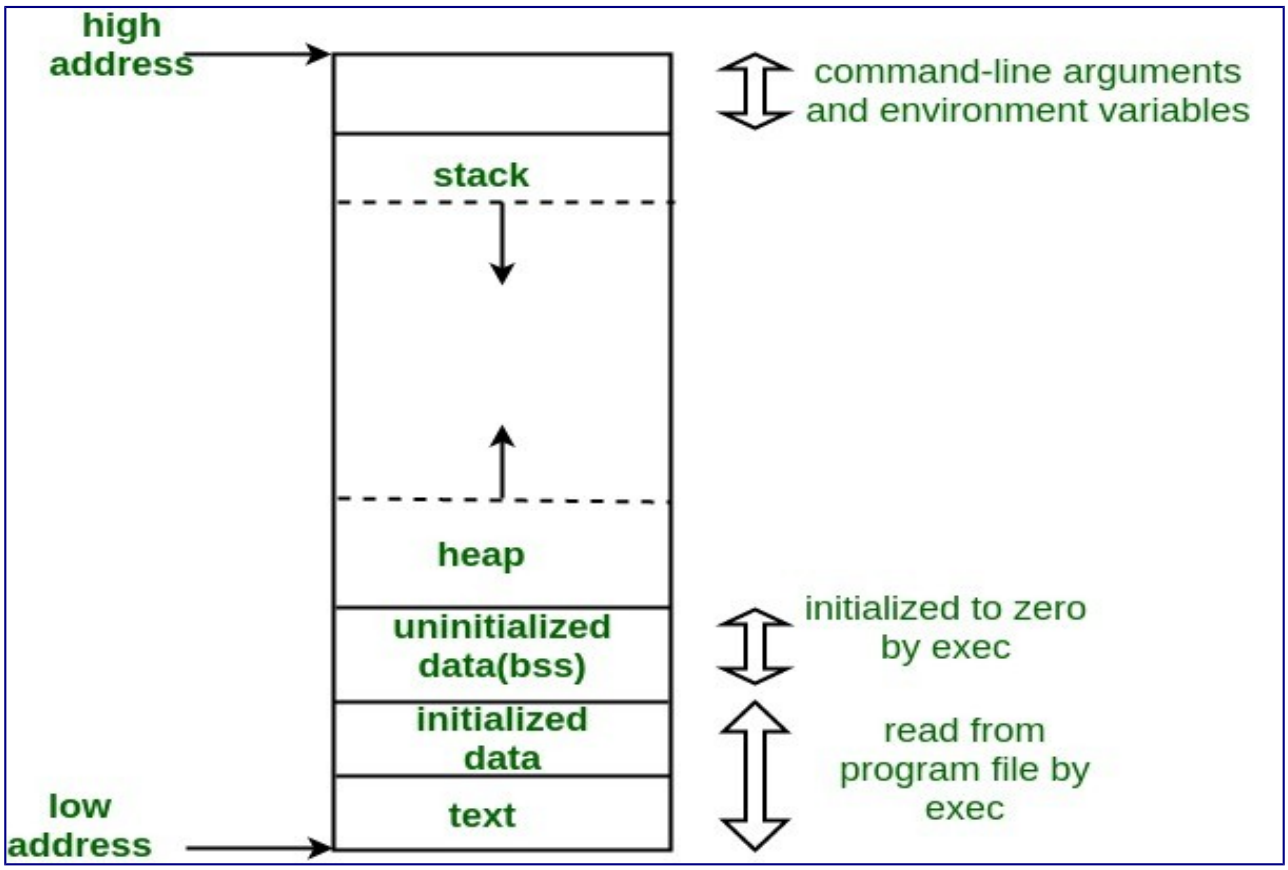
Memory Layout of C Programs

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



1. Text Segment:

A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized Data Segment:

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

3. Uninitialized Data Segment:

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.
For instance a global variable declared int j; would be contained in the BSS segment.

4. Stack:

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

5. Heap:

Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

How to deallocate memory without using free() in C?

void *realloc(void *ptr, size_t size);

If “size” is zero, then call to realloc is equivalent to “free(ptr)”. And if “ptr” is NULL and size is non-zero then call to realloc is equivalent to “malloc(size)”.

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

Malloc

“malloc” or “memory allocation” method is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

For Example:
ptr = (int*) malloc(100 * sizeof(int));

calloc()

“calloc” or “contiguous allocation” method is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

For Example:

ptr = (float*) calloc(25, sizeof(float));

free()

“free” method is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

free(ptr);

realloc()

“realloc” or “re-allocation” method is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory.

ptr = realloc(ptr, newSize);

The realloc function is used to resize the allocated block of the memory. It takes two arguments first one is a pointer to previously allocated memory and the second one is the newly requested size. The calloc function first deallocates the old object and allocates again with newly specified size. If the new size is lesser to the old size, the contents of the newly allocated memory will be same as prior but if any bytes in the newly created object goes beyond the old size, the values of the object will be indeterminate.

What is Memory Leak? How can we avoid?

Memory leak occurs when programmers create a memory in heap and forget to delete it.
Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

How does free() know the size of memory to be deallocated?

When memory allocation is done, the actual heap space allocated is one word larger than the requested memory. The extra word is used to store the size of the allocation and is later used by free( )

When requesting for memory allocation, the malloc(or calloc) function always allocates slightly more than you ask for, in order to store some bookkeeping information. This **bookkeeping information** is used to know how big the block is. when you call free().

calloc() versus malloc()

The name malloc and calloc() are library functions that allocate memory dynamically. It means that memory is allocated during runtime(execution of the program) from heap segment.

Initialization: malloc() allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block. malloc() doesn't initialize the allocated memory. If we try to acess the content of memory block then we'll get garbage values

calloc() allocates the memory and also initializes the allocates memory block to zero. If we try to access the content of these blocks then we'll get 0.

Number of arguments: Unlike malloc(), calloc() takes two arguments:
1) Number of blocks to be allocated.
2) Size of each block.

- **Return Value:** After successfull allocation in malloc() and calloc(), a pointer to the block of memory is returned otherwise **NULL** value is returned which indicates the failure of allocation.

We can achieve same functionality as calloc() by using malloc() followed by memset(),

```
ptr = malloc(size);
memset(ptr, 0, size);
```

Note: It would be better to use malloc over calloc, unless we want the zero-initialization because malloc is faster than calloc. So if we just want to copy some stuff or do something that doesn't require filling of the blocks with zeros, then malloc would be a better choice.

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

The calloc function initialize the allocated memory with 0 but malloc don't. So the memory which is allocated by the malloc have the garbage data. In another word you can say that calloc is equal to the combination of malloc and memeset.

See the below expression,

```
ptr = calloc(nmember, size); //is essentially equivalent to

ptr = malloc(nmember * size);
memset(ptr, 0, (nmember * size));
```

Note: If you don't want to initialize the allocated memory with zero, It would be better to use malloc over calloc.

What is memory management functions in C?

In C language, there are a lot of library functions (malloc, calloc, or realloc,..) which are used to allocate memory dynamically. One of the problems with dynamically allocated memory is that it is not destroyed by the compiler itself that means it is the responsibility of the user to deallocate the allocated memory.

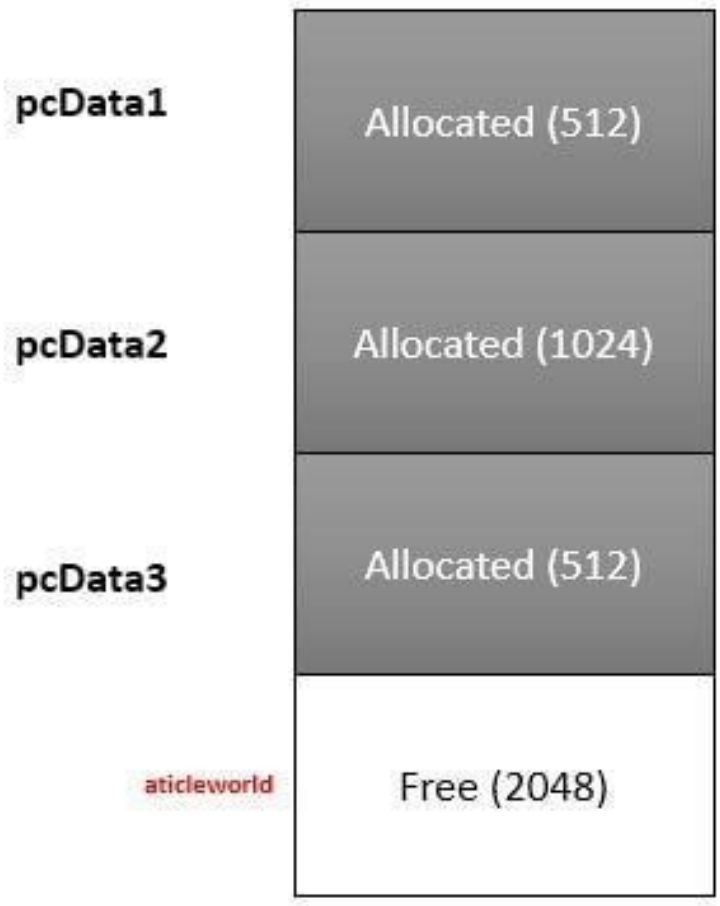
When we allocate the memory using the memory management function, they return a pointer to the allocated memory block and returned the pointer that points to the start address of the memory block. If there is no space available, the function will return a null pointer.

Some disadvantage of dynamic memory allocation in C

We have already discussed that compiler does not deallocate the dynamically allocated memory, the developer needs to clear the allocated memory. If the developer forgets to free the allocated memory, it can cause the memory leak and makes your program slow

The dynamic memory allocation can be the cause of the memory fragmentation.

Suppose heap had a capacity for 4K of memory. If the user consumes 2K of memory, the available memory would be 2K



When the user has deallocated the memory that is pointed by p2 than freed memory is available for the further use

Now, 3K of memory is available but contiguous memory is only 2k. If the user tries to allocate 3K of memory, the allocation would fail, even 3K of memory is free.

If you have allocated the memory dynamically, some extra bytes are wasted because it reserves a bookkeeping to put the information of the allocated memory. So dynamic memory allocation is beneficial when you need to allocate a large amount of memory.

There is one major problem with dynamic allocation, if you freed the memory before completed its task, then it can create hidden bug which is difficult to identify and can be a cause of the system crash or unpredictable value.

How to avoid memory leaks in C?

We can avoid the memory leak to perform the following tricks.

Create a counter to monitor allocated memory

It is a good technique to prevent the memory leaks. In this technique, we will create two global counters and initialize them with 0. In every successful allocation, we will increment the value of the counter1 (Allocate_Counter) and after the deallocating the memory we will increment the counter2 (Deallocate_Counter). At the end of the application, the value of both counters should be equal.

```
static unsigned int Allocate_Counter  = 0;

static unsigned int Deallocate_Counter  = 0;


void *Memory_Allocate (size_t size)
{
   void *pvHandle = NULL;

   pvHandle = malloc(size);
   if (NULL != pvHandle)
```

```
    {
        ++Allocate_Counter;
    }
    else
 {
 //Log error
 }
    return (pvHandle);
}


void Memory_Deallocate (void *pvHandle)
{
    if(pvHandle != NULL)
    {
        free(pvHandle);
        ++Deallocate_Counter;
    }
}


int Check_Memory_Leak(void)
{
 int iRet = 0;
    if (Allocate_Counter != Deallocate_Counter)
 {
 //Log error
 iRet = Memory_Leak_Exception;
 }
 else
 {
 iRet = OK;
 }
 return iRet;
}
```

Sometimes we have required allocated memory throughout the application, in that situation we have to write the free function after just writing the malloc in the handler that will invoke at the ending of the application.

For example,

Suppose there is a callback function DeactivateHandler() that is invoked at the end of the application, so we have to write the free function in DeactivateHandler() just after writing the malloc. These techniques reduce the probability to forget to free the memory.

Do not work on the original pointer

It is a good habit to work on a copy of thepointer, it preserves the address of allocating memory. If there is any accidental change occurred on the pointer, this technique helps you to get the actual address of allocating memory that is needed at the time of memory deallocation.

Avoid the orphaning memory location

At the time of memory deallocation, we need to free the memory from child to parent that means a child will be free first. If we free the parent first, it can be a cause of memory leak.

For example,

In below code, the pointer to context structure is freeing first. So the pointer that is pointing to space for the information data become orphan and it can be a cause of memory leak.

```
typedef struct
{
void *pvDataInfo;
}sContext;
 //Allocate the memory to pointer to context structure
sContext  *pvHandle = malloc(sizeof(sContext));
//Allocate the memory for Information data
pvHandle-> pvDataInfo  = malloc(SIZE_INFO_DATA);
free(pvHandle); // pvDataInfo  orphan
```

How can you determine the size of an allocated portion of memory?

Carry the length of allocated memory.

int *piArray = malloc (sizeof(int) * (n+1));

* piArray = n;

int * pTmpArray = piArray +1;

Now, whenever in a program you ever required the size of the array then you can get from copy pointer.

ArraySize = pTmpArray[-1];

What is static memory allocation and dynamic memory allocation?

According to C standard, there are four storage duration, static, thread (C11), automatic, and allocated. The storage duration determines the lifetime of the object.

The static memory allocation:

Static Allocation means, an object has external or internal linkage or declared with static storage-class. It's initialized only once, prior to program startup and its lifetime is throughout the execution of the program. A global and static variable is an example of static memory allocation.

The dynamic memory allocation:

In C language, there are a lot of library functions (malloc, calloc, or realloc,..) which are used to allocate memory dynamically. One of the problems with dynamically allocated memory is that it is not destroyed by the compiler itself that means it is the responsibility of the user to deallocate the allocated memory.

When we allocate the memory using the memory management function, they return a pointer to the allocated memory block and the returned pointer is pointing to the beginning address of the memory block. If there is no space available, these functions return a null pointer.

What is the return value of malloc (0)?

If the size of the requested space is zero, the behavior will be implementation-defined. The return value of the malloc could be a null pointer or it shows the behavior like that size is some nonzero value.It is suggested by the standard to not use the pointer to access an object that is returned by the malloc while size is zero.

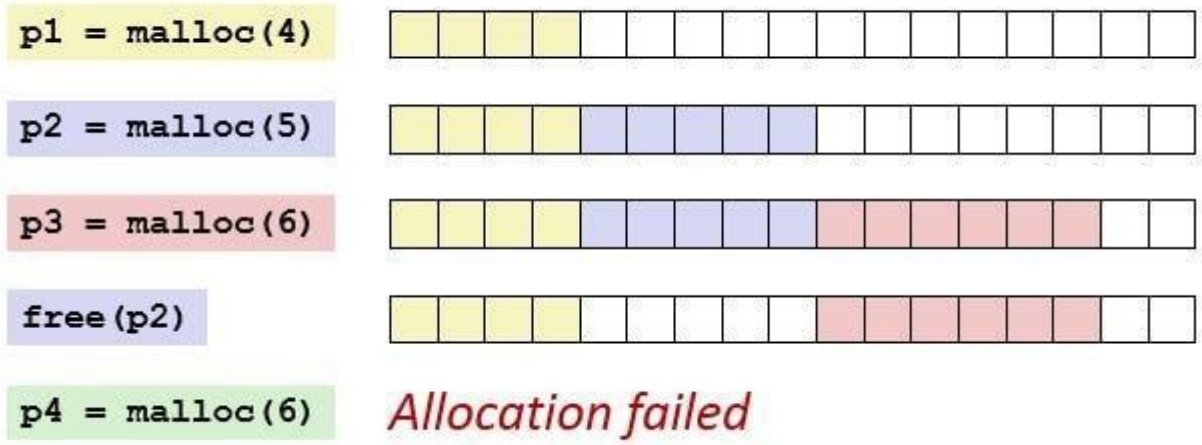
What is dynamic memory fragmentation?

The memory management function is guaranteed that if memory is allocated, then it would be suitably aligned to any object which has the fundamental alignment. The fundamental alignment is less than or equal to the largest alignment that's supported by the implementation without an alignment specification.

One of the major problems with dynamic memory allocation is fragmentation, basically, fragmentation occurred when the user does not use the memory efficiently. There are two types of fragmentation, external fragmentation, and internal fragmentation.

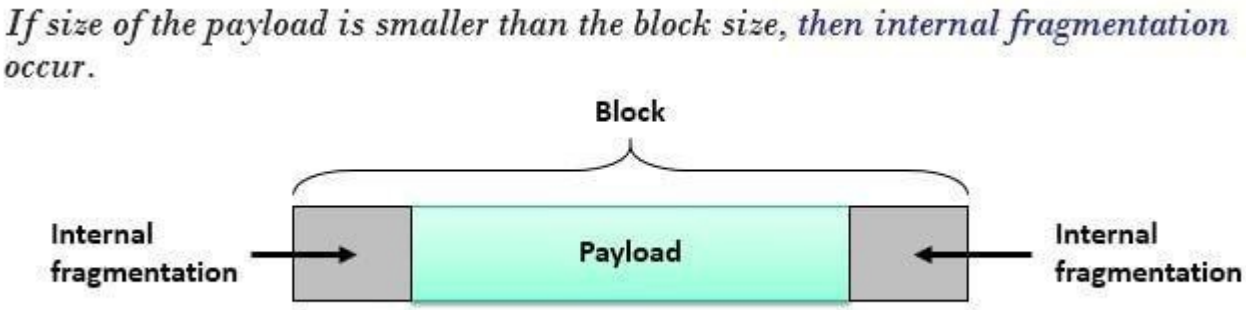
The external fragmentation is due to the small free blocks of memory (small memory hole) that is available on the free list but program not able to use it. There are different types of free list allocation algorithms that used the free memory block efficiently.

Consider a scenario where a program has 3 contiguous blocks of memory and the user frees the middle block of memory. In that scenario, you will not get a memory, if the required block of is larger than a single block of memory (but smaller or equal to the aggregate of the block of memory)



The internal fragmentation is the wasted of memory that is allocated for rounding up the allocated memory and in bookkeeping (infrastructure), the bookkeeping is used to keep the information of the allocated memory.

Whenever we called the malloc function then it reserves some extra byte (depend on implementation and system) for bookkeeping. This extra byte is reserved for each call of malloc and become a cause of the internal fragmentation.



For example,

See the below code, the programmer may think that system will be allocated 8 *100 (800) bytes of memory but due to bookkeeping (if 8 bytes) system will be allocated 8*100 extra bytes. This is an internal fragmentation, where 50% of the heap waste.

```
char *acBuffer[100];
int main()
{
  int iLoop = 0;
  while(iLoop < 100)
  {
   acBuffer[iLoop ] =  malloc(8);
   ++iLoop
  }
}
```

Common mistakes with memory allocation, you should avoid

Forget to check return value of malloc

It is a very common mistake and can be the cause of the segmentation fault. When we call the malloc (memory management function) then it returns the pointer to allocated memory. If there is no free space is available, the malloc function returns the NULL. It is good habits to verify the allocated memory because it can be NULL. You already know that if we try to dereference the null pointer, we will get the segmentation fault.

Initialization errors

Generally, c programmer uses malloc to allocate the block of memory. Some programmers assume that malloc allocated memory is initialized by the zero and they use the block of memory without any initialization. In some scenario, it does not reflect the bad effect but sometimes it creates hidden issues.

```
int * Foo(int *x, int n)
{
 int *piBuffer = NULL;
 int i = 0;
 //creating an integer array of size n.
```

```
piBuffer = malloc(n * sizeof(int));
//make sure piBuffer is valid or not
if (piBuffer == NULL)
{
// allocation failed, exit from the program
fprintf(stderr, "Out of memory!\n");
exit(1);
}
//Add the value of the arrays
for (i = 0; i < n; ++i)
{
piBuffer[i] = piBuffer[i] + x[i];
}
```

Freeing the same memory multiple times

A free function is used to deallocate the allocated memory. If piData (arguments of free) is pointing to a memory that has been deallocated (using the free or realloc function), the behavior of free function would be undefined.

The freeing the memory twice is more dangerous then memory leak, so it is very good habits to assigned the NULL to the deallocated pointer because the free function does not perform anything with the null pointer.

Freeing memory that was not allocated

The free function only deallocates the allocated memory. If piData is not pointing to a memory that is allocated by the memory management function, the behavior of the free function will be undefined.

Re-assignment of pointer

Sometimes reassignment of the pointer creates the problems.If you do not use the dynamically allocated memory properly (in the situation of shallow copy), it can cause of the code crashing or unwanted result.

Dereferencing a pointer without allocating some memory

When you will try to access a pointer without giving a proper memory, you will get the undefined result. Many new developers access the pointers without allocating a memory and frustrated with coming results. A pointer with not valid memory is called dangling pointers

```
#include<stdio.h>

int main()
{
int *piData;

*piData = 10; //piData is dangling pointer

return 0;

}
```

Stack vs Heap Memory Allocation

Stack Allocation : The allocation happens on contiguous blocks of memory. We call it stack memory allocation because the allocation happens in function call stack. The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated. This all happens using some predefined routines in compiler. Programmer does not have to worry about memory allocation and deallocation of stack variables.

Heap Allocation : The memory is allocated during execution of instructions written by programmers. Note that the name heap has nothing to do with heap data structure. It is called heap because it is a pile of memory space available to

programmers to allocated and de-allocate. If a programmer does not handle this memory well, memory leak can happen in the program.

Key Differences Between Stack and Heap Allocations

1. In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
2. Handling of Heap frame is costlier than handling of stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
4. Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
5. Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
6. Accessing time of heap takes is more than a stack.

How does a cache memory differ from registers?

CACHE:

It's a temporary storage. It resides within the processor chip. It's both very fast as well as nearer to cpu than ram. The **main aim is** to try to fill it with the data which might be needed again soon. Hence it speeds up the computations if next time cpu finds the required data in cache itself (& thus no need to search & fetch data from slower ram).

There are various techniques to efficiently use the limited cache memory like Least Recently Used (LRU), etc. These techniques tries to predict which data might be needed again & hence should be stored in cache.

It's of generally three types/levels, L1, L2 & L3. L1 is the fastest but smallest. L3 is the largest but slowest.

REGISTERS:

They are small memories within the cpu. They are nearest to the cpu. There are many types of registers like Accumulator, Data Register, Program Counter, General purpose, etc. Thus they can be used for various tasks.

Take example of accumulator register, its **aim is** to hold partial results and calculations. For example, if you want to multiply 789653 with 23442 i.e. (789653*23442), then you must store intermediate results and calculations before the final summation. This is the specific task of accumulator register.

Take another example of Program Counter, it stores the address of the next instruction to be fetched.

Thus there are various types of registers, some have specific tasks while others can be used for any purpose.

Summary of Similarities:

Both resides within the processor chip.

Both are faster than any other memory.

Both are divided in various types/levels.

Summary of Differences:

Registers are both faster as well as nearest to cpu than cache.

Registers are crucial for the cpu, without registers cpu will not perform in feasible amount of time.

Cache can be seen as faster ram, which can help only if the same data is needed again (& again) or if we could predict which data will be needed next (or soon)