PROBLEM SOLVNG & C PROGRAMMNG

Puzzles

Below are some interesting puzzles and their explanations on C structure and union:

Puzzle 1: Size of Structure

```
#include <stdio.h>

struct A {
   char c;
   int i;
   double d;
};

int main() {
   printf("Size of struct A: %lu\n", sizeof(struct A));
   return 0;
}
```

Question: What will the output be, and why?

Explanation:
The size of the structure depends on memory alignment. The alignment rules for each data type determine padding to ensure efficient access.
For most systems:

- char takes 1 byte.
- int takes 4 bytes.
- double takes 8 bytes.
  The actual size of the structure will include padding to align the members properly.

Puzzle 2: Union Behavior

```
#include <stdio.h>

union B {
   int i;
```

```
    float f;
    char c;
};

int main() {
    union B u;
    u.i = 65;
    printf("u.i: %d\n", u.i);
    printf("u.f: %f\n", u.f);
    printf("u.c: %c\n", u.c);
    return 0;
}
```

Question: What will be the output, and why?

Explanation:
In a union, all members share the same memory location. Setting one member (i) and reading another (f or c) can result in unpredictable behavior due to differences in how data is interpreted.

Puzzle 3: Nested Structure

```
#include <stdio.h>

struct C {
    int i;
    struct {
        char c;
        double d;
    } nested;
};

int main() {
    struct C obj = {10, {'A', 3.14}};
    printf("obj.i: %d, obj.nested.c: %c, obj.nested.d: %lf\n", obj.i, obj.nested.c, obj.nested.d);
    return 0;
}
```

Question: What will the output be, and how is the memory laid out?

Puzzle 4: Comparing Structure and Union

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

```
struct D {
   int i;
   char c;
};

union E {
   int i;
   char c;
};

int main() {
   printf("Size of struct D: %lu\n", sizeof(struct D));
   printf("Size of union E: %lu\n", sizeof(union E));
   return 0;
}
```

Question: Why is the size of the union smaller or equal to the size of the structure?

Explanation:
The size of a structure is the sum of its members plus padding, while the size of a union is determined by its largest member.

Puzzle 5: Array in Structures

```
#include <stdio.h>

struct F {
   int arr[5];
};

int main() {
   struct F obj = {1, 2, 3, 4, 5};
   printf("First element: %d, Last element: %d\n", obj.arr[0], obj.arr[4]);
   return 0;
}
```

Question: Can you initialize the array directly in this way?

Here are some puzzles that focus on macros, #define, #ifdef, #undef, #include, and more.

Puzzle 1: Macro Expansion

```
#include <stdio.h>

#define SQUARE(x) x * x

int main() {
    int a = 4;
    int b = SQUARE(a + 1);
    printf("Result: %d\n", b);
    return 0;
}
```

Question: What will the output be, and why?

Explanation:
The macro SQUARE(x) performs text substitution. The expression SQUARE(a + 1) becomes a + 1 * a + 1. Without parentheses, operator precedence leads to unexpected results.

Puzzle 2: Conditional Compilation

```
#include <stdio.h>

#define DEBUG

int main() {
#ifdef DEBUG
    printf("Debug mode enabled.\n");
#else
    printf("Debug mode disabled.\n");
#endif
    return 0;
}
```

Question: What will the output be if you comment out the #define DEBUG line?

Puzzle 3: Redefinition of Macros

```
#include <stdio.h>

#define VALUE 10
#define VALUE 20

int main() {
```

```
    printf("VALUE: %d\n", VALUE);
    return 0;
}
```

Question: Will this program compile? If yes, what will it print?

Hint: Redefining macros without #undef causes issues in some compilers.

Puzzle 4: Nested Macros

```
#include <stdio.h>

#define MULTIPLY(a, b) a * b
#define DOUBLE(x) MULTIPLY(x, 2)

int main() {
    int result = DOUBLE(3 + 1);
    printf("Result: %d\n", result);
    return 0;
}
```

Question: What will the output be, and how is the macro expanded?

Puzzle 5: Token Pasting (##)

```
#include <stdio.h>

#define CONCAT(a, b) a##b

int main() {
    int xy = 100;
    printf("Result: %d\n", CONCAT(x, y));
    return 0;
}
```

Question: What will the program output, and what does the ## operator do?

Puzzle 6: Include Guard

```
// file1.h
```

```
#ifndef FILE1_H
#define FILE1_H

#define VALUE 50

#endif

// main.c
#include <stdio.h>
#include "file1.h"
#include "file1.h"

int main() {
   printf("VALUE: %d\n", VALUE);
   return 0;
}
```

Question: Will the program compile successfully, and why?

Puzzle 7: Macro with Variable Arguments

```
#include <stdio.h>

#define PRINT_VAR(x, ...) printf(x, __VA_ARGS__)

int main() {
   PRINT_VAR("Sum of %d and %d is %d\n", 3, 4, 3 + 4);
   return 0;
}
```

Question: How does the __VA_ARGS__ work here, and what will the output be?

Puzzle 8: Undefining Macros

```
#include <stdio.h>

#define MESSAGE "Hello, World!"

#undef MESSAGE

int main() {
#ifdef MESSAGE
```

```
    printf("%s\n", MESSAGE);
#else
    printf("MESSAGE is undefined.\n");
#endif
    return 0;
}
```

Question: What will the program output, and why?

Puzzle 9: Predefined Macros

```
#include <stdio.h>

int main() {
    printf("File: %s\n", __FILE__);
    printf("Line: %d\n", __LINE__);
    printf("Date: %s\n", __DATE__);
    printf("Time: %s\n", __TIME__);
    return 0;
}
```

Question: What do these predefined macros represent, and what will the program output?

Puzzle 10: Function-like Macros vs Inline Functions

```
#include <stdio.h>

#define MAX(a, b) ((a) > (b) ? (a) : (b))

inline int max(int a, int b) {
    return (a > b) ? a : b;
}

int main() {
    printf("Macro MAX: %d\n", MAX(5, 10));
    printf("Function max: %d\n", max(5, 10));
    return 0;
}
```

Question: What are the differences between the macro and the inline function? Are there any edge cases?