C for C++ Programmers — Exercise Answers

March 04, 2026

1. Introduction

1. Think about it: C uses format specifiers in `printf` while C++ uses `operator<<` or `std::format`. What advantage does the format string approach give you when writing output to a log file? What is a disadvantage?

Answer: The format string approach lets you define the entire output layout in a single, readable string. This is especially useful for log files where you want a consistent format — you can see the pattern at a glance:

```
fprintf(log, "[%04d-%02d-%02d %02d:%02d:%02d] %s: %s\n",
        year, month, day, hour, min, sec, level, msg);
```

The format string can also be stored in a variable or configuration, making it easy to change the log format without restructuring the code.

The disadvantage is that `printf` is not type-safe. If you write `%d` but pass a `double`, the compiler may warn you, but the behavior is undefined. C++ `operator<<` and `std::format` catch type mismatches at compile time.

---

2. What does this print?

```
printf("%05d %x\n", 42, 255);
```

Answer:

```
00042 ff
```

`%05d` prints `42` in a 5-character-wide field padded with leading zeros. `%x` prints `255` as lowercase hexadecimal `ff`.

---

3. Calculation: How many bytes does the string literal `"hello"` occupy in memory?

Answer: 6 bytes. The five visible characters `h`, `e`, `l`, `l`, `o` plus the null terminator `'\0'`.

4. Where is the bug?

```
double pi = 3.14159;
printf("Pi is %d\n", pi);
```

Answer: The format specifier `%d` expects an `int`, but `pi` is a `double`. This is undefined behavior and will print garbage. The correct specifier is `%f`:

```
printf("Pi is %f\n", pi);
```

5. Write a program that prints a 5x5 multiplication table using `printf` with width formatting so the columns are aligned.

Answer:

```
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 5; i++) {
        for (int j = 1; j <= 5; j++) {
            printf("%4d", i * j);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25
```

2. Variables

1. Think about it: In C++, `auto x = 42;` lets the compiler deduce the type. C has no `auto` type deduction. What advantage does requiring explicit types give to someone reading unfamiliar C code?

Answer: When every variable has an explicit type, you can read a function and immediately know the size, range, and semantics of each variable without tracing through initializers or return types. In C++ with `auto`, you sometimes need to figure out the return type of a function or the result of a complex expression to know what type a variable is. In C, `int count = 0;` tells you everything at a glance — it is a 4-byte (typically) signed integer. This is especially helpful in large codebases where you may be reading code written by someone else. The trade-off is more typing and more verbosity, but the explicitness reduces cognitive load when reading.

---

2. What does this print?

```
int a[] = {10, 20, 30, 40};
printf("%zu %zu\n", sizeof(a), sizeof(a[0]));
```

Answer:

```
16 4
```

`a` is an array of 4 `int` elements. On a typical system where `int` is 4 bytes, `sizeof(a)` is `4 * 4 = 16` bytes. `sizeof(a[0])` is the size of one `int`, which is 4 bytes. (The common idiom `sizeof(a) / sizeof(a[0])` gives the element count: 4.)

---

3. Calculation: Given the following declarations on a system where `int` is 4 bytes, what is `sizeof(grid)`?

```
int grid[3][5];
```

Answer: 60 bytes. The array has 3 rows of 5 `int` elements each, for a total of `3 * 5 = 15` elements. Each `int` is 4 bytes, so the total is `15 * 4 = 60` bytes.

---

4. Where is the bug?

```
const int MAX = 100;
int *p = &MAX;
*p = 200;
printf("MAX = %d\n", MAX);
```

Answer: The code takes the address of a `const int` and stores it in a non-`const` pointer `int *p`. This discards the `const` qualifier, and the compiler will warn about (or reject) this assignment. Then `*p = 200` attempts to modify the `const` variable through the pointer, which is **undefined behavior**. The `const` qualifier is a promise that the value will not change, and the compiler may place it in read-only memory. To fix this, respect the `const`:

```
const int MAX = 100;
const int *p = &MAX;    /* pointer to const int */
/* *p = 200; */         /* not allowed - and that's the point */
printf("MAX = %d\n", MAX);
```

---

5. What does this print?

```
struct point { int x; int y; };
struct point a = {3, 7};
struct point b = a;
b.x = 99;
printf("%d %d\n", a.x, b.x);
```

Answer:

```
3 99
```

Assigning `b = a` copies the raw bytes of `a` into `b`. After the copy, `a` and `b` are independent. Changing `b.x` to 99 does not affect `a.x`, which remains 3.

---

6. Think about it: In C, assigning one struct to another copies the raw bytes. What problem could this cause if the struct contains a `char *` member that points to dynamically allocated memory (via `malloc`)? How is this different from what C++ does by default?

Answer: If the struct contains a `char *` pointing to `malloc`-ed memory, the byte-for-byte copy duplicates the pointer value — both structs now point to the same dynamically allocated memory. This creates two problems: (1) if one struct is used to `free` the memory, the other struct's pointer becomes a dangling pointer, and (2) if both try to `free` the memory, you get a double-free bug.

In C++, the compiler generates a default copy constructor and `operator=` that perform a member-wise copy — which has the same shallow-copy problem for raw pointers. However, C++ lets you define custom copy constructors and assignment operators to perform a deep copy. C++ also provides `std::string` and smart pointers that handle this automatically. In C, you must write a manual copy function that allocates new memory and copies the string data.

---

7. Write a program that declares a `struct student` with fields `name` (a `char` array), `id` (an `int`), and `gpa` (a `double`). Create an array of 3 students, initialize them with values, and print each student's information using a loop. Use `%s`, `%d`, and `%.2f` in your `printf`.

Answer:

```
#include <stdio.h>

struct student {
    char name[50];
    int id;
    double gpa;
};
```

```
int main(void) {
    struct student roster[] = {
        {"Maria", 1001, 3.85},
        {"Carlos", 1002, 3.42},
        {"Elena", 1003, 3.97}
    };
    int n = sizeof(roster) / sizeof(roster[0]);

    for (int i = 0; i < n; i++) {
        printf("Name: %s, ID: %d, GPA: %.2f\n",
               roster[i].name, roster[i].id, roster[i].gpa);
    }
    return 0;
}
```

Output:

```
Name: Maria, ID: 1001, GPA: 3.85
Name: Carlos, ID: 1002, GPA: 3.42
Name: Elena, ID: 1003, GPA: 3.97
```

3. Expressions

1. Think about it: In C++, you can overload operators to give `+`, `<<`, `==`, and others custom meanings for your classes. C does not allow operator overloading. What advantage does this give you when reading unfamiliar C code? Can you think of a situation where operator overloading would have been genuinely useful in C?

Answer: Without operator overloading, every operator in C has exactly one meaning. When you see `a + b`, you know it is arithmetic addition — never a string concatenation, a matrix addition, or something a library author decided it should mean. This makes C code easier to read and reason about, especially in unfamiliar codebases. You never have to look up what `+` does for a particular type.

A situation where operator overloading would have been genuinely useful is complex number arithmetic. C99 added `_Complex` as a built-in type with arithmetic operator support, but before that, working with complex numbers meant calling functions like `complex_add(a, b)` and `complex_mul(a, b)`, which made mathematical formulas hard to read. With operator overloading, you could write `a + b * c` instead of `complex_add(a, complex_mul(b, c))`.

---

2. What does this print?

```
int x = 10;
int y = x++ + ++x;
printf("%d %d\n", x, y);
```

(Be careful — is the answer even defined?)

Answer: This is **undefined behavior**. The variable `x` is modified twice (`x++` and `++x`) without a sequence point between the modifications. The C standard says the result is undefined — the compiler is free to produce any output. Different compilers, or the same compiler with different optimization levels, may give different results. The correct approach is to never modify a variable more than once in the same expression. Use separate statements:

```
int x = 10;
x++;
int y = x + x;
x++;
```

---

3. Calculation: What is the result of each of these expressions?

```
25 / 4
25 % 4
-25 % 4
(1 << 4) | (1 << 1)
0xFF & 0x0F
```

Answer:

- `25 / 4` = **6**. Integer division truncates toward zero: 25 / 4 = 6.25, truncated to 6.
- `25 % 4` = **1**. The remainder of 25 / 4: 25 - (6 * 4) = 1.
- `-25 % 4` = **-1**. In C99+, `%` preserves the sign of the dividend (left operand). -25 / 4 = -6 (truncated toward zero), remainder = -25 - (-6 * 4) = -25 + 24 = -1.
- `(1 << 4) | (1 << 1)` = **18**. `1 << 4` is 16 (binary `10000`), `1 << 1` is 2 (binary `00010`). OR-ing them gives binary `10010` = 18.
- `0xFF & 0x0F` = **0x0F** (15). AND-ing `11111111` with `00001111` keeps only the low 4 bits: `00001111`.

---

4. Where is the bug?

```
int status = 0x07;
if (status & 0x04 == 0x04) {
    printf("Bit 2 is set\n");
}
```

Answer: Operator precedence. The `==` operator has higher precedence than `&`. So this evaluates as `status & (0x04 == 0x04)`, which is `status & 1`, which is `0x07 & 0x01` = 1. This happens to be nonzero (true), so the message prints —

but for the wrong reason. If `status` were `0x06` (bit 2 set, bit 0 not set), `0x06 & 1` would be `0` (false), even though bit 2 *is* set. The fix is to add parentheses:

```c
if ((status & 0x04) == 0x04) {
    printf("Bit 2 is set\n");
}
```

---

5. What does this print?

```c
int a = 5, b = 10;
a ^= b;
b ^= a;
a ^= b;
printf("a=%d b=%d\n", a, b);
```

Answer:

```
a=10 b=5
```

This is the classic XOR swap trick. Step by step:

1. `a ^= b` → `a = 5 ^ 10 = 15` (a=15, b=10)
2. `b ^= a` → `b = 10 ^ 15 = 5` (a=15, b=5)
3. `a ^= b` → `a = 15 ^ 5 = 10` (a=10, b=5)

The values of `a` and `b` are swapped without using a temporary variable.

---

6. Where is the bug?

```c
int count = 0;
if (count = 0) {
    printf("El contador es cero\n");
} else {
    printf("El contador no es cero\n");
}
```

Answer: The condition uses `=` (assignment) instead of `==` (comparison). `count = 0` assigns `0` to `count` and then evaluates to `0`, which is false. So the `else` branch always executes, printing `"El contador no es cero"` — the opposite of what was intended. The fix:

```c
if (count == 0) {
    printf("El contador es cero\n");
}
```

---

7. Write a program that takes an `unsigned int` and prints its value in binary (most significant bit first). Use bitwise operators to test each bit. Test it with the values `0`, `1`, `255`, and `1024`.

Answer:

```c
#include <stdio.h>
#include <limits.h>

void print_binary(unsigned int val) {
    int bits = sizeof(unsigned int) * CHAR_BIT;
    int started = 0;

    for (int i = bits - 1; i >= 0; i--) {
        int bit = (val >> i) & 1;
        if (bit)
            started = 1;
        if (started)
            printf("%d", bit);
    }
    if (!started)
        printf("0");
    printf("\n");
}

int main(void) {
    print_binary(0);       /* 0 */
    print_binary(1);       /* 1 */
    print_binary(255);     /* 11111111 */
    print_binary(1024);    /* 10000000000 */
    return 0;
}
```

Output:

```
0
1
11111111
10000000000
```

4. Control Flow

1. Think about it: C uses `0` for false and nonzero for true, while C++ has a built-in `bool` type. What practical problems can arise from using integers as booleans? Can you think of a case where a nonzero value that is not `1` might cause a subtle bug?

Answer: The main problem is that any nonzero value is true, which can cause subtle bugs when comparing boolean results. Consider this example:

```c
int is_ready = 2;     /* intended as "true" */
int is_set = 1;        /* also "true" */
if (is_ready == is_set) {
    printf("Both true\n");    /* never prints! */
}
```

Both variables are logically true, but `2 != 1`, so the comparison fails. This bug arises when you compare boolean values with `==` instead of testing them individually. Another common case is using bitwise AND when you mean logical AND:

```c
int a = 2;    /* binary: 10 */
int b = 1;    /* binary: 01 */
if (a & b) {
    /* never enters - bitwise AND of 2 and 1 is 0 */
}
```

Both `a` and `b` are logically true, but `a & b` is `0` (false) because they have no bits in common. The fix is to use `&&` for logical operations and to test boolean values with `if (x)` rather than `if (x == 1)`.

---

2. What does this print?

```c
for (int i = 0; i < 5; i++) {
    if (i == 3)
        continue;
    printf("%d ", i);
}
printf("\n");
```

Answer:

```
0 1 2 4
```

The loop iterates `i` from 0 to 4. When `i` is 3, `continue` skips the `printf` and jumps to the next iteration. So 0, 1, 2, and 4 are printed, but 3 is skipped.

---

3. What does this print?

```c
int x = 2;
switch (x) {
case 1:
    printf("uno ");
case 2:
    printf("dos ");
case 3:
    printf("tres ");
    break;
default:
    printf("other ");
}
printf("\n");
```

Answer:

```
dos tres
```

`x` is 2, so execution jumps to `case 2` and prints `"dos "`. There is no `break` after `case 2`, so execution **falls through** to `case 3` and prints `"tres "`. The `break` at the end of `case 3` exits the `switch`. The output is `dos tres` followed by a newline.

---

4. Where is the bug?

```
int total = 0;
int i;
for (i = 0; i < 10; i++);
{
    total += i;
}
printf("Total: %d\n", total);
```

Answer: There is a stray semicolon after the `for` statement: `for (i = 0; i < 10; i++);`. This semicolon is an empty statement, making it the entire body of the loop. The loop runs 10 times doing nothing. Then the block `{ total += i; }` executes once with `i` equal to 10 (its value after the loop finishes). So `total` is 10, not 45. The fix is to remove the semicolon:

```
int total = 0;
int i;
for (i = 0; i < 10; i++) {
    total += i;
}
printf("Total: %d\n", total);    /* prints Total: 45 */
```

---

5. Calculation: How many times does the body of this loop execute?

```
int count = 0;
int i = 10;
do {
    count++;
    i--;
} while (i > 10);
```

Answer: 1 time. A `do-while` loop always executes the body at least once before checking the condition. After the first iteration, `count` becomes 1 and `i` becomes 9. Then the condition `i > 10` is checked: `9 > 10` is false, so the loop exits. Even though the condition was never true, the body executed exactly once.

---

6. Where is the bug?

```c
int level = 5;
if (level = 10) {
    printf("Max level!\n");
}
```

Answer: The condition uses `=` (assignment) instead of `==` (comparison). `level = 10` assigns `10` to `level` and then evaluates to `10`, which is nonzero (true). So `"Max level!"` is always printed, regardless of the original value of `level`. The fix:

```c
if (level == 10) {
    printf("Max level!\n");
}
```

---

7. Write a program that reads integers from the user (using `scanf`) until the user enters `0`. Print the sum and average of all numbers entered (not counting the `0`). Use a `do-while` or `while` loop.

Answer:

```c
#include <stdio.h>

int main(void) {
    int num;
    int sum = 0;
    int count = 0;

    printf("Enter integers (0 to stop):\n");

    do {
        printf("> ");
        scanf("%d", &num);
        if (num != 0) {
            sum += num;
            count++;
        }
    } while (num != 0);

    if (count > 0) {
        printf("Sum: %d\n", sum);
        printf("Average: %.2f\n", (double)sum / count);
    } else {
        printf("No numbers entered.\n");
    }

    return 0;
}
```

Example session:

```
Enter integers (0 to stop):
> 10
> 20
> 30
> 0
Sum: 60
Average: 20.00
```

5. Pointers

1. Think about it: In C++, you can pass by reference to modify a caller's variable. Why do you think C was designed with only pass by value? What does this simplify in the language?

Answer: Pass by value only simplifies the language and its calling conventions. Every function call copies its arguments, so you always know that a function cannot modify your local variables unless you explicitly give it a pointer. This makes reasoning about code simpler: if you see `f(x)`, you know `x` cannot change (unless `x` is already a pointer). It also simplifies the compiler — there is no need for reference semantics, no hidden indirection, and no ambiguity about whether an argument is an alias for another variable. When you do want a function to modify a variable, you explicitly pass `&x`, making the intent clear at the call site.

---

2. What does this print?

```
int a[] = {10, 20, 30, 40, 50};
int *p = a + 2;
printf("%d %d %d\n", *p, *(p - 1), p[1]);
```

Answer:

```
30 20 40
```

`p` points to `a[2]` (value 30). `*(p - 1)` is `a[1]` (value 20). `p[1]` is `*(p + 1)`, which is `a[3]` (value 40).

---

3. Calculation: On a 64-bit system, what is `sizeof(int *)`, `sizeof(char *)`, and `sizeof(double *)`?

Answer: All three are **8 bytes**. On a 64-bit system, all pointers are 8 bytes (64 bits) regardless of the type they point to. The type only affects pointer arithmetic and dereferencing, not the size of the pointer itself.

---

4. Where is the bug?

```c
int *get_value(void) {
    int result = 42;
    return &result;
}
```

Answer: The function returns a pointer to a local variable. When `get_value` returns, `result` is destroyed (its stack frame is reclaimed), and the returned pointer becomes a **dangling pointer**. Dereferencing it is undefined behavior. To fix this, either return the value directly (`return result` with return type `int`), use a `static` local, or allocate memory with `malloc`.

---

5. What does this print?

```c
int x = 10;
int *p = &x;
int **pp = &p;
**pp = 20;
printf("%d\n", x);
```

Answer:

```
20
```

`pp` points to `p`, and `p` points to `x`. `**pp` dereferences `pp` to get `p`, then dereferences `p` to get `x`. Assigning `**pp = 20` changes `x` to 20.

---

6. Where is the bug?

```c
struct song {
    char title[40];
    int year;
};

struct song *p = NULL;
printf("%s\n", p->title);
```

Answer: `p` is `NULL`, so `p->title` dereferences a null pointer. This is undefined behavior and will typically crash with a segmentation fault. You must point `p` to a valid `struct song` before accessing its fields:

```
struct song track = {"Something", 1985};
struct song *p = &track;
printf("%s\n", p->title);
```

---

7. Write a program that declares an array of 5 integers, uses a pointer to iterate through the array, and prints each element along with its memory address.

Answer:

```
#include <stdio.h>

int main(void) {
    int arr[] = {10, 20, 30, 40, 50};
    int *p = arr;

    for (int i = 0; i < 5; i++) {
        printf("arr[%d] = %d at address %p\n", i, *(p + i), (void *)(p + i));
    }
    return 0;
}
```

Output (addresses will vary):

```
arr[0] = 10 at address 0x7ffd...
arr[1] = 20 at address 0x7ffd...
arr[2] = 30 at address 0x7ffd...
arr[3] = 40 at address 0x7ffd...
arr[4] = 50 at address 0x7ffd...
```

6. Functions

1. Think about it: C does not have function overloading. How does the C standard library handle providing similar functionality for different types? Look at `abs` (for `int`) and `fabs` (for `double`) as examples. What naming convention do you see?

Answer: The C standard library uses different function names for different types, often with a prefix or suffix that indicates the type. For absolute value: `abs` handles `int`, `labs` handles `long`, `llabs` handles `long long`, and `fabs` handles `double`. For math functions, the base name is for `double` (e.g., `sqrt`), an `f` suffix is for `float` (e.g., `sqrtf`), and an `l` suffix is for `long double` (e.g., `sqrtl`).

This naming convention is consistent across the standard library: `printf` vs `fprintf` vs `sprintf`, `strtol` vs `strtod` vs `strtof`. The type information that C++ encodes in overload resolution is encoded in the function name in C. The downside is more names to remember, but the upside is that the type being operated on is always visible at the call site.

---

2. What does this print?

```c
void mystery(int x) {
    x = x * 2;
    printf("inside: %d\n", x);
}

int main(void) {
    int val = 5;
    mystery(val);
    printf("outside: %d\n", val);
    return 0;
}
```

Answer:

```
inside: 10
outside: 5
```

C is pass by value. The function `mystery` receives a copy of `val`. Doubling `x` inside the function modifies the copy, not the original. After `mystery` returns, `val` in `main` is still 5.

---

3. Where is the bug?

```c
int count_chars(const char *s) {
    int count;
    while (*s != '\0') {
        count++;
        s++;
    }
    return count;
}
```

Answer: The variable `count` is not initialized. In C, local variables have **indeterminate** (garbage) values when declared without an initializer. Incrementing an uninitialized variable is undefined behavior, and the function will return an unpredictable result. The fix:

```c
int count = 0;
```

---

4. Calculation: Given the struct below, approximately how many bytes are copied each time `process_data` is called with pass by value? Assume `int` is 4 bytes.

```c
struct data {
    int values[1000];
    int count;
};

void process_data(struct data d) { /* ... */ }
```

Answer: 4004 bytes. The `values` array is `1000 * 4 = 4000` bytes. The `count` field is 4 bytes. The total struct size is at least 4004 bytes (possibly more due to padding, but typically no padding is needed here since all fields are `int`). Every call to `process_data` copies all 4004 bytes onto the stack. This is why passing large structs by `const` pointer (`const struct data *d`) is strongly preferred.

---

5. What does this print?

```c
int apply(int (*fn)(int, int), int a, int b) {
    return fn(a, b);
}

int mul(int a, int b) { return a * b; }

int main(void) {
    printf("%d\n", apply(mul, 6, 7));
    return 0;
}
```

Answer:

```
42
```

`apply` takes a function pointer `fn` and two integers. It calls `fn(6, 7)`. Since `fn` points to `mul`, this computes `6 * 7 = 42`.

---

6. Where is the bug?

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(void) {
    int x = 10, y = 20;
    swap(x, y);
    printf("x=%d y=%d\n", x, y);
    return 0;
}
```

Answer: The `swap` function takes its parameters by value, so `a` and `b` are local copies. Swapping them has no effect on the caller's `x` and `y`. The program prints `x=10 y=20`. In C, you must pass pointers to modify the caller's variables:

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void) {
    int x = 10, y = 20;
    swap(&x, &y);
    printf("x=%d y=%d\n", x, y);    /* x=20 y=10 */
    return 0;
}
```

---

7. Write a program that defines a function `void transform(int *arr, int n, int (*fn)(int))` which applies the function `fn` to each element of `arr`, modifying the array in place. Test it with a function that doubles each element and another that negates each element.

Answer:

```
#include <stdio.h>

int double_it(int x) { return x * 2; }
int negate(int x) { return -x; }

void transform(int *arr, int n, int (*fn)(int)) {
    for (int i = 0; i < n; i++) {
        arr[i] = fn(arr[i]);
    }
}

void print_array(const int *arr, int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main(void) {
    int nums[] = {1, 2, 3, 4, 5};
    int n = sizeof(nums) / sizeof(nums[0]);

    printf("Original: ");
    print_array(nums, n);

    transform(nums, n, double_it);
    printf("Doubled:  ");
    print_array(nums, n);

    transform(nums, n, negate);
    printf("Negated:  ");
    print_array(nums, n);

    return 0;
}
```

Output:

```
Original: 1 2 3 4 5
Doubled:  2 4 6 8 10
Negated:  -2 -4 -6 -8 -10
```

7. Allocating Memory

1. Think about it: Why would you choose `calloc` over `malloc` followed by `memset` to zero?

Answer: `calloc` combines allocation and zeroing in a single call, which is simpler and less error-prone. It also takes the element count and size as separate arguments (`calloc(n, sizeof(int))`), which protects against integer overflow — `calloc` checks whether `n * size` overflows before allocating, while `malloc(n * sizeof(int))` silently overflows if `n` is very large. Using `calloc` is also clearer about intent: "I want `n` zero-initialized elements."

---

2. What does this print?

```
#include <stdio.h>

void counter(void) {
    static int n = 0;
    n++;
    printf("%d ", n);
}

int main(void) {
    counter(); counter(); counter();
    return 0;
}
```

Answer:

```
1 2 3
```

The `static` keyword means `n` is initialized once and retains its value between calls. The first call increments `n` from 0 to 1, the second from 1 to 2, and the third from 2 to 3.

---

3. Calculation: On a system where `int` is 32 bits, how many bytes does `malloc(5 * sizeof(int))` allocate?

Answer: 20 bytes. An `int` that is 32 bits is 4 bytes, and `5 * 4 = 20`.

---

4. Where is the bug?

```
int *p = malloc(10 * sizeof(int));
for (int i = 0; i < 10; i++) {
    p[i] = i;
}
free(p);
printf("%d\n", p[0]);
```

Answer: The code accesses `p[0]` after calling `free(p)`. This is a **use-after-free** bug — undefined behavior. Once memory is freed, the pointer becomes invalid and must not be dereferenced. The `printf` must be moved before the `free`, or the code should be restructured so the data is no longer needed after freeing.

---

5. Where is the bug?

```
int *a = malloc(5 * sizeof(int));
int *b = a;
free(a);
free(b);
```

Answer: This is a **double free**. Both `a` and `b` point to the same allocated memory. After `free(a)`, that memory is released. Calling `free(b)` frees the same block a second time, which is undefined behavior and can corrupt the heap. The fix is to only free the memory once. If you need to track that it has been freed, set the pointer to `NULL` after freeing:

```
free(a);
a = NULL;
b = NULL;
```

6. Write a program that uses `malloc` to allocate an array of `n` integers (where `n` is provided by the user via `scanf`), fills the array with squares (0, 1, 4, 9, . . .), prints them, and frees the memory.

Answer:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);

    int *arr = malloc(n * sizeof(int));
    if (arr == NULL) {
        fprintf(stderr, "Allocation failed\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = i * i;
    }

    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Example output for `n = 5`:

```
0 1 4 9 16
```

8. Strings

1. Think about it: Why does `strcmp` return 0 for equal strings rather than 1? How does this relate to the function's actual purpose?

Answer: `strcmp` computes the difference between two strings, not a boolean "are they equal?" check. It returns a negative value if the first string comes before the second lexicographically, a positive value if it comes after, and zero if there is no difference. Zero means "no difference," which is the natural result for equal strings. This three-way return value is useful for sorting (e.g., with `qsort`). If it returned 1 for equal, you would lose the ability to determine ordering in a single call.

---

2. What does this print?

```c
char s[] = "Ghostbusters";
printf("%zu %zu\n", strlen(s), sizeof(s));
```

Answer:

```
12 13
```

`strlen(s)` returns 12 — the number of characters before the null terminator. `sizeof(s)` returns 13 — the total size of the array in bytes, including the null terminator.

---

3. Calculation: What is `sizeof(buf)` for `char buf[20] = "Hola";`?

Answer: 20. `sizeof` returns the size of the declared array, not the length of the string stored in it. The array is declared as 20 bytes, so `sizeof(buf)` is 20 regardless of the string's content.

---

4. Where is the bug?

```c
char buf[10] = "Livin'";
strcat(buf, " on a Prayer");
printf("%s\n", buf);
```

Answer: Buffer overflow. `buf` is 10 bytes. `"Livin'"` uses 7 bytes (6 characters + `'\0'`), leaving 3 bytes free. `" on a Prayer"` is 12 characters, so the combined string needs 19 bytes (6 + 12 + `'\0'`), which far exceeds the 10-byte buffer. `strcat` writes past the end of `buf`, causing undefined behavior.

---

5. Where is the bug?

```c
char *a = "Hello";
char *b = "Hello";
if (a == b) {
    printf("Equal\n");
} else {
    printf("Not equal\n");
}
```

Answer: The `==` operator compares the **addresses** of the two string literals, not their contents. The output is **not guaranteed**. Some compilers merge identical string literals (so `a == b` would be true), but others store them separately (so `a == b` would be false). To correctly compare string content, use `strcmp`:

```
if (strcmp(a, b) == 0) {
    printf("Equal\n");
}
```

---

6. Write a program that reads a string from the user, reverses it in place using pointer arithmetic, and prints the result.

Answer:

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char buf[100];
    printf("Enter a string: ");
    if (fgets(buf, sizeof(buf), stdin) == NULL) {
        return 1;
    }

    /* Remove trailing newline from fgets */
    size_t len = strlen(buf);
    if (len > 0 && buf[len - 1] == '\n') {
        buf[len - 1] = '\0';
        len--;
    }

    /* Reverse in place using two pointers */
    char *left = buf;
    char *right = buf + len - 1;
    while (left < right) {
        char tmp = *left;
        *left = *right;
        *right = tmp;
        left++;
        right--;
    }

    printf("Reversed: %s\n", buf);
    return 0;
}
```

Example:

```
Enter a string: Hola
```

```
Reversed: aloH
```

9. Numbers and Casting

1. Think about it: C++ provides several different cast operators (`static_cast`, `reinterpret_cast`, etc.) whereas C provides only one. What are the advantages of C++'s approach over C's single cast syntax?

Answer: C++'s multiple cast operators provide better type safety and express intent more clearly. `static_cast` restricts you to compatible types, preventing accidental casts between pointers and integers. `reinterpret_cast` explicitly warns the reader (and compiler) that you are doing something inherently unsafe with bits. By using different named casts, developers can search for dangerous casts (`reinterpret_cast`) and the compiler can prevent logic errors that a C-style cast would silently accept. C's single cast syntax does whatever is necessary to force the compilation to succeed, often hiding real bugs when used incorrectly.

---

2. What does this print?

```
char letter = 'C';
printf("%c %d\n", letter + 2, letter + 2);
```

Answer:

```
E 69
```

`'C'` has the ASCII value 67. Adding 2 gives 69, which corresponds to `'E'` in ASCII. The `%c` specifier prints the character `'E'`, and `%d` prints the numeric value 69.

---

3. Calculation: Assuming a 64-bit system where pointers are 8 bytes and `int` is 4 bytes, what is the output of `sizeof("Danger")` and what is the output of `sizeof((int)0)`?

Answer: `sizeof("Danger")` is **7**. The string "Danger" has 6 characters plus the null terminator \0. `sizeof((int)0)` is **4**. Casting `0` to an `int` yields an integer value, which takes 4 bytes.

---

4. Where is the bug?

```
#include <stdio.h>

int main(void) {
    char *score_str = "100";
    int score = (int)score_str;
    printf("You got a %d percent!\n", score);
    return 0;
}
```

Answer: This code attempts to parse a string by casting the character pointer to an integer. This is a logic error. Instead of converting the text `"100"` to the value `100`, it truncates the memory address where the string literal is stored into an `int`, resulting in a meaningless number. To parse strings to integers, use `strtol`:

```
long score = strtol(score_str, NULL, 10);
```

---

5. Write a program that declares a `double` variable with a fractional component and uses casting to separate the integer part from the fractional part. Print both pieces.

Answer:

```
#include <stdio.h>

int main(void) {
    double value = 1986.55;

    // Cast to truncate to purely the integer part
    int whole = (int)value;

    // Subtract to find the remainder
    double fractional = value - whole;

    printf("Original: %f\n", value);
    printf("Whole part: %d\n", whole);
    printf("Fractional part: %f\n", fractional);

    return 0;
}
```

10. Standard I/O

1. Think about it: Why does `scanf` need the `&` operator for scalar variables but not for arrays?

Answer: `scanf` needs a pointer to the location where it should store the value it reads. For a scalar variable like `int x`, `x` is the value, not an address, so you must pass `&x` to give `scanf` the address. For an array like `char name[50]`, the array name already decays to a pointer to its first element. Writing `name` in an expression is equivalent to `&name[0]`, so no explicit `&` is needed.

---

2. What does this print?

```c
char buf[50];
sprintf(buf, "%s: %d", "Score", 100);
printf("%zu\n", strlen(buf));
```

Answer:

```
10
```

`sprintf` writes `"Score: 100"` into `buf`. That string is 10 characters: "Score" is 5, ":" is 2, "100" is 3, totaling 10. `strlen` returns 10.

---

3. Calculation: If `buf` is declared as `char buf[20]` and you call `snprintf(buf, sizeof(buf), "Year: %d", 1984)`, how many characters (excluding the null terminator) are written to `buf`?

Answer: 10. The formatted string is `"Year: 1984"` which is 10 characters. Since this fits within the 20-byte buffer, all 10 characters are written (plus a null terminator).

---

4. Where is the bug?

```c
int x;
scanf("%d", x);
```

Answer: Missing the `&` operator. `scanf` needs the **address** of `x` to store the value, but this passes the **value** of `x` (which is uninitialized). This is undefined behavior and will likely crash. The fix:

```c
scanf("%d", &x);
```

---

5. Where is the bug?

```c
FILE *f = fopen("noexist.txt", "r");
fprintf(f, "Hello\n");
fclose(f);
```

Answer: The code does not check whether `fopen` succeeded. If the file does not exist, `fopen` returns `NULL`, and `fprintf(NULL, ...)` is undefined behavior (typically a crash). The fix:

```c
FILE *f = fopen("noexist.txt", "r");
if (f == NULL) {
    perror("fopen");
    return 1;
}
```

```c
fprintf(f, "Hello\n");
fclose(f);
```

Also, opening a nonexistent file with `"r"` (read) will always fail. If the intent is to write, the mode should be `"w"`.

---

6. Think about it: You run `./program > output.txt` and your program contains both `printf` and `fprintf(stderr, ...)` calls. Which messages appear in `output.txt` and which appear on the screen? Why?

Answer: The `>` operator redirects `stdout` to `output.txt`, so all `printf` output (which goes to `stdout`) ends up in the file. `stderr` is **not** redirected by `>`, so `fprintf(stderr, ...)` messages still appear on the screen. This is by design — error messages should remain visible even when normal output is redirected. To redirect `stderr` as well, you would use `2> errors.txt` or `2>&1` to merge it with `stdout`.

---

7. Write a program that opens a text file, writes five lines to it (your choice of content), closes it, reopens it for reading, reads and prints each line using `fgets`, then closes it again.

Answer:

```c
#include <stdio.h>

int main(void) {
    FILE *f = fopen("test_output.txt", "w");
    if (f == NULL) {
        perror("fopen write");
        return 1;
    }

    fprintf(f, "Line 1: Hola\n");
    fprintf(f, "Line 2: Que tal\n");
    fprintf(f, "Line 3: Muy bien\n");
    fprintf(f, "Line 4: Gracias\n");
    fprintf(f, "Line 5: Adios\n");
    fclose(f);

    f = fopen("test_output.txt", "r");
    if (f == NULL) {
        perror("fopen read");
        return 1;
    }

    char line[100];
    while (fgets(line, sizeof(line), f) != NULL) {
        printf("%s", line);
    }
    fclose(f);

    return 0;
}
```

Output:

```
Line 1: Hola
Line 2: Que tal
Line 3: Muy bien
Line 4: Gracias
Line 5: Adios
```

11. Low-Level I/O

1. Think about it: Why would you use low-level `read`/`write` instead of `fprintf`/`fscanf`? When would `stdio` be the better choice?

Answer: Low-level I/O is useful when you need precise control: writing binary data without translation, avoiding buffering overhead, working with file descriptors from system calls (e.g., `pipe`, `socket`), or performing atomic operations with `pread`/`pwrite`. It is also necessary when interacting with devices or special files.

`stdio` is the better choice for formatted text I/O — it provides convenient formatting with `printf`/`scanf`, handles buffering automatically for better performance on many small writes, and is portable across all platforms. For most programs that read and write text files, `stdio` is simpler and sufficient.

---

2. What does this print?

```
write(1, "ABC", 3);
write(1, "DEF\n", 4);
```

Answer:

```
ABCDEF
```

The first `write` outputs `ABC` (no newline). The second outputs `DEF\n`. Since `write` does no buffering, the bytes appear immediately and consecutively, producing `ABCDEF` followed by a newline.

---

3. Calculation: If `read(fd, buf, 1024)` returns 512, what does that tell you? Does it mean there was an error?

Answer: No, it is not an error. A return value of 512 means that 512 bytes were successfully read. `read` can return fewer bytes than requested for several reasons: the file may have only 512 bytes remaining, the data is coming from a pipe or socket and only 512 bytes are available right now, or the operating system interrupted the read. A return value of 0 means end of file, and -1 means an error occurred. Any positive value is a successful read.

---

4. Where is the bug?

```
int fd = open("newfile.txt", O_WRONLY | O_CREAT);
write(fd, "Hello\n", 6);
close(fd);
```

Answer: When `O_CREAT` is used, `open` requires a third argument specifying the file permissions. Without it, the permissions are undefined (whatever garbage value happens to be on the stack). The fix:

```
int fd = open("newfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

---

5. Think about it: Explain the difference between `lseek(fd, 0, SEEK_END)` and `lseek(fd, -1, SEEK_END)`. What does each return?

Answer: `lseek(fd, 0, SEEK_END)` moves the file position to exactly the end of the file — one byte past the last byte. It returns the file size in bytes. For a 100-byte file, it returns 100 and positions at byte 100 (where a subsequent `read` would return 0 since there is no data past the end).

`lseek(fd, -1, SEEK_END)` moves to one byte *before* the end — the position of the last byte in the file. It returns the file size minus 1. For a 100-byte file, it returns 99 and positions at byte 99 (the last byte). A subsequent `read` would read that one last byte.

---

6. Write a program that uses low-level I/O (`open`, `read`, `write`, `close`) to copy the contents of one file to another. The source and destination filenames should be taken from `argv`.

Answer:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        write(2, "Usage: copy src dst\n", 20);
        return 1;
    }

    int src = open(argv[1], O_RDONLY);
    if (src == -1) {
        perror("open source");
        return 1;
    }

    int dst = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dst == -1) {
        perror("open dest");
        close(src);
        return 1;
    }

    char buf[4096];
    ssize_t n;
    while ((n = read(src, buf, sizeof(buf))) > 0) {
        ssize_t written = 0;
        while (written < n) {
            ssize_t w = write(dst, buf + written, n - written);
            if (w == -1) {
                perror("write");
                close(src);
                close(dst);
                return 1;
            }
            written += w;
        }
    }

    if (n == -1) {
        perror("read");
    }

    close(src);
    close(dst);
    return (n == -1) ? 1 : 0;
}
```

12. Odds and Ends

1. Think about it: In C++ you would use exceptions for error handling. In C there are no exceptions. What strategies can you use to handle errors in deeply nested function calls? When is `exit` appropriate and when is it not?

Answer: The main strategies are:

- **Return codes:** Each function returns an error code (0 for success, -1 or an error enum for failure). The caller checks the return value and either handles the error or propagates it up. This is the most common approach.
- **`errno` and `perror`:** Many C library functions set `errno` on failure. You can call `perror` or `strerror(errno)` to get a human-readable message.
- **`goto` cleanup:** When a function acquires multiple resources, use `goto` to jump to cleanup labels that release resources in reverse order. This is C's version of RAII.
- **`exit`:** Terminates the program immediately.

`exit` is appropriate for truly fatal errors where there is no reasonable recovery — for example, failing to open a required configuration file at startup, or running out of memory for a critical allocation. It is not appropriate in libraries (which should let the caller decide how to handle errors) or when partial cleanup is needed (since `exit` bypasses the normal unwinding of the call stack).

---

2. What happens here?

```c
#include <stdlib.h>
#include <stdio.h>

void cleanup(void) {
    printf("Adios!\n");
}

int main(void) {
    atexit(cleanup);
    printf("Starting...\n");
    exit(0);
}
```

Answer:

```
Starting...
Adios!
```

`atexit` registers `cleanup` to be called when the program exits. `printf` prints `"Starting..."`, then `exit(0)` triggers the `atexit` handlers, which calls `cleanup`, printing `"Adios!"`. The `exit` function flushes all `stdio` streams and calls `atexit` handlers before terminating.

---

3. Where is the bug?

```c
char *get_greeting(void) {
    char buf[50];
    sprintf(buf, "Hola, mundo");
    return buf;
}
```

Answer: The function returns a pointer to the local array `buf`. When `get_greeting` returns, `buf` is destroyed (its stack frame is reclaimed), and the returned pointer becomes a **dangling pointer**. Dereferencing it is undefined behavior. The fix is to use `strdup` to allocate heap memory:

```
char *get_greeting(void) {
    char buf[50];
    sprintf(buf, "Hola, mundo");
    return strdup(buf);  /* caller must free */
}
```

Or have the caller pass in a buffer:

```
void get_greeting(char *buf, size_t size) {
    snprintf(buf, size, "Hola, mundo");
}
```

---

4. Think about it: You call a function `char *get_name(int id)` from a library. How would you determine whether you need to `free` the returned pointer?

Answer: Check the documentation. Look for phrases like "the caller must free the returned pointer" or "the returned pointer points to a static buffer." If the documentation does not say, read the source code to see if the function calls `malloc`, `calloc`, or `strdup` internally. You can also check if the function returns a pointer to a `static` buffer (which you must not free) or to a field within a library-managed struct. As a last resort, you can test: call the function twice and see if both returned pointers point to the same address (suggesting a static buffer). When in doubt, consult the library's header files for ownership annotations or comments.

---

5. Where is the bug? (Hint: ownership)

```
char *name = strdup("Walking on Sunshine");
char *alias = name;
free(name);
printf("%s\n", alias);
```

Answer: `alias` is assigned the same address as `name`. After `free(name)`, both `name` and `alias` point to freed memory. The `printf` dereferences `alias`, which is a **use-after-free** — undefined behavior. The fix is to either print before freeing, or duplicate the string for `alias`:

```c
char *alias = strdup(name);
free(name);
printf("%s\n", alias);
free(alias);
```

---

6. Calculation: Given `int nums[] = {5, 10, 15, 20};`, what is the value of `sizeof(nums) / sizeof(nums[0])`?

Answer: 4. The array contains 4 `int` elements. `sizeof(nums)` is `4 * sizeof(int)` (typically 16 bytes), and `sizeof(nums[0])` is `sizeof(int)` (typically 4 bytes). Dividing gives the number of elements: 4. This is a common C idiom for computing array length at compile time.

---

7. What does this print?

```c
int compare_desc(const void *a, const void *b) {
    int ia = *(const int *)a;
    int ib = *(const int *)b;
    return (ib > ia) - (ib < ia);
}

int main(void) {
    int vals[] = {3, 1, 4, 1, 5};
    qsort(vals, 5, sizeof(int), compare_desc);
    printf("%d %d %d %d %d\n", vals[0], vals[1], vals[2], vals[3], vals[4]);
    return 0;
}
```

Answer:

```
5 4 3 1 1
```

The comparison function compares `ib` to `ia` (instead of `ia` to `ib`), which reverses the sort order. `qsort` sorts the array in **descending** order: 5, 4, 3, 1, 1.

---

8. Write a program that uses `qsort` to sort an array of strings in reverse alphabetical order. Write a custom comparison function that calls `strcmp` with the arguments swapped.

Answer:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int compare_reverse(const void *a, const void *b) {
    const char *sa = *(const char **)a;
    const char *sb = *(const char **)b;
    return strcmp(sb, sa);  /* swapped for reverse order */
}

int main(void) {
    const char *words[] = {"delta", "alpha", "charlie", "bravo", "echo"};
    int n = sizeof(words) / sizeof(words[0]);

    qsort(words, n, sizeof(char *), compare_reverse);

    for (int i = 0; i < n; i++) {
        printf("%s\n", words[i]);
    }
    return 0;
}
```

Output:

```
echo
delta
charlie
bravo
alpha
```

---

9. Write a program in C++ that uses `extern "C"` to call the C function `strlen` from `<string.h>`, passes it a string, and prints the result. Compile it with `c++` to verify it works.

Answer:

```
#include <cstdio>

extern "C" {
    #include <string.h>
}

int main() {
    const char *msg = "Hola, amigo";
    size_t len = strlen(msg);
    printf("'%s' has %zu characters\n", msg, len);
    return 0;
}
```

Output:

```
'Hola, amigo' has 11 characters
```

Note: In practice, C++ standard headers like `<cstring>` already handle the `extern "C"` linkage for you. This exercise demonstrates the mechanism explicitly.

Appendix A: Macros

1. Think about it: C++ uses `constexpr` and `inline` functions to replace many uses of macros. What specific problems do macros have that these C++ features solve? Why does C still rely on macros despite these problems?

Answer: Macros have several problems that C++ features solve:

- **No type safety.** Macros operate on text, not types. `constexpr` and `inline` functions are type-checked by the compiler, catching mismatches at compile time.
- **Double evaluation.** `SQUARE(i++)` evaluates `i++` twice. A real `inline` function evaluates its argument once.
- **No scope.** A `#define` is visible from its point of definition to the end of the file (or until `#undef`). `constexpr` variables and `inline` functions respect block scope and namespaces.
- **Hard to debug.** Debuggers step through source code, but macros are expanded before the compiler sees the code, so you cannot step into a macro.
- **No recursion.** Macros cannot call themselves recursively.

C still relies on macros because it lacks alternatives. C has no `constexpr` (until C23's limited version), no templates, and `inline` is only a hint — the compiler may ignore it. Macros remain the only way to do conditional compilation (`#ifdef`), include guards, and compile-time code generation like X-macros.

---

2. What does this produce?

```
#define DOUBLE(x) ((x) + (x))

int i = 5;
printf("%d\n", DOUBLE(i++));
```

Answer: Undefined behavior. `DOUBLE(i++)` expands to `((i++) + (i++))`. This modifies `i` twice without a sequence point between the modifications, which is undefined behavior in C. The compiler is free to produce any result. In practice, many compilers will produce `11` (5 + 6) or `10` (5 + 5), but you cannot rely on any particular output. This is the classic double-evaluation trap — never pass expressions with side effects to function-like macros.

---

3. Calculation: Given the macro `#define BUFSIZE 256`, how many bytes does `char buf[BUFSIZE + 1]` allocate? Why is the `+ 1` a common pattern?

Answer: 257 bytes. The preprocessor substitutes `BUFSIZE` with `256`, giving `char buf[256 + 1]`, which is `char buf[257]`. The `+ 1` is a common pattern because C strings need a null terminator `'\0'`. If you want to store strings of up to 256 characters, you need 257 bytes — 256 for the characters plus 1 for the null terminator.

---

4. Where is the bug?

```
#define MUL(a, b)  a * b

int result = MUL(2 + 3, 4 + 5);
printf("%d\n", result);
```

Answer: Missing parentheses in the macro. `MUL(2 + 3, 4 + 5)` expands to `2 + 3 * 4 + 5`. Due to operator precedence, `3 * 4` is evaluated first, giving `2 + 12 + 5 = 19` instead of the intended `5 * 9 = 45`. The fix is to parenthesize each parameter use and the entire body:

```
#define MUL(a, b)  ((a) * (b))
```

Now `MUL(2 + 3, 4 + 5)` expands to `((2 + 3) * (4 + 5))` = `5 * 9` = `45`.

---

5. What does this produce?

```
#define STRINGIFY(x)  #x
#define XSTRINGIFY(x) STRINGIFY(x)
#define VERSION 3

printf("[%s] [%s]\n", STRINGIFY(VERSION), XSTRINGIFY(VERSION));
```

Answer:

```
[VERSION] [3]
```

`STRINGIFY(VERSION)` stringifies its argument *before* expansion, producing the literal string `"VERSION"`. `XSTRINGIFY(VERSION)` first expands `VERSION` to `3` (because the outer macro does not use `#`), then passes `3` to `STRINGIFY`, producing `"3"`. This is why the two-level indirect pattern is needed when you want the expanded value as a string.

---

6. Where is the bug?

```
#define LOG_IF(cond, msg) \
    if (cond) \
        printf("[WARN] %s\n", msg);
```

```
if (x > 100)
    LOG_IF(x > 200, "very high");
else
    printf("normal\n");
```

Answer: The macro creates a **dangling else** problem. After expansion, the code becomes:

```
if (x > 100)
    if (x > 200)
        printf("[WARN] %s\n", "very high");
;
else
    printf("normal\n");
```

The `else` binds to the inner `if` (from the macro), not the outer `if`. Also, the semicolon after `LOG_IF(...)` becomes a separate empty statement, which terminates the outer `if` before the `else` — causing a syntax error. The fix is to wrap the macro body in `do { ... } while (0)`:

```
#define LOG_IF(cond, msg) do { \
    if (cond)                  \
        printf("[WARN] %s\n", msg); \
} while (0)
```

---

7. Write a program that defines an X-macro list of at least four colors, then uses it to generate both an `enum` and a function that returns the string name for a given enum value. Print each color's enum value and name.

Answer:

```
#include <stdio.h>

#define COLORS(X) \
    X(RED)        \
    X(GREEN)      \
    X(BLUE)       \
    X(YELLOW)

#define AS_ENUM(name) name,
enum color { COLORS(AS_ENUM) COLOR_COUNT };

#define AS_STRING(name) #name,
const char *color_names[] = { COLORS(AS_STRING) };

const char *color_name(enum color c) {
    if (c >= 0 && c < COLOR_COUNT) {
```

```c
        return color_names[c];
    }
    return "UNKNOWN";
}

int main(void) {
    for (int i = 0; i < COLOR_COUNT; i++) {
        printf("%d = %s\n", i, color_name(i));
    }
    return 0;
}
```

Output:

```
0 = RED
1 = GREEN
2 = BLUE
3 = YELLOW
```