101 C, C++ and Linux Programming Problems

-DevNaga

Contents

The third argument is not given in the `printf` statement, but a third format specifier is provided. Printf would substitute it with a garbage data without evaulating the number of arguments and matching against the given number of format specifiers.

Printf.2

```c
#include <stdio.h>

int main()
{
    int a = 1;
    char *ptr = "string";
    double f = 1.2;

    printf("%s %f %d\n", ptr, f, a);

    return 0;
}
```

Problem. 9: Printf statement

Solution:

```
strng 1.200000 1
```

The printf format specifiers are properly used. The string is denoted with `%s`, the double denoted with `%f` and the integer with `%d`.

Printf.3

```c
#include <stdio.h>

int main()
{
    char *ptr = NULL;

    printf("%s\n", ptr);

    return 0;
}
```

Problem. 10: Printf statement

Solution:

Program crash.

The `printf` function derefs the character pointer given the `%s` format specifier. Thus resulting in a dereference of the `NULL` pointer resulting in the crash.

Variable argument print functions

Va_Arg.Printf.1

Implement `log_msg` function that accepts any type of arguments like the built-in `printf` function.

Solution:

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

static void log_msg(char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
}

int main()
{
    int a = 1;
    float f = 1.1;
    char *str = "string";
    double d = 1.211222222;

    log_msg("testing log msg with int %d float %f string %s and double %.8lf\n",
            a, f, str, d);
}
```

Problem. 9: va_args printf

The above program defines a `log_msg` function which uses `va_arg` macro definitions to create a `printf` like behavior.

`stdarg.h` header includes these macros.

`va_start` is used to collect the arguments. `vfprintf` prints the message to `stderr` stream. `vfprintf` behaves the same way as `printf` but allows the function to direct the messages to a stream. `va_end` performs the cleanup.

If statement

if.1

What would be the value of the following program?

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

```c
int main()
{
    int a = 1;

    if (a) {
        printf("a1\n");
    }

    a = 0;

    if (a) {
        printf("a2\n");
    }

    if (!a) {
        printf("a3\n");
    }

    if (!!a) {
        printf("a4\n");
    }

    if (a = !a) {
        printf("a5\n");
    }

    if (a == !!a) {
        printf("a6\n");
    }
}
```

Problem. 10: if condition checks

Solution:

```
a1
a3
a5
a6
```

`a1` will appear because a is set to 1. If condition checks non-zero statements.

`a3` will appear because a is set to 0 and !a is 1. `a5` will appear because if condition contains assignment `a = !a`. `a6` will appear because `a` is set to 1 before by the assignment `a = !a` and `!!a` results in double inversion of 1, i.e. 1.

if.else.2

What would be the solution of the following program?

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

int main()
{
    int a = 4;

    if (a != 4) {
        printf("a not 4\n");
    } else {
        printf("a is %d\n", a);
    }

    return 0;
}
```

Problem. If else case

Solution:

```
a is 4
```

if.elseif.3

What would be the solution of the following program?

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

int main()
{
    int a = 4;

    if (a != 4) {
        printf("a not 4\n");
    } else if (a == 4) {
        printf("a is 4\n");
    } else {
        printf("a not 4\n");
    }

    return 0;
}
```

Problem. If else if case

3. given the `struct r2` the largest elements are ordered in the beginning. This results in the 2 bytes hole in the `struct r2`.

Structs.4

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

struct s {
    int r;
};

int main() {
    // your code goes here
    struct s *s1;

    s1 = calloc(1, sizeof(struct s));
    if (!s1) {
        return -1;
    }

    s1->r = 1;

    printf("%d\n", s1->r);

    free(s1);

    s1 = calloc(10, sizeof(struct s));
    if (!s1) {
        return -1;
    }

    for (int i = 0; i < 10; i ++) {
        s1[i].r = i + 1;
    }

    for (int i = 0; i < 10; i ++) {
        printf("%d\n", s1[i].r);
    }

    free(s1);
    return 0;
}
```

Problem. 30: Structure allocation

Solution:

```
1
1
2
3
4
5
6
7
8
9
10
```

The statement `s1 = calloc(1, sizeof(struct s))` allocates structure pointer variable `s1` of type `struct s`. There is only one pointer.

The statement `s1 = calloc(10, sizeof(struct s))` allocates 10 such structure pointer variables `s1` which can be indexed like array. Another way to allocate the same above is `s1 = calloc(1, sizeof(struct s) * 10)`.

Now when accessed beyond the allocated size result in unknown behavior. Sometimes leading to guessed answer and sometimes results in a crash.

For example,

```
s1[12] = 12;
```

the above statement would result in an undefined behavior and may be result in a crash.

Structs.5

Is there anything wrong with the below program?

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

struct r {
    int a;
};

int main()
{
    struct r *r1 = NULL;
    struct r *r2 = NULL;

    r1 = calloc(1, sizeof(struct r));
    if (!r1) {
        return -1;
    }
```