Write a chess game using bit-fields and masks

Using bit-fields and masks is a common method to combine data without using structures.

By Jim Hall (Correspondent)

August 23, 2021 | 2 Comments | 5 min read

75 readers like this.

Image by: Let's say you were writing a chess game in C. One way to track the pieces on the board is by defining a structure that defines each possible piece on the board, and its color, so every square contains an element from that structure. For example, you might have a structure that looks like this:

```
struct chess_pc {
   int piece;
   int is_black;
}
```

With this programming structure, your program will know what piece is in every square and its color. You can quickly identify if the piece is a pawn, rook, knight, bishop, queen, or king—and if the piece is black or white. But there's a more straightforward way to track the same information while using less data and memory. Rather than storing a structure of two `int` values for every square on a chessboard, we can store a single `int` value and use binary *bit-fields* and *masks* to identify the pieces and color in each square.

Bits and binary

When using bit-fields to represent data, it helps to think like a computer. Let's start by listing the possible chess pieces and assigning a number to each. I'll help us along to the next step by representing the number in its binary form, the way the computer would track it. Remember that binary numbers are made up of *bits*, which are either zero or one.

- **`00000000:`** empty (0)
- **`00000001:`** pawn (1)
- **`00000010:`** rook (2)
- **`00000011:`** knight (3)
- **`00000100:`** bishop (4)
- **`00000101:`** queen (5)
- **`00000110:`** king (6)

To list all pieces on a chessboard, we only need the three bits that represent (from right to left) the values 1, 2, and 4. For example, the number 6 is binary `110`. All of the other bits in the binary representation of 6 are zeroes.

And with a bit of cleverness, we can use one of those extra always-zero bits to track if a piece is black or white. We can use the number 8 (binary `00001000`) to indicate if a piece is black. If this bit is 1, it's black; if it's 0, it's white. That's called a *bit-field*, which we can pull out later using a binary *mask*.

Storing data with bit-fields

To write a chess program using bit-fields and masks, we might start with these definitions:

```
/* game pieces */

#define EMPTY 0
#define PAWN 1
#define ROOK 2
#define KNIGHT 3
#define BISHOP 4
#define QUEEN 5
#define KING 6

/* piece color (bit-field) */

#define BLACK 8
#define WHITE 0

/* piece only (mask) */

#define PIECE 7
```

When you assign a value to a square, such as when initializing the chessboard, you can assign a single `int` value to track both the piece and its color. For example, to store a black rook in position 0,0 of an array, you would use this code:

```
  int board[8][8];

..

  board[0][0] = BLACK | ROOK;
```

The `|` is a binary OR, which means the computer will combine the bits from two numbers. For every bit position, if that bit from *either* number is 1, the result for that bit position is also 1. Binary OR of the value `BLACK` (8, or binary `00001000`) and the value `ROOK` (2, or binary `00000010`) is binary `00001010`, or 10:

```
    00001000 = 8
 OR 00000010 = 2
    ________
    00001010 = 10
```

Similarly, to store a white pawn in position 6,0 of the array, you could use this:

```
  board[6][0] = WHITE | PAWN;
```

This stores the value 1 because the binary OR of `WHITE` (0) and `PAWN` (1) is just 1:

```
       00000000 = 0
    OR 00000001 = 1
       ________
       00000001 = 1
```

Getting data out with masks

During the chess game, the program will need to know what piece is in a square and its color. We can separate the piece using a binary mask.

For example, the program might need to know the contents of a specific square on the board during the chess game, such as the array element at `board[5][3]`. What piece is there, and is it black or white? To identify the chess piece, combine the element's value with the `PIECE` mask using the binary AND:

```
  int board[8][8];
  int piece;

..

  piece = board[5][3] & PIECE;
```

The binary AND operator (`&`) combines two binary values so that for any bit position, if that bit in *both* numbers is 1, then the result is also 1. For example, if the value of `board[5][3]` is 11 (binary `00001011`), then the binary AND of 11 and the mask PIECE (7, or binary `00000111`) is binary `00000011`, or 3. This is a knight, which also has the value 3.

```
    00001011 = 11
AND 00000111 = 7
    ________
    00000011 = 3
```

Separating the piece's color is a simple matter of using binary AND with the value and the `BLACK` bit-field. For example, you might write this as a function called `is_black` to determine if a piece is either black or white:

```
int
is_black(int piece)
{
  return (piece & BLACK);
}
```

This works because the value `BLACK` is 8, or binary `00001000`. And in the C programming language, any non-zero value is treated as True, and zero is always False. So `is_black(board[5][3])` will return a True value (8) if the piece in array element `5,3` is black and will return a False value (0) if it is white.

Bit fields

Using bit-fields and masks is a common method to combine data without using structures. They are worth adding to your programmer's "tool kit." While data structures are a valuable tool for ordered programming where you need to track related data, using separate elements to track single On or Off values (such as the colors of chess pieces) is less efficient. In these cases, consider using bit-fields and masks to combine your data more efficiently.