnitByte_decodeByte (2 ms)
[ RUN      ] _03_Advanced.UnitByte_decomposeByte
[       OK ] _03_Advanced.UnitByte_decomposeByte (1 ms)
[----------] 3 tests from _03_Advanced (3 ms total)

[----------] Global test environment tear-down
[==========] 11 tests from 3 test suites ran. (12 ms total)
[  PASSED  ] 11 tests.
```