# Bit manipulation

Raul P. Pelaez

October 21, 2025

## Contents



# 1 Introduction

Today, we will explore how to manipulate individual bits within integers using bitwise operations. An integer consists (typically) of 32 bits that we can interpret as boolean values (0 or 1). Bitwise operations allow us to perform logical operations on these bits directly, enabling efficient data manipulation and storage. We will see some examples of how this knowledge is exploited in the wild.

# 2 Bitwise operations

Bitwise operations are operations that directly manipulate bits. The most common bitwise operations are AND, OR, XOR, NOT, and bit shifts. Let us see some of them.

The following operators receive two integer operands and perform the operation bit by bit:

- `AND` (`&`)
- `OR` (`|`)
- `XOR` (`^`)
- Left shift (`<<`): shifts all bits to the left, filling with zeros on the right.
- Right shift (`>>`) : shifts all bits to the right. The behavior on the left side depends on the type (logical vs arithmetic shift).

The following operator receives one integer operand and inverts all bits:

- NOT (`~`): inverts all bits (0s become 1s and vice versa).

We can also combine these operators with the assignment operator (`=`) to create compound assignment operators, like `&=`, `|=`, `^=`, `<<=`, and `>>=`.

When given two operands, this operators will perform the operation bit by bit. For instance, say we use two characters, which occupy 1 byte (8 bits) each. The operation `0b00000101 & 0b00000110` will yield `0b00000100`, because only the third bit is 1 in both operands. We can write this operation in C as:

```c
#include <stdio.h>
int main() {
    char a = 0b00000101; // 5 in decimal
    char b = 0b00000110; // 6 in decimal
    char c = a & b;      // Bitwise AND
    printf("Result of a & b: %d\n", c); // Outputs: 4
    return 0;
}
```

**Output**

```
Result of a & b: 4
```

**Info**

**Literals in non-decimal base**

In C, you can represent integer literals in different bases:

- Binary: Prefix with `0b` or `0B` (e.g., `0b1010` for 10 in decimal).
- Octal: Prefix with `0` (e.g., `012` for 10 in decimal). Quite confusing, right?
- Hexadecimal: Prefix with `0x` or `0X` (e.g., `0xA` for 10 in decimal).

**Examples**

```c
#include <stdio.h>
 int main() {
     unsigned int a = 5;  // Binary: 0000...0101
     unsigned int b = 3;  // Binary: 0000...0011

     printf("a & b = %u\n", a & b);   // AND: 0000...0001 (1)
     printf("a | b = %u\n", a | b);   // OR:  0000...0111 (7)
     printf("a ^ b = %u\n", a ^ b);   // XOR: 0000...0110 (6)
     printf("~a = %u\n", ~a);         // NOT: 1111...1010 (depends on int size)
     printf("a << 1 = %u\n", a << 1); // Left shift: 0000...1010 (10)
     printf("a >> 1 = %u\n", a >> 1); // Right shift: 0000...0010 (2)

     return 0;
 }
```

**Output**

```
a & b = 1
a | b = 7
a ^ b = 6
~a = 4294967290
a << 1 = 10
a >> 1 = 2
```

Advice

**Left shift multiplies by 2**

If we left shift the number `0001` (1 in decimal) by 3 positions, we get `1000` (8 in decimal). This is equivalent to multiplying the original number by `2^3` (or 8). Check it out:

```c
#include <stdio.h>
int multiply_by_power_of_two(int num, int power) {
  return num << power; // Left shift by 'power' positions
}

int main() {
  printf("%d multiplied by 2^%d is %d\n", 1, 3, multiply_by_power_of_two(1, 3));
  printf("%d multiplied by 2^%d is %d\n", 5, 2, multiply_by_power_of_two(5, 2));
  return 0;
}
```

Output

```
1 multiplied by 2^3 is 8
5 multiplied by 2^2 is 20
```

Be careful with overflowing (the result being larger than what can be represented with the number of bits available) if you use this trick.

## 2.1 Some tricks of the trade

Info

**Setting and clearing bits**

Sometimes the need arises to set (turn to 1) or clear (turn to 0) specific bits in an integer. We can achieve this using bitwise operations. To set a specific bit, we can use the bitwise OR operation with a mask that has the target bit set to 1 and all other bits set to 0. For example, to set the 3rd bit (counting from 0) of an integer `num`, we can do:

```c
num |= (1 << 3); // Set the 3rd bit
```

To clear a specific bit, we can use the bitwise AND operation with a mask that has the target bit set to 0 and all other bits set to 1. For example, to clear the 3rd bit of an integer `num`, we can do:

```c
num &= ~(1 << 3); // Clear the 3rd bit
```

The key in these lies in the construct `(1 << i)`, which represents $2^i$, written in binary as all zeros, except for the i-th bit, which is a 1. The NOT operator (`~`) inverts all bits, so we get all ones except for the i-th bit, which becomes a 0. This way, when we AND with this mask, we clear the i-th bit while leaving all other bits unchanged.

Info

**Checking if a bit is set**

To check if a specific bit is set (1) in an integer, we can use the bitwise AND operation with a mask that has the target bit set to 1 and all other bits set to 0. If the result is non-zero, the bit is set; otherwise, it is not. For example, to check if the 3rd bit of an integer `num` is set, we can do:

```c
if (num & (1 << 3)) {
    // The 3rd bit is set
} else {
    // The 3rd bit is not set
}
```

Info

**Counting set bits**

To count the number of bits set to 1 in an integer, we can use a loop that repeatedly checks the least significant bit and then right shifts the integer until it becomes zero. Here is a simple implementation:

```c
#include <stdio.h>
int count_set_bits(unsigned int num) {
    int count = 0;
    while (num) {
        count += num & 1; // Increment count if the least significant bit is 1
        num >>= 1;        // Right shift to check the next bit
    }
    return count;
}
int main() {
    unsigned int number = 29; // Binary: 11101
    printf("Number of set bits in %u is %d\n", number, count_set_bits(number));
    return 0;
}
```

**Output**

```
Number of set bits in 29 is 4
```

Info

**Toggling bits**

To toggle (invert) a specific bit in an integer, we can use the bitwise XOR operation with a mask that has the target bit set to 1 and all other bits set to 0. For example, to toggle the 3rd bit of an integer `num`, we can do:

```c
num ^= (1 << 3); // Toggle the 3rd bit
```

If the 3rd bit was 0, it becomes 1; if it was 1, it becomes 0.

Info

**MSB and LSB**

The Most Significant Bit (MSB) is the bit in a binary number that has the highest value position. In a standard 32-bit integer, the MSB is the 31st bit (counting from 0). The MSB is often used to indicate the sign of the number in signed integer representations (0 for positive, 1 for negative). The Least Significant Bit (LSB) is the bit in a binary number that has the lowest value position. In a standard 32-bit integer, the LSB is the 0th bit.

Info

**Endianness**

Endianness refers to the order in which bytes are arranged within larger data types (like integers) when stored in memory. There are two main types of endianness:

- Big-endian: The most significant byte (the "big end") is stored at the lowest memory address.
- Little-endian: The least significant byte (the "little end") is stored at the lowest memory address.

Most modern computers use little-endian format (like x86 architecture). Endianness is important to consider when performing low-level data manipulation, especially when dealing with binary file formats or network protocols, as it can affect how data is interpreted.

**Example**

Here is a simple example to illustrate endianness:

```c
#include <stdio.h>
void print_bytes(unsigned int num) {
    unsigned char *byte_ptr = (unsigned char *)&num;
    for (int i = 0; i < sizeof(num); i++) {
        printf("Byte %d: 0x%02x\n", i, byte_ptr[i]);
    }
}
int main() {
    unsigned int number = 0x12345678;
    printf("Number: 0x%x\n", number);
    print_bytes(number);
    return 0;
}
```

**Output**

```
Number: 0x12345678
Byte 0: 0x78
Byte 1: 0x56
Byte 2: 0x34
Byte 3: 0x12
```

Info

**Order of bits in a byte**

When we talk about the order of bits within a byte, we typically refer to the bit numbering convention. The most common convention is to number bits from right to left, starting with 0 for the least significant bit (LSB) and ending with 7 for the most significant bit (MSB). For example, the number `0b00000101` (5 in decimal) has its bits numbered as follows:

```c
#include <stdio.h>
void print_bits(unsigned char byte) {
    for (int i = 7; i >= 0; i--) {
        printf("%d", (byte >> i) & 1);
    }
    printf("\n");
}
int main() {
    unsigned char byte = 0b00000101; // 5 in decimal
    printf("Byte: ");
    print_bits(byte);
    return 0;
}
```

**Output**

```
Byte: 00000101
```

Look at this code carefully, note that the loop goes from 7 to 0, so we print the most significant bit (the leftmost) first.

## 3 Encoding negative numbers

The computer can only represent numbers as a sequence of bits (0s and 1s). Representing positive integers is straightforward using binary notation. Lets exemplify how with 8 bit unsigned integers:

```c
#include <stdio.h>
#include <stdint.h>
#include <math.h>
int main()
{

    uint8_t max_unsigned = 255; // 0b11111111
    printf("Max unsigned 8-bit integer: %u\n", max_unsigned);
    // Print in binary
    printf("Binary representation: 0b");
    for (int i = 7; i >= 0; i--)
    {
        printf("%d", (max_unsigned & (1 << i)) ? 1 : 0);
    }
    printf("\n");
    // Reconstruct from bits
    uint8_t reconstructed = 0;
    for (int i = 0; i < 8; i++)
    {
        reconstructed |= ((max_unsigned >> i) & 1) << i;
        // Equivalent to reconstructed += bit * pow(2, i);
    }
    printf("Reconstructed value: %u\n", reconstructed);
    return 0;
}
```

**Output**

```
Max unsigned 8-bit integer: 255
Binary representation: 0b11111111
Reconstructed value: 255
```

In other words, if number $x \in [0, 2^n - 1]$ is represented in binary as bits $x_0, x_1, \ldots, x_{n-1}$ (where $x_0$ is the LSB), then the decimal value of $x$ is given by:

$$x = \sum_{i=0}^{n} x_i 2^i$$

## 3.1 Two's complement

Representing negative integers is more complicated. The most common method is to use two's complement representation. In this scheme, the MSB is used as the sign bit (0 for positive, 1 for negative). To represent a negative number, we take its absolute value, invert all bits, and add 1. For example, to represent -5 in an 8-bit two's complement system:

1. Start with the binary representation of 5: `00000101`
2. Invert all bits: `11111010`
3. Add 1: `11111010 + 00000001 = 11111011`

Thus, -5 is represented as `11111011` in an 8-bit two's complement system. To convert back to decimal, we can check the MSB. If it is 1, we know the number is negative. We can then invert the bits and add 1 to get the absolute value, and finally negate it.

```c
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
void print_bits(int8_t byte) {
  for (int i = 7; i >= 0; i--) {
    printf("%d", (byte >> i) & 1);
  }
  printf("\n");
}
int main() {
  int8_t five = 5; // 0b00000101
  int8_t neg_five = -5; // 0b11111011 in two's complement
  assert(neg_five == (int8_t)(~five + 1)); // Verify two's complement representation
  printf("Binary representation of 5: ");
  print_bits(five);
  printf("Binary representation of -5: ");
  print_bits(neg_five);
  return 0;
}
```

**Output**

```
Binary representation of 5: 00000101
Binary representation of -5: 11111011
```

Note the passing assertion, in which we verify that the two's complement representation of -5 is indeed equal to the bitwise NOT of 5 plus 1.

Insight

**The name "two's complement"**
The name has a mathematical origin in something called the Method of complements.

Advanced

**Other representations of negative numbers**
While two's complement is the most widely used method for representing negative integers in modern computing, there are other methods that have been used historically or in specific applications. Some of these include:

- One's Complement: In this method, negative numbers are represented by inverting all bits of their positive counterparts. For example, -5 would be represented as `11111010` in an 8-bit system. However, this method has two representations for zero (positive and negative zero), which can complicate arithmetic operations.
- Sign-Magnitude: In this representation, the MSB is used as a sign bit (0 for positive, 1 for negative), and the remaining bits represent the magnitude of the number. For example, -5 would be represented as `10000101` in an 8-bit system. This method also has two representations for zero and can complicate arithmetic operations.

# 4 Enumerations as flags

We can use enumerations to define a set of named integer constants that can be used as flags. Each flag can be represented by a single bit in an integer, allowing us to combine multiple flags using bitwise operations. This is a common technique in C programming for managing options and settings. In essence, we can "pack" and "unpack" multiple options into a single variable using bitwise operations if we define a special enum. Let me give you an example:

```c
#include <stdio.h>
typedef enum {
    READ    = 1<<0,  // 1
    WRITE   = 1<<1,  // 2
    EXECUTE = 1<<2,  // 4
} Permission;

void check_permissions(int permission_flags) {
    if (permission_flags & READ) {
        printf("Read permission granted.\n");
    }
    if (permission_flags & WRITE) {
        printf("Write permission granted.\n");
    }
    if (permission_flags & EXECUTE) {
        printf("Execute permission granted.\n");
    }
}
int main() {
  int permissions = READ | WRITE; // Combine READ and WRITE permissions
  check_permissions(permissions);
  return 0;
}
```

**Output**

```
Read permission granted.
Write permission granted.
```

The trick requires to define each enum value as a power of two (1, 2, 4, 8, 16, ...), so that each value corresponds to a unique bit in the binary representation of the integer. This way, we can combine multiple flags using the bitwise OR operator (`|`) and check for specific flags using the bitwise AND operator (`&`). In the example above, the bit representation of the `permissions` variable is `011` (binary), which we created as a combination of the `READ` (`001`) and `WRITE` (`010`) flags with the bitwise OR operator. When we check for the `READ` permission using the bitwise AND operator, we get `001` (true), and when we check

for the `EXECUTE` permission, we get `000` (false).

A function (like `check_permissions` in the example) can then take a single integer parameter that encodes a plethora of possible combination of options. The alternative would be to design `check_permissions` like this:

```
void check_permissions(bool can_read, bool can_write, bool can_execute);
```

You can see how this approach is less scalable and much more error-prone. For instance, it would be really easy to mess up the order of the parameters when calling the function.

As another example, the library SDL2 uses this technique extensively. For instance, when configuring a new window.

```
#include <SDL2/SDL.h>
int main() {
  if (SDL_Init(SDL_INIT_VIDEO) != 0) {
    printf("SDL_Init Error: %s\n", SDL_GetError());
    return 1;
  }
  int options = SDL_WINDOW_SHOWN | SDL_WINDOW_RESIZABLE;
  SDL_Window *win = SDL_CreateWindow("Hello World!", 100, 100, 640, 480, options);
  if (win == NULL) {
    printf("SDL_CreateWindow Error: %s\n", SDL_GetError());
    SDL_Quit();
    return 1;
  }
  SDL_Delay(3000); // Wait for 3 seconds
  SDL_DestroyWindow(win);
  SDL_Quit();
  return 0;
}
```

SDL is a library for multimedia and game development. Check the definition of the `SDL_WINDOW_FLAGS` enum here, note how each flag is a power of two, so they can be combined using bitwise OR.

Note how, in principle, if enums are defined with 32 bit integers, we can define up to 32 different flags. In practice, it is better to limit the number of flags to avoid mistakes.

# 5 Applications in the wild

Bit manipulation is widely used in various applications, including graphics programming, cryptography, and data compression. Lets go over a couple examples.

Warning

**This section is just for entertaining**
The examples in this section are real-life applications of the techniques in this lesson, but they are very advanced, mind bending and dare I say, humbling. Feel free to skip this section if you are not feeling it.

## 5.1 Carmack's fast inverse square root

One of the most famous examples of bit manipulation in graphics programming is Carmack's fast inverse square root algorithm. This algorithm was used in the Quake III Arena game engine to quickly compute the inverse square root of a floating-point number, which is a common operation in 3D graphics. The algorithm uses bit manipulation to approximate the result, significantly speeding up the computation compared to traditional methods. When Quake III was written (1999), floating point operations such as `1.0/sqrt(x)` were much more expensive than integer operations, so this trick was a big deal.

Here is the original implementation in C (taken from Wikipedia, which in turn comes from the original Quake III source code, comments included):

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  i  = * ( long * ) &y;                       // evil floating point bit level hacking
  i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
  //y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

  return y;
}
```

Is it not awesome that you have now all the tools to understand this code? I mean, *what* it is doing is obscure, but you can extract the *how*. You can read all the details about how this code works in the Wikipedia page, but lets cover the fundamentals.

**Insight**

John Carmack was the technical developer of the famous DOOM game, which was truly revolutionary at the time. He wrote DOOM, as well as subsequent games, in pure C. Some of his games (DOOM I, II and III, Quake,...) are open source and available in GitHub. I recommend you to take a look at those codebases. If you are interested in the history of the DOOM game and in Carmack in general, I cannot recommend enough the book "Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture". It is a truly insane read.

We can describe all lines of the code above except for the last one as a clever and cheap way to produce a first initial guess of the solution to use in Newton's method. The last line is just one iteration of Newton's method to refine the result. The second iteration (commented out) can be used to improve accuracy, but apparently Carmack deemed it unnecessary for the purpose of the game.

The code above is approximating the function $\frac{1}{\sqrt{x}}$ via the following algorithm:

1. Approximate $\log_2 x$
2. Use the fact that $\log_2\left(\frac{1}{\sqrt{x}}\right) = -\frac{1}{2}\log_2(x)$
3. Approximate the 2-base exponential to get $\exp\left(-\frac{1}{2}\log_2 x\right) = \frac{1}{\sqrt{x}}$
4. Refine the result using one iteration of Newton's method

Now, it is fair to look at these steps and be absolutely clueless about the relation between them and the code. Lets go step by step.

**Step 1: Approximating $\log_2 x$**

```
float y  = number;
long i  = * ( long * ) &y;
```

Reinterpreting the bits of a float (y) as a long integer (32 bits in both cases) gives us a *very* rough approximation of $\log_2 x$. This is because of the way floating point numbers are represented in memory (IEEE 754 format). The exponent part of the float gives us information about the order of magnitude of the number, which is related to its logarithm.

**Step 2: Using the property of logarithms**

Via a rough linear approximation of the logarithm, and using $\log_2(y) = \frac{1}{2}\log_2(x)$ (where $y = \frac{1}{\sqrt{x}}$), we can approximate the logarithm of the inverse square root of x. This is done in the following line:

```
i  = 0x5f3759df - ( i >> 1 );
```

The second term is just dividing the approximation of $\log_2 x$ by 2 (which is equivalent to taking the square root in the logarithmic domain). The first term is a "magic number" that is used to essentially produce a rough linear approximation of the exponential. The exact value of this magic number was found empirically by trial and error, although it is pretty close to the theoretical optimal value. The subtraction is implementing the negation in the formula. You can read the mathematical derivation in the Wikipedia page. The key is still to approximate floating point operations with integer operations.

**Step 3: A single Newton iteration**

```
y  = * ( float * ) &i;
y  = y * ( threehalfs - ( (0.5F*number) * y * y ) );
```

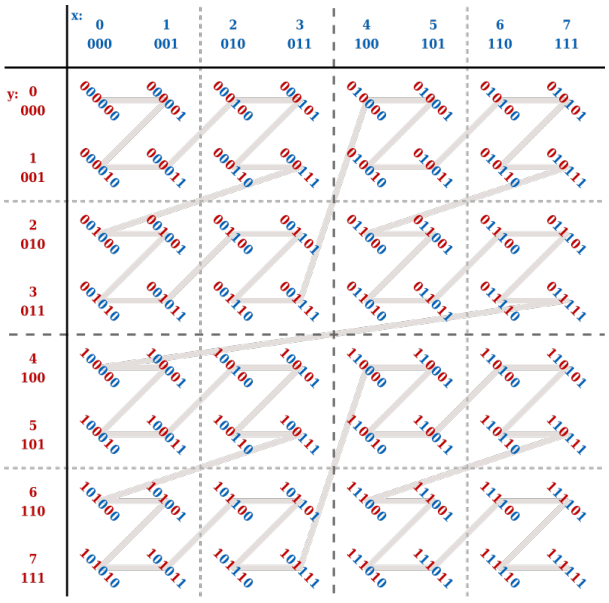
Here we go back to interpreting the bits as a float, which yields a rough approximation of the result. The next step is a refinement using one iteration of Newton's method to solve the equation $f(y) = \frac{1}{y^2} - x = 0$. This step improves the accuracy of the result significantly. The rest of the algorithm thus far is just a clever way to get a good initial guess for Newton's method using bit manipulation.

## 5.2 Morton codes for spatial hashing

Morton codes (also known as Z-order curve) are a method for mapping multidimensional data to one dimension while preserving spatial locality. This is particularly useful in computer graphics and spatial indexing. I have used them in the past for improving the access pattern of a neighbor list in a particle system, which improved performance significantly due to better cache coherence.

The idea is to interleave the bits of the coordinates of a point in space to create a single integer that represents the point. If we then sort points by their Morton codes, points that are close in space will also be close in the sorted list. In many situations this (a correlation between distances in memory and space) is a desirable property.

This image from Wikipedia illustrates the concept in 2D (interleaving the bits of the x and y coordinates):




Here is a simple implementation of a function that computes the Morton code for 2D coordinates:

```
#include <stdio.h>

unsigned int encodeMorton(unsigned int x){
  x &= 0x0000ffff;
  x = (x ^ (x <<  8)) & 0x00ff00ff;
  x = (x ^ (x <<  4)) & 0x0f0f0f0f;
  x = (x ^ (x <<  2)) & 0x33333333;
  x = (x ^ (x <<  1)) & 0x55555555;
```

```c
  return x;
}

unsigned int morton2D(unsigned int x, unsigned int y) {
  return encodeMorton(x) | (encodeMorton(y) << 1);
}

int main() {
  unsigned int x = 3; // Binary: 011
  unsigned int y = 5; // Binary: 101
  unsigned int morton_code = morton2D(x, y); // Binary: 100111 (39)
  printf("Morton code for (%u, %u) is %u\n", x, y, morton_code);
  return 0;
}
```

**Output**

```
Morton code for (3, 5) is 39
```

The `encodeMorton` function takes an integer and interleaves its bits with zeros. So the number with bits $a_0a_1a_2a_3$ becomes $0a_00a_10a_20a_3$. The `morton2D` function then combines the interleaved bits of the x and y coordinates to produce the final Morton code. In this case, the bits of x are placed in the even positions and the bits of y in the odd positions.

# 6 Exercises

## 6.1 Printing bits

**Goal**

Write a function that prints the binary representation of an 32-bit integer without using libraries beyond `stdio.h`.

```c
#include <stdio.h>

void print_bits32(int x);

int main(void){
  print_bits32(0xDEADBEEF);
  return 0;
}
```

**Advanced milestone**

Generalize it to support any type.

**hint**

Make the function take a `void*` and a size in bytes as parameters.

**Advanced milestone**

Solve the problem in two steps. First write a function that transforms a input to a sequence of bytes (characters). Then, write a function that turns a byte into a string of '0' and '1' characters. Finally, combine both functions to solve the problem.

**hint**

Use `union` to reinterpret the input as a sequence of bytes.

## 6.2 Negative integers

**Goal**

Use the function from the previous exercise to print the binary representation of negative integers. Explain the output.

```c
#include <stdio.h>

void print_bits32(int x);

int main(void){
  print_bits32(1);
  print_bits32(-1);
  return 0;
}
```

## 6.3 An interesting puzzle

Advanced

**This is a hard one**
This exercise falls outside the contents of this lesson, but does not require anything we have not covered thus far in the course. Being able to solve it, though, will mean that you have a rather profound understanding of the C language.

Goal

I found this snippet in the source code of the Quake III Arena videogame, you can access the original here. The `entityState_t` struct is defined here, its just a collection of values.

Insight

Although not relevant to the exercise, this code is part of the delta compression system in Quake III. This subsystem tried to overcome the network bandwidth limitations at the time, by sending to the players only the difference between the current game state and the previously sent one (thus the delta word).

```
typedef struct {
        char        *name;
        int                 offset;
        int                 bits;                // 0 = float
} netField_t;

// using the stringizing operator to save typing...
#define         NETF(x) #x,(int)&((entityState_t*)0)->x

netField_t        entityStateFields[] =
{
{ NETF(pos.trTime), 32 },
{ NETF(pos.trBase[0]), 0 },
// ...the code follows with similar lines for a while...
```

---

Your goal is to explain what the `NETF` macro is doing. Reproduce the technique and explain in comments how it works. I will get you started.

Milestone

Write a small reproducible example that does the same thing, for instance:

```
typedef struct {
  int a;
  double b;
  char c;
} repro_t;
typedef struct {
  char *name;
  int offset;
} netField_t;

# define NETF(x) #x,(int)&(((repro_t*)0)->x)

netField_t reproFields[] =
{
    { NETF(a) },
    { NETF(b) },
    { NETF(c) },
};
int main() {
    for (int i = 0; i < 3; i++) {
        printf("Field: %s, Offset: %d\n",
                reproFields[i].name, reproFields[i].offset);
    }
    return 0;
}
```

Explain the output of the code. Can you guess what information is being stored in `offset`? Now explain, step by step, how the macro works. Start with the innermost part: `((repro_t*)0)`, which is reinterpreting the number 0 (the null pointer) as a pointer to a `repro_t` struct.

hint

- The operator `#` inside a macro is called the stringizing operator. It turns the argument into a string literal. So `#x` becomes a string with the name of the field, e.g. `NETF(pos)` would turn `#x` into `"pos"`.