**The C Language Basics - C Interview Questions and Answers:**

5. Can the last case of a *switch* statement skip including the break?

Even though the last case of a switch statement does not require a break statement at the end, you should add *break* statements to all cases of the *switch* statement, including the last case. You should do so primarily because your program has a strong chance of being maintained by someone other than you who might add cases but neglect to notice that the last case has no *break* statement.

This oversight would cause what would formerly be the last case statement to "fall through" to the new statements added to the bottom of the *switch* statement. Putting a *break* after each case statement would prevent this possible mishap and make your program more "bulletproof." Besides, most of today's optimizing compilers will optimize out the last break, so there will be no performance degradation if you add it.

6. Other than in a *for* statement, when is the comma operator used?

The comma operator is commonly used to separate variable declarations, function arguments, and expressions, as well as the elements of a for statement. Look closely at the following program, which shows some of the many ways a comma can be used:

#include <stdio.h>

#include <stdlib.h>

void main(void);

void main()

{

 /\* Here, the comma operator is used to separate

 three variable declarations. \*/

 int i, j, k;

 /\* Notice how you can use the comma operator to perform

 multiple initializations on the same line. \*/

 i = 0, j = 1, k = 2;

 printf("i = %d, j = %d, k = %d\n", i, j, k);

 /\* Here, the comma operator is used to execute three expressions

 in one line: assign k to i, increment j, and increment k.

 The value that i receives is always the rightmost expression. \*/

 i = (j++, k++);

 printf("i = %d, j = %d, k = %d\n", i, j, k);

 /\* Here, the while statement uses the comma operator to

 assign the value of i as well as test it. \*/

 while (i = (rand() % 100), i != 50)

 printf("i is %d, trying again...\n", i);

 printf("\nGuess what? i is 50!\n");

}

Notice the line that reads

*i = (j++, k++);*

This line actually performs three actions at once. These are the three actions, in order:

1. Assigns the value of *k* to *i*. This happens because the left value (*lvalue*) always evaluates to the rightmost argument. In this case, it evaluates to *k*. Notice that it does not evaluate to *k++*, because *k++* is a postfix incremental expression, and *k* is not incremented until the assignment of *k* to *i* is made. If the expression had read *++k*, the value of *++k* would be assigned to *i* because it is a prefix incremental expression, and it is incremented before the assignment is made.

2. Increments *j*.

3. Increments *k*.

Also, notice the strange-looking while statement:

while (i = (rand() % 100), i != 50)

 printf("i is %d, trying again...\n");

Here, the *comma* operator separates two expressions, each of which is evaluated for each iteration of the *while* statement. The first expression, to the left of the comma, assigns *i* to a random number from 0 to 99.

The second expression, which is more commonly found in a *while* statement, is a conditional expression that tests to see whether *i* is not equal to 50. For each iteration of the *while* statement, *i* is assigned a new random number, and the value of *i* is checked to see that it is not 50. Eventually, *i* is randomly assigned the value 50, and the *while* statement terminates.

7. How can you tell whether a loop ended prematurely?

Generally, loops are dependent on one or more variables. Your program can check those variables outside the loop to ensure that the loop executed properly. For instance, consider the following example:

#define REQUESTED\_BLOCKS 512

int x;

char\* cp[REQUESTED\_BLOCKS];

/\* Attempt (in vain, I must add...) to

 allocate 512 10KB blocks in memory. \*/

for (x=0; x< REQUESTED\_BLOCKS; x++)

{

 cp[x] = (char\*) malloc(10000, 1);

 if (cp[x] == (char\*) NULL)

 break;

}

/\* If x is less than REQUESTED\_BLOCKS,

 the loop has ended prematurely. \*/

if (x < REQUESTED\_BLOCKS)

 printf("Bummer! My loop ended prematurely!\n");

Notice that for the loop to execute successfully, it would have had to iterate through 512 times. Immediately following the loop, this condition is tested to see whether the loop ended prematurely. If the variable *x* is anything less than 512, some error has occurred.

8. What is the difference between *goto* and long *jmp( )* and *setjmp()*?

A *goto* statement implements a local jump of program execution, and the *longjmp()* and *setjmp()* functions implement a nonlocal, or far, jump of program execution. Generally, a jump in execution of any kind should be avoided because it is not considered good programming practice to use such statements as *goto* and *longjmp* in your program.

A *goto* statement simply bypasses code in your program and jumps to a predefined position. To use the *goto* statement, you give it a labeled position to jump to. This predefined position must be within the same function. You cannot implement gotos between functions. Here is an example of a goto statement:

void bad\_programmers\_function(void)

{

 int x;

 printf("Excuse me while I count to 5000...\n");

 x = 1;

 while (1)

 {

 printf("%d\n", x);

 if (x == 5000)

 goto all\_done;

 else

 x = x + 1;

 }

all\_done:

 printf("Whew! That wasn't so bad, was it?\n");

}

This example could have been written much better, avoiding the use of a *goto* statement. Here is an example of an improved implementation:

void better\_function(void)

{

 int x;

 printf("Excuse me while I count to 5000...\n");

 for (x=1; x<=5000; x++)

 printf("%d\n", x);

 printf("Whew! That wasn't so bad, was it?\n");

}

As previously mentioned, the *longjmp()* and *setjmp()* functions implement a nonlocal *goto*. When your program calls *setjmp()*, the current state of your program is saved in a structure of type *jmp\_buf*. Later, your program can call the *longjmp()* function to restore the program's state as it was when you called *setjmp()*. Unlike the *goto* statement, the *longjmp()* and *setjmp()* functions do not need to be implemented in the same function.

However, there is a major drawback to using these functions: your program, when restored to its previously saved state, will lose its references to any dynamically allocated memory between the *longjmp()* and the *setjmp()*. This means you will waste memory for every *malloc()* or *calloc()* you have implemented between your *longjmp()* and *setjmp()*, and your program will be horribly inefficient. It is highly recommended that you avoid using functions such as *longjmp()* and *setjmp()* because they, like the *goto* statement, are quite often an indication of poor programming practice.

Here is an example of the *longjmp()* and *setjmp()* functions:

#include <stdio.h>

#include <setjmp.h>

jmp\_buf saved\_state;

void main(void);

void call\_longjmp(void);

void main(void)

{

 int ret\_code;

 printf("The current state of the program is being saved...\n");

 ret\_code = setjmp(saved\_state);

 if (ret\_code == 1)

 {

 printf("The longjmp function has been called.\n");

 printf("The program's previous state has been restored.\n");

 exit(0);

 }

 printf("I am about to call longjmp and\n");

 printf("return to the previous program state...\n");

 call\_longjmp();

}

void call\_longjmp(void)

{

 longjmp(saved\_state, 1);

}