UST Global Interview Questions Papers

www.oureducation.in, blog.oureducation.in

Questions on C

What is C language?

C is a general-purpose computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the UNIX operating system.

What are the storage classes in C?

A storage class is an attribute that tells us where the variable would be stored, what will be the initial value of the variable if no value is assigned to that variable, life time of the variable and scope of the variable.

There are four storage classes in C:

1) Automatic storage class
2) Register storage class
3) Static storage class
4) External storage class

What is Pointer?

A pointer is a variable that holds a memory address. This address is the location of another object (typically, a variable) in memory. That is, if one variable contains the address of another variable, the first variable is said to point to the second.

What is dangling pointer?

Dangling pointers and wild pointers in computer programming are pointers that do not point to a valid object of the appropriate type.

Both the malloc() and the calloc() functions are used to allocate dynamic memory. Each operates slightly different from the other. malloc() takes a size and returns a pointer to a chunk of memory at least that big.

Is it better to use malloc() or calloc()?

void *malloc(size_t size);
calloc() takes a number of elements, and the size of each, and returns a pointer to a chunk of memory at least big enough to hold them all:
void *calloc(size_t numElements, size_t sizeOfElement);

UST Global Interview Questions Papers

What is meant by "bit masking"?

Bit masking means selecting only certain bits from byte(s) that might have many bits set. To examine some bits of a byte, the byte is bitwise "ANDed" with a mask that is a number consisting of only those bits of interest. For instance, to look at the one's digit (rightmost digit) of the variable flags, you bitwise AND it with a mask of one (the bitwise AND operator in C is &):

```
flags & 1;
```

To set the bits of interest, the number is bitwise "ORed" with the bit mask (the bitwise OR operator in C is |). For instance, you could set the one's digit of flags like so:

```
flags = flags | 1;
```

Or, equivalently, you could set it like this:

```
flags |= 1;
```

To clear the bits of interest, the number is bitwise ANDed with the one's complement of the bit mask. The "one's complement" of a number is the number with all its one bits changed to zeros and all its zero bits changed to ones. The one's complement operator in C is ~. For instance, you could clear the one's digit of flags like so:

```
flags = flags & ~1;
```

Or, equivalently, you could clear it like this:

```
flags &= ~1;
```

Sometimes it is easier to use macros to manipulate flag values.

Example Program : Macros that make manipulating flags easier.

```
/* Bit Masking */
/* Bit masking can be used to switch a character
   between lowercase and uppercase */
#define BIT_POS(N)          ( 1U << (N) )
#define SET_FLAG(N, F)      ( (N) |= (F) )
#define CLR_FLAG(N, F)      ( (N) &= -(F) )
#define TST_FLAG(N, F)      ( (N) & (F) )
```

```
#define BIT_RANGE(N, M)      ( BIT_POS((M)+1 - (N))-1 << (N) )
#define BIT_SHIFTL(B, N)     ( (unsigned)(B) << (N) )
#define BIT_SHIFTR(B, N)     ( (unsigned)(B) >> (N) )
#define SET_MFLAG(N, F, V)   ( CLR_FLAG(N, F), SET_FLAG(N, V) )
#define CLR_MFLAG(N, F)      ( (N) &= ~(F) )
#define GET_MFLAG(N, F)      ( (N) & (F) )
#include <stdio.h>
void main()
{
  unsigned char ascii_char = 'A';       /*  char = 8 bits only */
  int test_nbr = 10;
  printf("Starting character = %c\n", ascii_char);
  /*  The 5th bit position determines if the character is
     uppercase or lowercase.
     5th bit = 0  - Uppercase
     5th bit = 1  - Lowercase     */
  printf("\nTurn 5th bit on = %c\n", SET_FLAG(ascii_char, BIT_POS(5)) );
  printf("Turn 5th bit off = %c\n\n", CLR_FLAG(ascii_char, BIT_POS(5)) );
  printf("Look at shifting bits\n");
  printf("====================\n");
  printf("Current value = %d\n", test_nbr);
  printf("Shifting one position left = %d\n",
       test_nbr = BIT_SHIFTL(test_nbr, 1) );
  printf("Shifting two positions right = %d\n",
       BIT_SHIFTR(test_nbr, 2) );
}
```

BIT_POS(N) takes an integer N and returns a bit mask corresponding to that single bit position (BIT_POS(0) returns a bit mask for the one's digit, BIT_POS(1) returns a bit mask for the two's digit, and so on). So instead of writing

```
#define A_FLAG 4096

#define B_FLAG 8192
```

you can write

```
#define A_FLAG BIT_POS(12)

#define B_FLAG BIT_POS(13)
```

which is less prone to errors.

The SET_FLAG(N, F) macro sets the bit at position F of variable N. Its opposite is CLR_FLAG(N, F), which clears the bit at position F of variable N. Finally, TST_FLAG(N, F) can be used to test the value of the bit at position F of variable N, as in

```
if (TST_FLAG(flags, A_FLAG))
    /* do something */;
```

The macro BIT_RANGE(N, M) produces a bit mask corresponding to bit positions N through M, inclusive. With this macro, instead of writing

```
#define FIRST_OCTAL_DIGIT 7 /* 111 */

#define SECOND_OCTAL_DIGIT 56 /* 111000 */
```

you can write

```
#define FIRST_OCTAL_DIGIT BIT_RANGE(0, 2) /* 111 */

#define SECOND_OCTAL_DIGIT BIT_RANGE(3, 5) /* 111000 */
```

which more clearly indicates which bits are meant.

The macro BIT_SHIFT(B, N) can be used to shift value B into the proper bit range (starting with bit N). For instance, if you had a flag called C that could take on one of five possible colors, the colors might be defined like this:

```
#define C_FLAG        BIT_RANGE(8, 10)     /* 11100000000 */
/* here are all the values the C flag can take on */
#define C_BLACK       BIT_SHIFTL(0, 8)     /* 00000000000 */
#define C_RED         BIT_SHIFTL(1, 8)     /* 00100000000 */
#define C_GREEN       BIT_SHIFTL(2, 8)     /* 01000000000 */
#define C_BLUE        BIT_SHIFTL(3, 8)     /* 01100000000 */
#define C_WHITE       BIT_SHIFTL(4, 8)     /* 10000000000 */
#define C_ZERO        C_BLACK
#define C_LARGEST     C_WHITE
/* A truly paranoid programmer might do this */
#if C_LARGEST > C_FLAG
    Cause an error message. The flag C_FLAG is not
    big enough to hold all its possible values.
#endif /* C_LARGEST > C_FLAG */
```

The macro SET_MFLAG(N, F, V) sets flag F in variable N to the value V. The macro CLR_MFLAG(N, F) is identical to CLR_FLAG(N, F), except the name is changed so that all the operations on multibit flags have a similar naming convention. The macro GET_MFLAG(N, F) gets the value of flag F in variable N, so it can be tested, as in

```
if (GET_MFLAG(flags, C_FLAG) == C_BLUE)
    /* do something */;
```

What are multibyte characters ?

Multibyte characters are another way to make internationalized programs easier to write. Specifically, they help support languages such as Chinese and Japanese that could never fit into eight-bit characters. If your programs will never need to deal with any language but English, you don't need to know about multibyte characters.

Inconsiderate as it might seem, in a world full of people who might want to use your software, not everybody reads English. The good news is that there are standards for fitting the various special characters of European languages into an eight-bit character set. (The bad news is that there are several such standards, and they don't agree.)

Go to Asia, and the problem gets more complicated. Some languages, such as Japanese and Chinese, have more than 256 characters. Those will never fit into any eight-bit character set. (An eight-bit character can store a number between 0 and 255, so it can have only 256 different values.)

The good news is that the standard library has the beginnings of a solution to this problem. <stddef.h> defines a type, wchar_t, that is guaranteed to be long enough to store any character in any language a C program can deal with. Based on all the agreements so far, 16 bits is enough. That's often a short, but it's better to trust that the compiler vendor got wchar_t right than to get in trouble if the size of a short changes.

The mblen, mbtowc, and wctomb functions transform byte strings into multibyte characters. See your compiler manuals for more information on these functions.

Is it possible to execute code even after the program exits the main() function?

The standard C library provides a function named atexit() that can be used to perform "cleanup" operations when your program terminates. You can set up a set of functions you want to perform automatically when your program exits by passing function pointers to the atexit() function. Here's an example of a program that uses the atexit() function:

```
#include <stdio.h>
#include <stdlib.h>
void close_files(void);

void print_registration_message(void);
int main(int, char**);
int main(int argc, char** argv)
{
    ...
    atexit(print_registration_message);
    atexit(close_files);
    while (rec_count < max_records)
    {
        process_one_record();
    }
    exit(0);
}
```

This example program uses the atexit() function to signify that the close_files() function and the print_registration_message() function need to be called automatically when the program exits. When the main() function ends, these two functions will be called to close the files and print the registration message. There are two things that should be noted regarding the atexit() function. First, the functions you specify to execute at program termination must be declared as void functions that take no parameters. Second, the functions you designate with the atexit() function are stacked in the order in which they are called with atexit(), and therefore they are executed in a last-in, first-out (LIFO) method. Keep this information in mind when using the atexit() function. In the preceding example, the atexit() function is stacked as shown here:

```
atexit(print_registration_message);
atexit(close_files);
```

Because the LIFO method is used, the close_files() function will be called first, and then the print_registration_message() function will be called.

The atexit() function can come in handy when you want to ensure that certain functions (such as closing your program's data files) are performed before your program terminates.