

Q. 1) Why does `malloc(0)` return valid memory address? What's the use?

Answer: `malloc(0)` does not return a non-NULL under every implementation.

An implementation is free to behave in a manner it finds suitable, if the allocation size requested is zero. The implementation may choose any of the following actions:

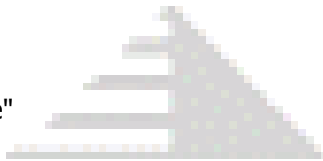
- * A null pointer is returned.
- * The behavior is same as if a space of non-zero size was requested. In this case, the usage of return value yields to undefined-behavior.

Q. 2) Why do we need to test weather it is memory leak or not? How are we going to know that?

Answer:

Possibilities are:

- 1) Array will print "Garbage Value"
- 2) Message by the Compiler!



Q. 3) How can you determine the size of an allocated portion of memory?

Answer: You can't, really. `free()` can, but there's no way for your program to know the trick `free()` uses. Even if you disassemble the library and discover the trick, there's no guarantee the trick won't change with the next release of the compiler.

Q. 4) What is the purpose of `realloc()`?

Answer: The function `realloc(ptr, n)` uses two arguments. The first argument `ptr` is a pointer to a block of memory for which the size is to be altered. The second argument `n` specifies the new size. The size may be increased or decreased. If `n` is greater than the old size and if sufficient space is not available subsequent to the old region, the function `realloc()` may create a new region and all the old data are moved to the new region.

Q. 5) What are advantages and disadvantages of external storage class?

Answer:

Advantages of external storage class:

- 1.) Persistent storage of a variable retains the latest value
- 2.) The value is globally available

Disadvantages of external storage class:

- 1.) The storage for an external variable exists even when the variable is not needed
- 2.) The side effect may produce surprising output
- 3.) Modification of the program is difficult
- 4.) Generality of a program is affected

Q. 6) What is the difference between `new/delete` and `malloc/free`?

Answer:

new =>Memory is allocated to a variable.

Delete =>Physically frees the memory

malloc =>Memory is allocated to a variable

free =>Logically frees the memory

Q. 7) Difference between `calloc()` and `malloc()`?

Answer: `malloc`: `malloc` create the single block of given size by user.

`calloc`: `calloc` creates multiple blocks of given size both return void pointer (`void *`) so both requires type casting.

`malloc`: eg:

```
int *p;  
p=(int*)malloc(sizeof(int)*5)
```

above syntax tells that `malloc` occupies the 10 bytes memory and assign the address of first byte to P.

`calloc`: eg:

```
p=(int*)calloc(5, sizeof(int)*5)
```

above syntax tells that `calloc` occupies 5 blocks each of the 10 bytes memory and assign the address of first byte of first block to P

Q. 8) Is it better to use `malloc()` or `calloc()`?

Answer: 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:

```
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 );
```

There's one major difference and one minor difference between the two functions. The major difference is that `malloc()` doesn't initialize the allocated memory. The first time `malloc()` gives you a particular chunk of memory, the memory might be full of zeros. If memory has been allocated, freed, and reallocated, it probably has whatever junk was left in it. That means, unfortunately, that a program might run in simple cases (when memory is never reallocated) but break when used harder (and when memory is reused). `calloc()` fills the allocated memory with all zero bits. That means that anything there you're going to use as a char or an int of any length, signed or unsigned, is guaranteed to be zero. Anything you're going to use as a pointer is set to all zero bits. That's usually a null pointer, but it's not guaranteed. Anything you're going to use as a float or double is set to all zero bits; that's a floating-point zero on some types of machines, but not on all.

The minor difference between the two is that `calloc()` returns an array of objects; `malloc()` returns one object. Some people use `calloc()` to make clear that they want an array.

Q. 9) What is the heap?

Answer: The heap is where `malloc()`, `calloc()`, and `realloc()` get memory.

Getting memory from the heap is much slower than getting it from the stack. On the other hand, the heap is much more flexible than the stack. Memory can be allocated at any time and deallocated in any order. Such memory isn't deallocated automatically; you have to call `free()`.

Recursive data structures are almost always implemented with memory from the heap. Strings often come from there too, especially

strings that could be very long at runtime. If you can keep data in a local variable (and allocate it from the stack), your code will run faster than if you put the data on the heap. Sometimes you

can use a better algorithm if you use the heap, faster, or more robust, or more flexible. It's a tradeoff.

If memory is allocated from the heap, it's available until the program ends. That's great if you remember to deallocate it when you are done. If you forget, it's a problem. A "memory leak" is some allocated memory that's no longer needed but isn't deallocated. If you have a memory leak inside a loop, you can use up all the memory on the heap and not be able to get any more. (When that happens, the allocation functions return a null pointer.) In some environments, if a program doesn't deallocate everything it allocated, memory stays unavailable even after the program ends.

Q. 10) What's wrong with this code? `Char*p *p=malloc(10);`

Answer: `malloc` function returns raw address in memory and for storing that address we need a pointer variable not value pointed by a pointer variable, because ordinary variables cannot store address.

So the right way out is,

```
void *p; p=(int*)malloc(n*sizeof(int));
```

Q. 11) How to write `calloc()` in terms of `malloc()`? ie. `malloc()` should initialise the memory to zero after allocating it.

Answer:

```
char *ptr = (char *)malloc(20);  
memset(ptr,0,20); //Sets 0 in memory which is pointed by ptr
```

Q. 12) How can you determine the size of an allocated portion of memory ?

Answer: You can't, really. `free()` can , but there's no way for your program to know the trick `free()` uses. Even if you disassemble the library and discover the trick, there's no guarantee the trick won't change with the next release of the compiler.

Q. 13) what is the difference between the functions `memmove()` and `memcpy()`?

Answer: `memcpy()`: It copies count characters from the array pointed from source to destination. If the array overlaps the behaviour of `memcpy()` is undefined.

`memmove()`:

It copies count characters from the array pointed from source to destination. If the array overlaps the copy will take place correctly.

```
Ex: char a[]="Hello World";
memmove(&a[2], "She", 3);
prints "HeShe world"
memcpy(&a[2], "llo", 3);
some times o/p is unexpected.
```

Q. 14) when should the volatile modifier be used?

Answer: The volatile modifier is a directive to the compiler's optimizer that operations involving this variable should not be optimized in certain ways. There are two special cases in which use of the volatile modifier is desirable. The first case involves memory-mapped hardware (a device such as a graphics adaptor that appears to the computer's hardware as if it were part of the computer's memory), and the second involves shared memory (memory used by two or more programs running simultaneously).

Most computers have a set of registers that can be accessed faster than the computer's main memory. A good compiler will perform a kind of optimization called "redundant load and store removal." The compiler looks for places in the code where it can either remove an instruction to load data from memory because the value is already in a register, or remove an instruction to store data to memory because the value can stay in a register until it is changed again anyway.

If a variable is a pointer to something other than normal memory, such as memory-mapped ports on a peripheral, redundant load and store optimizations might be detrimental. For instance, here's a piece of code that might be used to time some operation:

```
time_t time_addition(volatile const struct timer *t, int a)
{
int n;
int x;
time_t then;
x = 0;
then = t->value;
```

```
for (n = 0; n < 1000; n++)  
{  
x = x + a;  
}  
return t->value - then;  
}
```

Q. 15) when memory will be created after defining in c and c++;

Answer: when we execute program.(with the dynamically memory operator.)
when we compile the program.(as an stake to tempo.var.)

Q. 16) What is static memory allocation and dynamic memory allocation?

Answer: Static memory allocation: The compiler allocates the required memory space for a declared variable. By using the address of operator, The reserved address is obtained and this address may be assigned to a pointer variable. Since most of the declared variable have static memory, This way of assigning pointer value to a pointer variable is known as static memory allocation. memory is assigned during compilation time. Dynamic memory allocation: It uses functions such as `malloc()` or `calloc()` to get memory dynamically. If these functions are used to get memory dynamically and the values returned by these functions are assigned to pointer variables, such assignments are known as dynamic memory allocation. Memory is assigned during run time.

Q. 17) Difference between malloc and calloc?

Answer: malloc allocates a single block of storage space, where as calloc allocates multiple blocks of same size of storage space and initializes all to zero bytes,

syntax:

```
malloc=>ptr=(cast-type*)malloc(byte size);  
calloc=>ptr=(cast type*)calloc(n,element size);
```

Q. 18) Difference between calloc and malloc ?

Answer: malloc: allocate n bytes

calloc: allocate m times n bytes initialized to 0

Q. 19) What is the difference between calloc() and malloc() ?

Answer:

1.) calloc(...) allocates a block of memory for an array of elements of a certain size. By default the block is initialized to 0. The total number of memory allocated will be (number_of_elements * size). malloc(...) takes in only a single argument which is the memory required in bytes. malloc(...) allocated bytes of memory and not blocks of memory like calloc(...).

2.) malloc(...) allocates memory blocks and returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

calloc(...) allocates an array in memory with elements initialized to 0 and returns a pointer to the allocated space. calloc(...) calls malloc(...) in order to use the C++ _set_new_mode function to set the new handler mode.