

#define, const and Macros

- `#define` is often used to declare constants. Need to be careful:

- `#define` has no notion of scope.

- `const` declarations should be used instead. Especially for a constant that should exist inside a function, but not outside.

```
lectura-> cat definel.c
void f()
{
#define N 10
  int a[N];
  int i;
  for (i = 0; i < N; i++)
    a[i] = i + i;
} /* f */
void g()
{
#define N 20
  int a[N];
  int i;
  for (i = 0; i < N; i++)
    a[i] = i * i;
} /* g */
```

```
lectura-> gcc -Wall -o definel definel.c
definel.c:14:1: warning: "N" redefined
definel.c:4:1: warning: this is the location of the
previous definition
```

Solution:

Inside each function, put:

```
const int N = ...;
```

Each `N` becomes a (read-only) variable whose scope ends at the end of the function.

#define, const and Macros (continued):

- A `const` is not allowed when sizing a global array:

```
const int NCHARS = 128;
```

```
int counts[NCHARS];
```

```
int main()
{...
```

Produces the following error:
`zap.c:4: error: variably modified 'counts' at file scope`

- A `#define` is the right solution in this case. Besides, it is likely that `NCHARS` will be used elsewhere in the program (for example, to check that access does not go outside the array bounds).

```
#define NCHARS 128
```

Note: there is no semi-colon here. Why?

```
int counts[NCHARS];
```

```
int main()
{...
```

- `#define`'s "evaporate", once the pre-processor finishes, there are no definitions left.
 - `gdb` cannot print any `#define`'d values.
- Compilers tend to generate faster code for `#define`'s than for `const`'s.

#define, const and Macros (continued):

- Parameters can be specified for a `#define`. For example:

```
lectura-> cat define2.c
#include <stdio.h>

#define MIN(x,y) (((x) < (y)) ? (x) : (y))
int
main()
{
  int m1 = MIN(3, -1);
  double m2 = MIN(12.34, 10.1);
  printf("m1 = %d, m2 = %g\n", m1, m2);
  return 0;
} /* main */
lectura-> gcc -Wall -o define2 define2.c && define2
m1 = -1, m2 = 10.1
```

- Note: `MIN` does not place any constraints on its operands.

#define, const and Macros (continued):

- Macros like `min()` can be useful, but there are pitfalls. Here is an example:

```
wolf-> cat define3.c
#include <stdio.h>
```

```
#define mult(x,y) x * y
```

```
int main()
{
```

```
  printf("%d\n", mult(1+2, 2+3));
```

```
  return 0;
```

```
} /* main */
```

```
wolf-> gcc -Wall -o define3 define3.c && define3
```

```
8
```

- What is the problem?
- How to fix it?

#define, const and Macros (continued):

- ()'s are why min() looks the way it does.
- Another pitfall: expressions with side effects:

```
lectura-> cat define4.c
#include <stdio.h>
```

```
#define MIN(x,y) ((x) < (y)) ? (x) : (y)
int
main()
{
    int i = 1;
    printf("i = %d, min = %d\n", i, MIN(i+=1, 10));
    printf("i = %d\n", i);
    i = 100;
    printf("i = %d, min = %d\n", i, MIN(i+=1, 10));
    printf("i = %d\n", i);
    return 0;
} /* main */
```

```
lectura-> gcc -Wall -o define4 define4.c
define4.c: In function 'main':
define4.c:8: warning: operation on 'i'
may be undefined
define4.c:12: warning: operation on 'i'
may be undefined
lectura-> define4
i = 3, min = 3
i = 3
i = 101, min = 10
i = 101
```

- What happened? Can the macro be fixed?

gcc and #define:

- You can create a #define on gcc's command-line using the -D option:

```
lectura-> cat define5.c
#include <stdio.h>
```

```
int
main()
{
    int i, sum = 0;
    for (i = 1; i <= N; i++)
        sum += i;
    printf("sum = %d\n", sum);
    return 0;
} /* main */
```

```
lectura-> gcc -Wall -o define5 -DN=10 define5.c && define5
sum = 55
lectura-> gcc -Wall -o define5 -DN=100 define5.c && define5
sum = 5050
```

- Document this!
 - It can cause no end of frustration for someone who is trying to find the definition of the constant!
- Useful technique within a *Makefile*.

Conditional compilation:

- There are several directives related to conditional compilation:

```
#ifdef identifier
...code...
#endif
```

- If a #define has been encountered for *identifier* then the source code between the #ifdef and #endif is passed through to the compiler. If not, the lines turn into whitespace.

```
#ifdef DEBUG
    printf("main: some debugging info here...\n");
#endif /* DEBUG */
```

- You can put into a .h file:

```
#define DEBUG
```

- Or, you can use the -D option:

```
gcc -DDEBUG ...
```

Conditional compilation (continued):

- The #if directive allows evaluation of constant integral expressions:

```
static void
#ifdef (__STDC__ || defined(__cplusplus))
setflags(char s[])
#else
setflags(s)
    char s[];
#endif
{
    if (s[0] == 'D')
        sch_debugflag = 1; /* general debugging */
    else if (s[0] == 'M')
        sch_msg_debugflag = 1; /* message tracing */

    if (s[1] == 'P')
        traceflag = 1; /* procedure call/reply tracing */

    if (s[2] == 'T')
        transport = s[3]; /* specific comm protocol requested */
} /* setflags */
```

ANSI C (C89) function definition.

Original K&R C function definition.

Conditional compilation (continued):

- Here is an easy way to “comment out” code; works when nested comments are present:

```
#if 0
for (...) {
    some code here /* comment here */
    more code here
}
#endif
```

- Standard .h files use #if in many places. Here is an example from /usr/include/string.h (on Fedora):

```
/*
 *      ISO C99 Standard: 7.21 String handling <string.h>
 */

#ifndef _STRING_H
#define _STRING_H    1

#include <features.h>
...
#endif /* string.h */
```

Note: the 'n' in ifndef.
Means “if not defined”.

- Known as an “Include guard”.
- Useful when .h files might be included more than once (because other .h files include them).
- Prevents multiple definitions of the same **struct** definition, for example.

Conditional compilation (continued):

- Program translation (compilation) can be terminated with #error:

```
#ifndef MAX_FLOAT
#error Whoa! No definition for MAX_FLOAT!
#endif /* MAX_FLOAT */
```

- You can “turn off” definitions using #undef:

```
int func_with_problem(...)
{
#define DEBUG 1
    function code here, including
    if ( DEBUG )
        printf...
#undef DEBUG
} /* func_with_problem */
```

Conditional compilation (continued):

- There are predefined macros. Here are some useful ones:

```
__DATE__ Current date as a string
__TIME__ Current time as a string
__FILE__ Source file name as a string
__LINE__ Line number as an int
```

- Example:

```
printf("Compiled at %s on %s\n", __TIME__, __DATE__);
```

- After preprocessing, this printf becomes:

```
printf("Compiled at %s on %s\n", "07:00:38", "Dec 1 2005");
```

Unions

- A *union* can be thought of as a **struct** in which **all members start at the same address**.
- A *union* declaration is very similar to a **struct** declaration, but uses **union** instead of **struct**.

```
union type_u {
    int i;
    long L;
    char ch;
    char bytes[sizeof(long)];
};
```

- Example:

```
lectura-> cat union1.c
#include <stdio.h>
int
main( int argc, char *argv[] )
{
    union type_u U;

    printf("&U      = %p\n", &U);
    printf("&U.i    = %p\n", &U.i);
    printf("&U.L    = %p\n", &U.L);
    printf("&U.ch   = %p\n", &U.ch);
    printf("U.bytes = %p\n", U.bytes);

    return 0;
} /* main */
```

```
lectura-> gcc -Wall -o union1 union1.c && union1
&U      = 0x7fffffff58850
&U.i    = 0x7fffffff58850
&U.L    = 0x7fffffff58850
&U.ch   = 0x7fffffff58850
U.bytes = 0x7fffffff58850
```

Unions (continued):

- The fields within a **union** all occupy the same space.

- Thus, only one value can be in the **union**.

- Side-effect: a value can be stored in a **union** as one type and accessed as another:

- On OS X on a G5 processor:

```
wolf-> cat union2.c
#include <stdio.h>
int
main( int argc, char *argv[])
{
    int i;
    union type_u U;
    U.L = 0x11223344;
    for (i = 0; i < sizeof(U.bytes); i++)
        printf("%x\n", U.bytes[i]);
    return 0;
} /* main */
wolf-> gcc -Wall -o union2 union2.c && union2
11
22
33
44
```

On lectura, on a dual-core Opteron:

```
lectura-> gcc -Wall -o union2 union2.c && union2
44
33
22
11
0
0
0
0
```

```
union type_u {
    int i;
    long L;
    char ch;
    char bytes[sizeof(long)];
};
```

Unions (continued):

- Unions commonly appear as one, or more, fields within a **struct**. A type field is useful to indicate how to interpret the union.

- Robot example. The structure defines a robot command and uses a union to hold one of three commands:

```
#define MOVE 1
#define ROTATE 2
#define BEEP 3
#define MAXBEEP 20
struct MoveCommand {
    char direction; /* L, R, B, F */
    int distance;
};
struct RotateCommand {
    float degrees;
};
struct RobotCommand {
    int type; /* MOVE, ROTATE, BEEP */
    union command {
        struct MoveCommand move;
        struct RotateCommand rotate;
        struct BeepCommand {
            char sequence[MAXBEEP];
        } beep;
    } command;
};

int
main( int argc, char *argv[])
{
    struct RobotCommand cmds[3], *cmdp = cmds;
    cmdp->type = BEEP;
    strcpy(cmdp->command.beep.sequence, "10101110");
    cmdp++;
    cmdp->type = ROTATE;
    cmdp->command.rotate.degrees = 125.3;
    cmdp++;
    cmdp->type = MOVE;
    cmdp->command.move.direction = 'R';
    cmdp->command.move.distance = 10;
    /* send the commands to the robot */
    SendCommand(cmds, 3);
    return 0;
} /* main */
```

Unions (continued):

- Unions commonly appear as one, or more, fields within a **struct**. A type field is useful to indicate how to interpret the union.

- Robot example. The structure defines a robot command and uses a union to hold one of three commands:

```
#define MOVE 1
#define ROTATE 2
#define BEEP 3
#define MAXBEEP 20
struct RobotCommand {
    int type; /* MOVE, ROTATE, BEEP */
    union command {
        struct MoveCommand {
            char direction; /* L, R, B, F */
            int distance;
        } move;
        struct RotateCommand {
            float degrees;
        } rotate;
        struct BeepCommand {
            char sequence[MAXBEEP];
        } beep;
    } command;
};

int
main( int argc, char *argv[])
{
    struct RobotCommand cmds[3], *cmdp = cmds;
    cmdp->type = BEEP;
    strcpy(cmdp->command.beep.sequence, "10101110");
    cmdp++;
    cmdp->type = ROTATE;
    cmdp->command.rotate.degrees = 125.3;
    cmdp++;
    cmdp->type = MOVE;
    cmdp->command.move.direction = 'R';
    cmdp->command.move.distance = 10;
    /* sends commands to the robot */
    SendCommand(cmds, 3);
    return 0;
} /* main */
```

Unions (continued):

- Unions are typically used for two reasons:

- Reinterpretation of a value by storing it as one type and accessing it as another.

- Especially true when you need to access a type in a piecewise fashion.

- To save space. (How big would a **RobotCommand** be if a **union** were not used?)

typedef

- The typedef specifier is used to create a new name for an existing type.
 - `typedef int integer;`
 - `integer` is now a synonym for `int`.
- Extends the language by creating a new type that can be used anywhere a type is required.
 - Examples:

```
integer i;
integer a[10];
i = (integer)2.5;
integer *ptr = &i;
```
- The general form:

```
typedef type-specification typename;
```
- The *type-specification* can be arbitrarily complex. A simple example:

```
typedef char *String;
```

 - The *type-specification* here is `char *` and the *typename* is `String`.

```
String s1 = "abc", s2 = "xyz";
String a[] = { s1, s2 };
```

typedef (continued):

- A type defined with `typedef` can be used in a `typedef`:

```
typedef String StringPair[2];
StringPair names;
names[0] = "abc";
names[1] = "xyz";
```
- Although `names` is declared as `StringPair`, the underlying type is still `char **` and `names` can be used as a `char **`. This makes the following correct:

```
char ch = **names;
```
- Using `typedef` can improve readability in some cases.
 - But if the reader does not know the underlying type, confusion results!

typedef (continued):

- Type checking is always performed on the underlying type.

```
typedef double weight;
weight f( weight w );
```
- The intent might be that the compiler will complain if `f()` is called with something besides a `weight`:

```
wolf-> cat typedef3.c
#include <stdio.h>

typedef double weight;          weight f(weight w)
weight f(weight w);           {
                                printf("f: w = %f\n", w);
                                return (w * 2.0);
                                } /* f */

int main( int argc, char *argv[] )
{
    double dd;
    weight ww = 200.0;
    dd = f(ww);
    printf("main: dd = %f\n", dd);

    dd = 300;
    ww = f(dd);
    printf("main: ww = %f\n", ww);
    return 0;
} /* main */

lectura-> gcc -Wall -o typedef3 typedef3.c && typedef3
f: w = 200.000000
main: dd = 400.000000
f: w = 300.000000
main: ww = 600.000000
```

typedef and struct:

- The typical `struct` definition defines a tag. I.e., the following defines a structure tag named `Rectangle`:

```
struct Rectangle {
    int width;
    int height;
};
```
- To define an instance of the structure, both `struct` and the tag name are used:

```
struct Rectangle rect1;
```
- A `typedef` can create a `Rectangle` type based on the structure tag:

```
typedef struct Rectangle Rectangle;
```

 - The *type-specification* is `struct Rectangle` and the *typename* is `Rectangle`.
 - The following then work:

```
Rectangle rect, *rectPtr, rects[5];
int area_Rectangle(Rectangle *ptr);
```

typedef and struct (continued):

- Some variations are possible. One is to define a type, but not a tag:

```
typedef struct {
    double x;
    double y;
} Point;

Point p1; // works
struct Point p2; // Does not work: storage size of 'p2' is not known
```

- Can specify both a tag and a type:

```
typedef struct Point {
    double x;
    double y;
} Point;

Point p1; //works
struct Point p2; // works!
```

- Self-referential structures require additional attention:

```
typedef struct {
    int value;
    Node * next; // Error: parse error before "Node"
} Node;
```

typedef and struct (continued):

- Here is one way to fix the problem:

```
typedef struct Node {
    int value;
    struct Node *next;
} Node;
```

- Here is another way to fix the problem:

```
typedef struct Node Node;
struct Node {
    int value;
    Node *next;
};
```

Creates an equivalency:

"Node" can be used in place of "struct Node"
Even though the compiler does not (yet) know
the meaning of "struct Node"

typedef and struct (continued):

- Scope:

- `typedef` scope is the same as variables: it can be local to a function, or global.

```
typedef int GLOBAL_INT;
void f() {
    typedef int LOCAL_INT; // usable only in f()
    GLOBAL_INT i;
    LOCAL_INT j;
}
void g() {
    LOCAL_INT b; // Error: LOCAL_INT has no scope in g()
}
```

typedef and struct (continued):

Example taken from the source code of the Minix Operating System:

```
/* Types relating to messages. */
#define M1 1
#define M3 3
#define M4 4
#define M3_STRING 14

typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
typedef struct {int m3i1, m3i2; char *m3p1; char m3cal[M3_STRING];} mess_3;
typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
typedef struct {char m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;} mess_5;
typedef struct {int m6i1, m6i2, m6i3; long m6l1; sighandler_t m6f1;} mess_6;

typedef struct {
    int m_source; /* who sent the message */
    int m_type; /* what kind of message is it */
    union {
        mess_1 m_m1;
        mess_2 m_m2;
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_6 m_m6;
    } m_u;
} message;
```

typedef and struct (continued):

- We can then create and fill in a message using the first message format:

```
message m;
m.m_source =
m.m_type =
m.m_u.m.m1.m1i1 =
m.m_u.m.m1.m1i2 =
m.m_u.m.m1.m1i3 =
m.m_u.m.m1.m1p1 =
m.m_u.m.m1.m1p2 =
m.m_u.m.m1.m1p3 =
```

- Ugh!

typedef and struct (continued):

```
/* The following defines provide names for useful members. */
#define m1_i1 m_u.m.m1.m1i1
#define m1_i2 m_u.m.m1.m1i2
#define m1_i3 m_u.m.m1.m1i3
#define m1_p1 m_u.m.m1.m1p1
#define m1_p2 m_u.m.m1.m1p2
#define m1_p3 m_u.m.m1.m1p3
#define m2_i1 m_u.m.m2.m2i1
#define m2_i2 m_u.m.m2.m2i2
#define m2_i3 m_u.m.m2.m2i3
#define m2_l1 m_u.m.m2.m2l1
#define m2_l2 m_u.m.m2.m2l2
#define m2_p1 m_u.m.m2.m2p1
#define m3_i1 m_u.m.m3.m3i1
#define m3_i2 m_u.m.m3.m3i2
#define m3_p1 m_u.m.m3.m3p1
#define m3_cal m_u.m.m3.m3cal
#define m4_l1 m_u.m.m4.m4l1
#define m4_l2 m_u.m.m4.m4l2
#define m4_l3 m_u.m.m4.m4l3
#define m4_l4 m_u.m.m4.m4l4
#define m4_l5 m_u.m.m4.m4l5
#define m5_c1 m_u.m.m5.m5c1
#define m5_c2 m_u.m.m5.m5c2
#define m5_i1 m_u.m.m5.m5i1
#define m5_i2 m_u.m.m5.m5i2
#define m5_l1 m_u.m.m5.m5l1
#define m5_l2 m_u.m.m5.m5l2
#define m5_l3 m_u.m.m5.m5l3
#define m6_i1 m_u.m.m6.m6i1
#define m6_i2 m_u.m.m6.m6i2
#define m6_i3 m_u.m.m6.m6i3
#define m6_l1 m_u.m.m6.m6l1
#define m6_f1 m_u.m.m6.m6f1
```

typedef and struct (continued):

- We can then create and fill in a message using the first message format:

```
message m;
m.m_source =
m.m_type =
m.m_u.m.m1.m1i1 = becomes m.m1_i1 =
m.m_u.m.m1.m1i2 = becomes m.m1_i2 =
m.m_u.m.m1.m1i3 = becomes m.m1_i3 =
m.m_u.m.m1.m1p1 = becomes m.m1_p1 =
m.m_u.m.m1.m1p2 = becomes m.m1_p2 =
m.m_u.m.m1.m1p3 = becomes m.m1_p3 =
```

- The macros are used throughout the message-handling code in Minix to avoid errors from typing (and confusion on the part of the code writer!).

Bitwise Operators

- Bitwise operators perform calculations on integer values.
 - The integer values are treated as a string of binary digits.
- The ~ (tilde) operator performs a bit-by-bit complement of its operand's value, changing 0 to 1 and 1 to 0.

```
short a = 0x0F61;    0000 1111 0110 0001
short b = ~a;      1111 0000 1001 1110
```
- The & operator performs a bit-by-bit and operation on its two operands. It produces 1 if both input bits are 1; otherwise 0.

```
short a = 0xF0A1;    1111 0000 1010 0001
short b = 0x00F3;    0000 0000 1111 0011
short c = a & b;     0000 0000 1010 0001
```
- Note that **short** values are used in these examples; but the bitwise operators work on all integer types (1 byte, 2 bytes, 4 bytes, 8 bytes, ...).

Bitwise Operators (continued):

- The `|` operator performs a bit-by-bit or operation on its two operands. It produces a `1` if either input bit is `1`; otherwise `0`.

```
short a = 0xF0A1;    1111 0000 1010 0001
short b = 0x00F3;    0000 0000 1111 0011
short c = a | b;     1111 0000 1111 0011
```

- The `^` operator performs a bit-by-bit exclusive or operation on its two operands. It produces a `1` if the two inputs differ; otherwise `0`.

```
short a = 0xF0A1;    1111 0000 1010 0001
short b = 0x00F3;    0000 0000 1111 0011
short c = a ^ b;     1111 0000 0101 0010
```

Bitwise Operators (continued):

- The binary shift operators: `<<` left shift and `>>` right shift.

`a << 1`

- An expression that produces the bits of `a` shifted to the left by one bit position,
 - With a `0` inserted as the rightmost bit.
 - The leftmost bit is discarded.
 - The right-hand operand (the `1` above) can be any integer value.

```
short a = 1;        0000 0000 0000 0001
a = a << 1;         0000 0000 0000 0010
a = a << 4;         0000 0000 0010 0000
a <<= 5;            0000 0100 0000 0000
a = 0xFF << 8;     1111 1111 0000 0000
```

Bitwise Operators (continued):

- Right shift has a (potential) pitfall:
 - If the value is unsigned, `0`'s are shifted in from the left. (No problem here)
 - If the value is signed, `0`'s may be shifted in or the sign bit may be propagated.
 - The behavior is compiler-dependent. (Problem here).
- Not a problem: unsigned example:

```
unsigned short a = 0x8000;    1000 0000 0000 0000
a = a >> 1;                   0100 0000 0000 0000
a >>= 5;                       0000 0010 0000 0000
```

- Problem: signed example:

```
short a = -32768;           1000 0000 0000 0000
a = a >> 1;                   1100 0000 0000 0000
a >>= 7;                       1111 1111 1000 0000
```

- Solaris 9, Fedora 4 on `lectura`, OS X 10.4.11 (Tiger) and 10.5.1 (Leopard): shift right filling with the sign bit.
- Java has both `>>` (signed) and `>>>` (unsigned) binary operators. C does not have `>>>`.

Bitwise Operators (continued):

- Here is a test to determine which bit is shifted in on the left:

```
lectura-> cat bitwisel.c
#include <stdio.h>

int
main( int argc, char *argv[] )
{
    unsigned short u = 0x8000;
    short s = 0x8000;

    u = u >> 3; printf("u >> 3 = %x\n", u);

    s = s >> 3; printf("s >> 3 = %x\n", s);

    return 0;
} /* main */
lectura-> gcc -Wall -o bitwisel bitwisel.c && bitwisel
u >> 3 = 1000
s >> 3 = fffff000
```

Bitwise Operators (continued):

- Some examples:

```
i = i >> 1; // divide by 2, faster than using: i = i / 2;  
j = j << 2; // multiply by 4, faster than using: j = j * 4;  
a = a ^ a; // what is the value of a after this line is executed?  
a ^= a;
```

- Determine the number of bits in an int:

```
lectura-> cat bitwise2.c  
#include <stdio.h>  
int  
main( int argc, char *argv[] )  
{  
    unsigned int i = -0; // what does this do?  
    int nbits = 0;  
  
    while (i) {  
        nbits++;  
        i >>= 1;  
    }  
    printf("%d bits\n", nbits);  
  
    return 0;  
} /* main */  
lectura-> gcc -Wall -o bitwise2 bitwise2.c && bitwise2  
32 bits
```

Bitwise Operators (continued):

- More examples:

```
lectura-> cat bitwise3.c  
#include <stdio.h>  
  
int first_bit_set(int w);  
  
int  
main( int argc, char *argv[] )  
{  
    int i = 25, j = 16384;  
    printf("i = %d, i = %x, first bit = %d\n", i, i, first_bit_set(i));  
    printf("j = %d, j = %x, first bit = %d\n", j, j, first_bit_set(j));  
    return 0;  
} /* main */  
  
int  
first_bit_set(int w)  
{  
    int i;  
    for (i = sizeof(i)*8 - 1; i >= 0; i--) // Assumes 8-bit chars  
        if (w & 1<<i)  
            return i;  
    return -1;  
} /* first_bit_set */
```

```
lectura-> gcc -Wall -o bitwise3 bitwise3.c && bitwise3  
i = 25, i = 19, first bit = 4  
j = 16384, j = 4000, first bit = 14
```

Bitwise Operators (continued):

- Convert an int to a character string of 0's and 1's:

```
lectura-> cat bitwise4.c  
#include <stdio.h>  
void val_to_bits(int val, int size, char *buf)  
{  
    int nbits = size * 8, i; // Assumes 8-bit chars  
    char bit;  
  
    for (i = nbits - 1; i >= 0; i--) {  
        bit = (val & 1<<i) ? '1' : '0';  
        *buf++ = bit;  
        if (i % 4 == 0 && i != 0)  
            *buf++ = ' ';  
    }  
    *buf = 0;  
} /* val_to_bits */  
lectura-> gcc -Wall -o bitwise4 bitwise4.c  
lectura-> bitwise4  
CAFEBABE is 1100 1010 1111 1110 1011 1010 1011 1110  
X is 0101 1000  
  
int main( int argc, char *argv[] )  
{  
    int i = 0xCafeBabe; // first four bytes of a .class file  
    char buf[sizeof(i)*8 + 1]; // Assumes 8-bit chars  
    val_to_bits(i, sizeof(i), buf);  
    printf("%X is %s\n", i, buf);  
    val_to_bits('X', sizeof(char), buf);  
    printf("%c is %s\n", 'X', buf);  
    return 0;  
} /* main */
```