Dynamic Memory Management

So far, we have used variables which are created when the block is entered and which are removed when the block is exited. We have no control over the creation and destruction of variables.

Is it possible to get memory whenever we require? Is it possible to relinquish memory when we no more require? Can we get memory which is not bound to any particular type and then we can create variables of a particular type within that memory?

It is possible. We use a technique called dynamic memory management to achieve this.

Before we get into this topic, let us revise the concept of a variable.

A variable in a block is created on entry and is not initialized by default. A variable uninitialized or otherwise has a location – therefore an address.

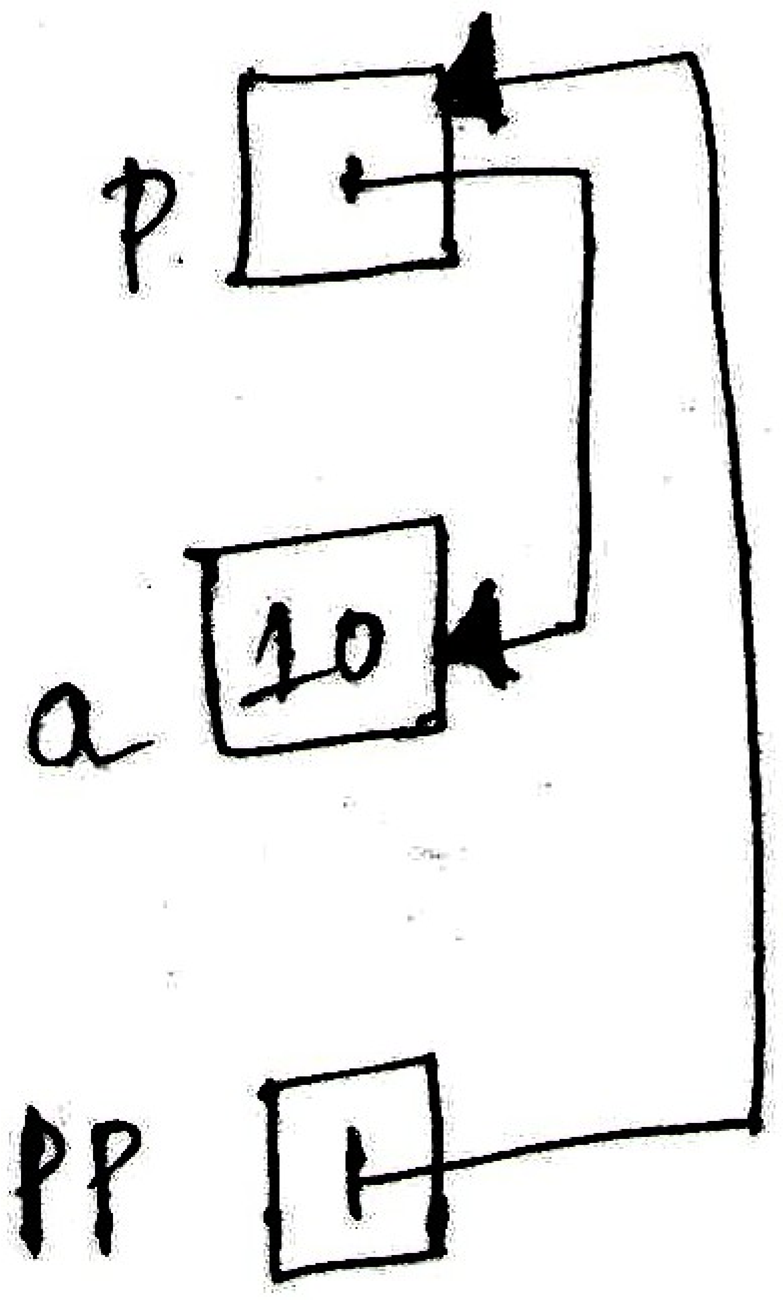
If a variable is of pointer type, we should initialize the variable to point to a variable – dereferencing before initialization is clearly an undefined behaviour.

Observe the program given below.

```
// 1_ex.c
#include <stdio.h>
int main()
{
#if 0
	int *p;
	printf("&p : %p\n", &p); // ok
	printf("p : %p\n", p); // logically wrong; using an uninitialized variable
	// program will not crash
	printf("*p : %d\n", *p); // terrible code; looking content of an uninitialized
variable
	// as a  pointer and dereferencing it; could crash; undefined behaviour
#endif
	int *p;
	int a = 10;
	p = &a;
	printf("&p : %p\n", &p); // ok
	printf("p : %p\n", p);  // ok; p has been assigned a value
	printf("*p : %d\n", *p); // same as a; ok if a has been given a value

	int **pp;
	//pp = & &a; //1_ex.c:21:7: error: lvalue required as unary ‘&’ operand
	pp = &p;
	printf("&pp : %p\n", &pp);
	printf("pp : %p\n", pp);
	printf("*pp : %p\n", *pp);
	printf("**pp : %d\n", **pp);

}
```



Also, observe that if a is a variable, &a is a pointer to it and it is a r-value. & & a is a guaranteed compile time error as &a is not a lvalue – cannot appear to the left of assignment.

The variable pp is a pointer to a pointer by type – can point to a variable which can point to an int.

Clearly understand what these expressions stand for.

&pp : pointer to a variable whose type is pointer to pointer to int
pp : variable whose type is pointer to pointer
*pp : expression(both l and r value) which points to an int
**pp : expression (both l and r value) which is an int

There are no operators to support dynamic memory management in 'C'. There are functions using which we can effect dynamic memory management.
This logical memory area used by dynamic memory manager is called heap.

malloc : allocates memory of a given size and returns a pointer
free : returns the memory allocated back to the memory manager
calloc : allocates memory and initializes the memory with 0
realloc : reallocates memory; copies the old values into the new memory locations.
Please check the man pages for these functions.

```
// 2_ex.c
#include <stdio.h>
// dynamic memory management : malloc and free allocate and deallocate memory on the heap
int main()
{
#if 0
	int *p;
	// request for memory
	//	function : malloc requires the # of bytes to be passed as argument
	// returns a generic pointer : void*
	// convert pointer to void to pointer to int
	// creates a variable with no name; can be accessed using a pointer
	p = (int*)malloc(sizeof(int));
	*p = 20;
	// return the location back to runtime : free
	free(p);
#endif
#if 0
	int *p;
	p = (int *)malloc(sizeof(int) * 4); // array on the heap
	p[0] = 11; *(p + 1) = 22;
	free(p);
// on allocation of memory, malloc stores the size of memory allocated in some location
// That is based on implementation.
// Given the pointer value returned by malloc, it is possible for the implementation
//	to find the size allocated
// the function free finds the book keeping or house keeping info and deallcoates the
//	required amount of memory
#endif

	int *p;
	p = (int *)malloc(sizeof(int) * 4);

	free(p + 1); // TERRIBLE thing to do
	// p + 1 was not returned by malloc
	// there is no meaningful book keeping associated with p + 1
	// anything may happen
	// UNDEFINED BEHAVIOUR
}
```

Let us examine the program 2_ex.c.

int *p;

At this point, we have the variable. The variable p has not been initialized. Dereferencing p at this point is criminal.

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

We ask malloc to give us as many bytes as an int requires. Observe we do not hard code the # of bytes an int occupies as it depends on implementation.

The function malloc returns a generic pointer – void*. We convert that to pointer to int by casting.

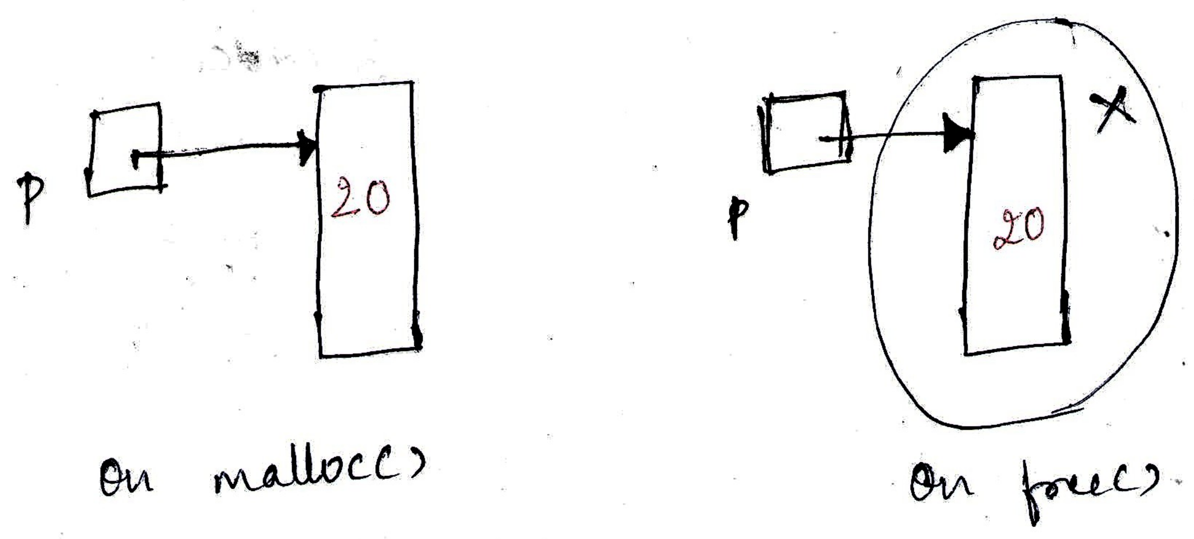
*p = 20;

We assign 20 to the int to which p points. Observe that, this variable does not have a name.

// return the location back to runtime : free

free(p);

Gives the location back to the runtime management. Observe as the parameter passing is by value, the value of p cannot change when free is called. It will still have the old value.



Let us consider another example of memory management.

int *p;

p = (int *)malloc(sizeof(int) * 4); // array on the heap

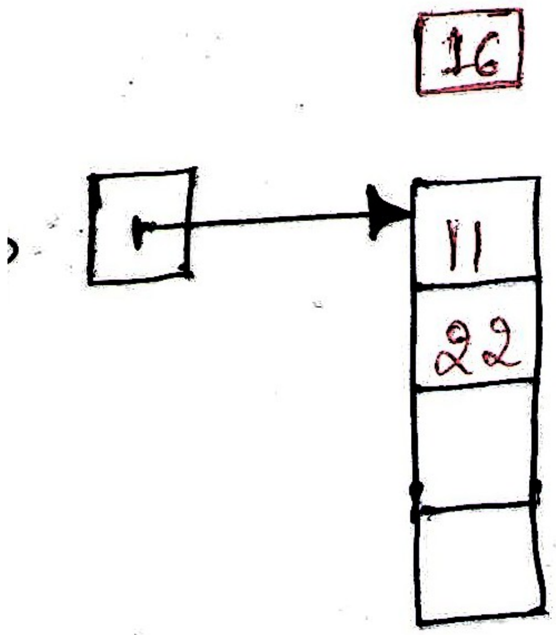
The above statement causes allocation of memory for 4 integers. Once allocated, p points to an array of 4 integers. We say that p is an array on the heap.

p[0] = 11; *(p + 1) = 22;

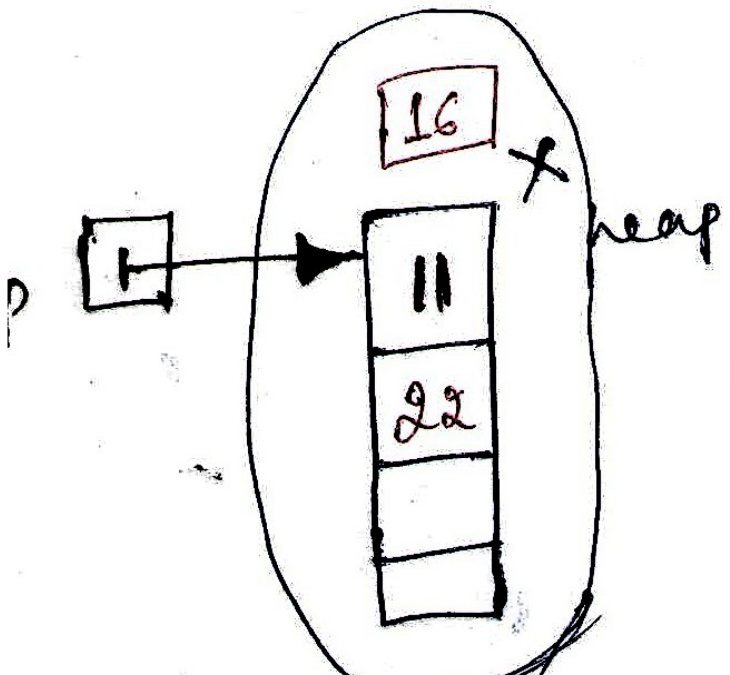
free(p);

The function free releases the whole block.

// Book keeping / hows



On allocation of memory

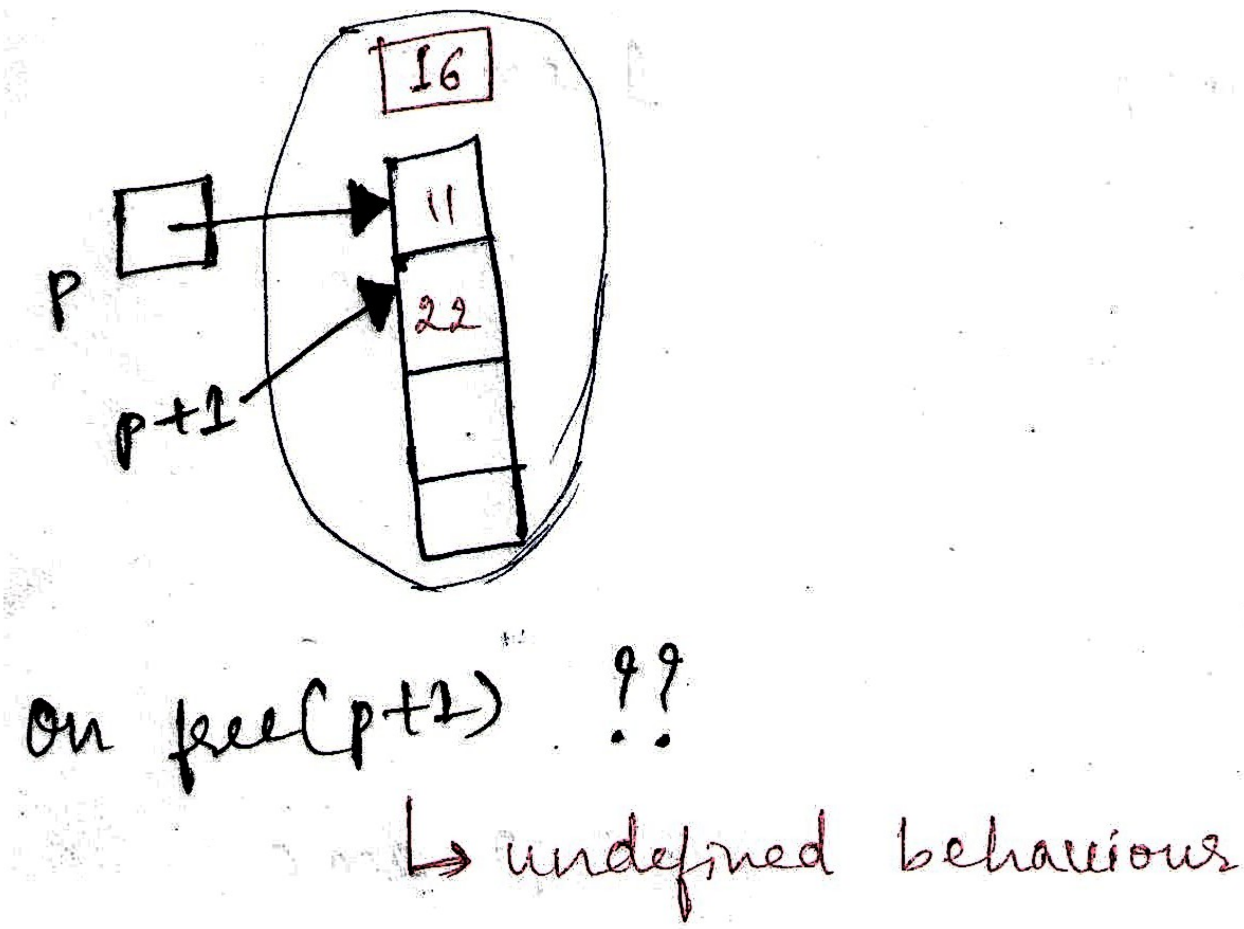


On free

You may wonder why free is not given an additional argument indicate the size of the block allocated. You may wonder how come free is able to free a single int location in the first case or an array of int in the second case.

These is an interesting concept called book keeping or house keeping. When malloc allocates memory, it remembers somewhere in memory the # of bytes allocated. This additional information maintained is called book keeping or house keeping. The function malloc returns a pointer which can be used to access the location where the # of bytes is stored. We do not know where the book keeping is kept, but the implementation knows it. When we call free with a pointer, free accesses the book keeping and finds the # of bytes to deallocate and deallocates so much of memory.

What happens if free is called on a pointer which does not point to heap?
What happens if free is called on a pointer value not returned by a previous malloc call?
What happens if free is called second time?
All result in undefined behaviour.
We can delete locations only on the heap. The pointer should be bound to a location where book keeping is kept. Double deletion is an undefined behaviour.



```
int *p;
p = (int *)malloc(sizeof(int) * 4);

free(p + 1); // TERRIBLE thing to do; no booking keeping associated with p + 1.
```

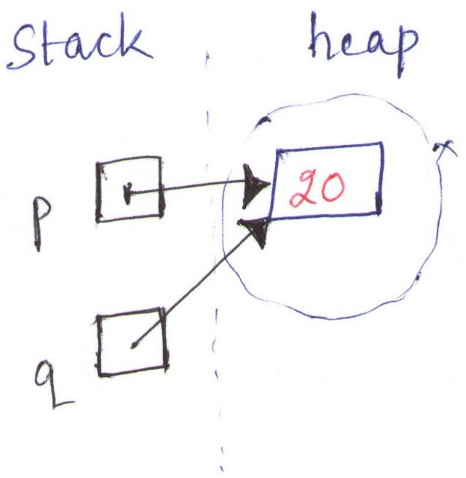
Let us now examine a couple of issues with dynamic memory management. The program is 3_ex.c.

```
        int *p;
        p = (int*)malloc(sizeof(int));
        *p = 20;
        int *q = p;
```

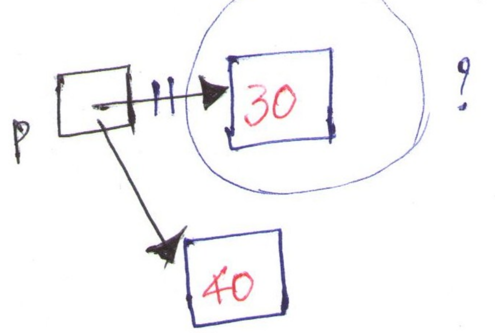
In the

free(q); // valid statement; ok; what matters is the value of the pointer and not the name

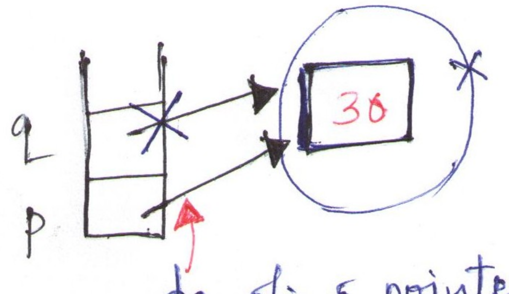
Here, we allocate memory to p, initialize q to get a copy of p. So both p and q point to the same location – they are aliases. Can we call free(q) safely. The answer is YES. What matters is the value of the pointer and not the name. The value of the pointer is associated with book keeping. So, free can find the size allocated through the value of the pointer and release so much of memory.



on free(q)



garbage ; location with no access



dangling pointer ; no location ; access ;

```
// 3_ex.c
#include <stdio.h>
int main()
{
#if 0
	int *p;
	p = (int*)malloc(sizeof(int));
	*p = 20;
	int *q = p;
	free(q); // valid statement; ok; what matters is the value of the pointer
and not the name
	// ???
#endif

#if 0
	int *p;
	p = (int*)malloc(sizeof(int));
	*p = 30;
	p = (int*)malloc(sizeof(int));
	// changing of p makes us loose the pointer to the location allocated by
the earlier
	//	malloc call

	// creates garbage; we have a location but no access
	*p = 40;
	// no garbage collector (python does based on reference count)
	// garbage in turn becomes a memory leak

#endif
	int *p;
	{
		p = (int*)malloc(sizeof(int));
		*p = 30;
		int *q = p;
		free(q); // cannot change q; parameter passing by value
		// q = NULL: or q = 0; // some people think it is safe programming
		// dereferencing a NULL pointer is a guaranteed crash
	}
	// p is a dangling pointer; no location; access;
	printf("*p : %d\n", *p);
}
```

Garbage and Memory Leak:

```
p = (int*)malloc(sizeof(int));
*p = 30;
p = (int*)malloc(sizeof(int));
```

Observe the code above. We allocate memory and p points to it. The second allocation will overwrite p. We loose access to the first allocated memory as it has no name and only access was through p, but p has changed now.

This integer location – first allocated memory - cannot be accessed. It is called garbage. Garbage is a case where a location exists, but there is no access to it.

In some languages, there are mechanisms to make the garbage reuseful – this is called garbage collection. In Python, we have garbage collection based on reference count.

‘C’ does not support garbage collection as it is costly in terms of space and time. The onus of keeping memory clean is on the programmer and not the ‘C’ runtime. So, garbage in ‘C’ in turn becomes memory leak.

Dangling pointer:

```
int *p;
p = (int*)malloc(sizeof(int));
*p = 30;
int *q = p;
free(q);
printf("*p : %d\n", *p); // ????
```

In the example above, p gets allocation through malloc. The unnamed location pointed to by p is assigned. Both q and p point to the same location. Then q is freed. Now p still points to the location which no more logically exists. This is a case where we have no location, but we have access.

Both p and q are dangling pointers – access but no location.

Please note the following.

Dangling pointers are not dangerous – dereferencing them is dangerous.

Dangling pointers can happen anywhere, but memory leak can only happen in the heap.

Allocation with initialization:

The function calloc allocates memory and initializes the locations with 0. Observe that calloc takes two arguments where malloc takes 1. The calloc does not give us any choice regarding initialization – it always each byte to 0.

```
// 4_ex.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
	int *p;
	//p = (int *)malloc(sizeof(int) * 4);
	p = (int *)calloc(4, sizeof(int));
	printf("p : %p\n", p);
	for(int i = 0; i < 4; ++i)
	{
		printf("%d ", p[i]);
	}
	printf("\n");

	// change the array size?
	// increase the size:
	//		may get the same sequence of locations
	//				different sequence of locations
	//		old pointer is conceptually dangling
	//		values are copied; old location freed if we get a new block of locations
	// decrease the size
	//		values at locations which remain are kept
	int *q = (int *)realloc(p, 10000);
	printf("p : %p q : %p\n", p, q);

	free(q);
}
```

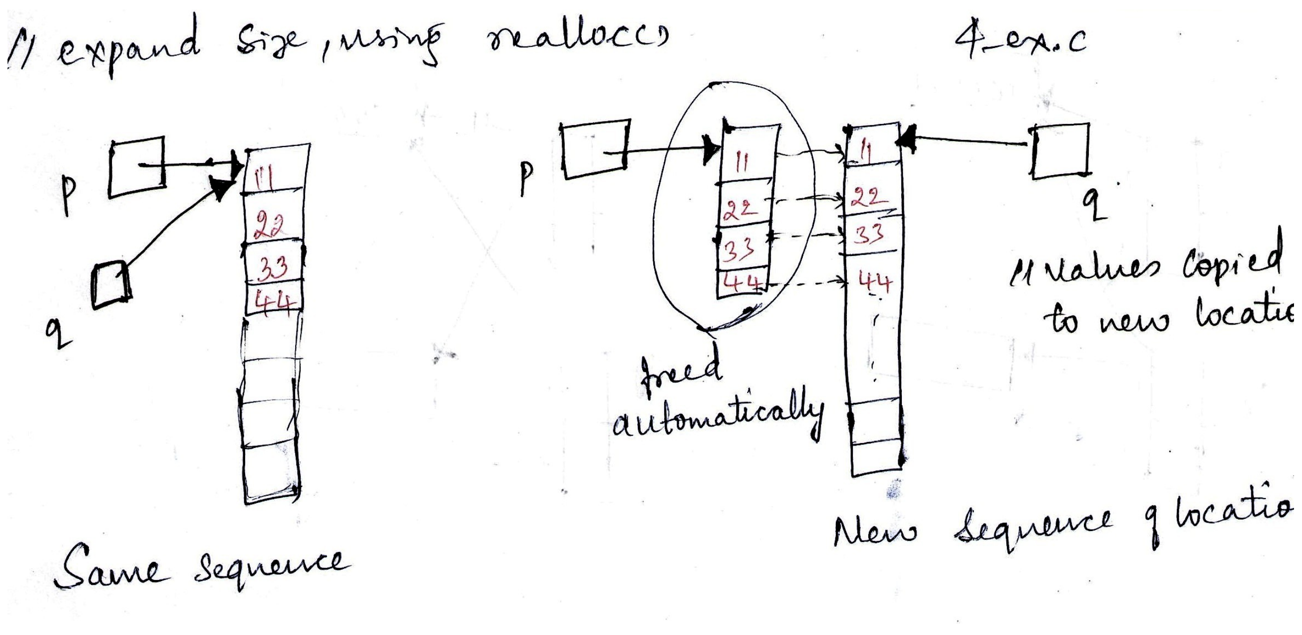
Reallocation:

```
int *q = (int *)realloc(p, 10000);
```

If p points to a block of locations on the heap with some size, the size can be altered by reallocation.

A few points to note:

- size can be increased or decreased
- the pointer returned by realloc may not be same as the earlier value.
  So, allocation may happen in some other part of memory
- on increase in size, all old values are copied and the rest of the locations remain uninitialized.
- On decrease in size, only the values of those locations which find new locations get copied. The rest are lost.



Allocation of structure:

```
// 5_ex.c
#include <stdio.h>
#include <stdlib.h>

struct What
{
        int *p;
        int a;
};
typedef struct What what_t;
int main()
{
// case 1:
#if 0
        what_t x;
        what_t y;
        int z = 10000;
```

```
	x.a = 100;
	x.p = &z;

	printf("x : %d %d\n", x.a, *(x.p)); // 100 10000
	y = x;
	printf("y : %d %d\n", y.a, *(y.p)); // 100 10000
	x.a = 200; *(x.p) = 20000;
	printf("y : %d %d\n", y.a, *(y.p)); // 100 20000
#endif
// case 2:
#if 0
	what_t x;
	what_t y;

	x.a = 100;
	x.p = (int *)malloc(sizeof(int));
	*(x.p) = 10000;

	y = x;
	printf("y : %d %d\n", y.a, *(y.p));
	free(y.p);
	// x.p : ?  dangling
#endif
// case 3:
#if 0
	what_t x;
	what_t y;
	x.a = 100;
	x.p = (int *)malloc(sizeof(int));
	*(x.p) = 10000;

	y = x;

	y.p = (int *)malloc(sizeof(int)); // ?? safe
```

```
#endif
// case 4:
#if 0
        what_t x;
        what_t y;
        x.a = 100;
        x.p = (int *)malloc(sizeof(int));
        *(x.p) = 10000;

        y = x;

        x.p = (int *)malloc(sizeof(int));  // ok
#endif
// case 5:
        what_t x;
        x.a = 100;
        x.p = (int *)malloc(sizeof(int));
        *(x.p) = 10000;
        x.p = NULL;
}
```

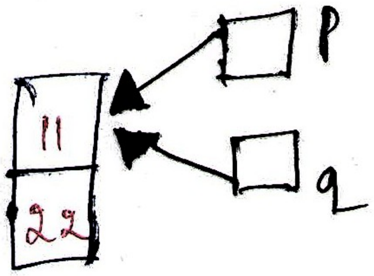
In case 1, observe that the assignment of structures causes the pointer fields to become aliases.
x.p and y.p both point to z. Changing z through any one of these pointers affects the other.

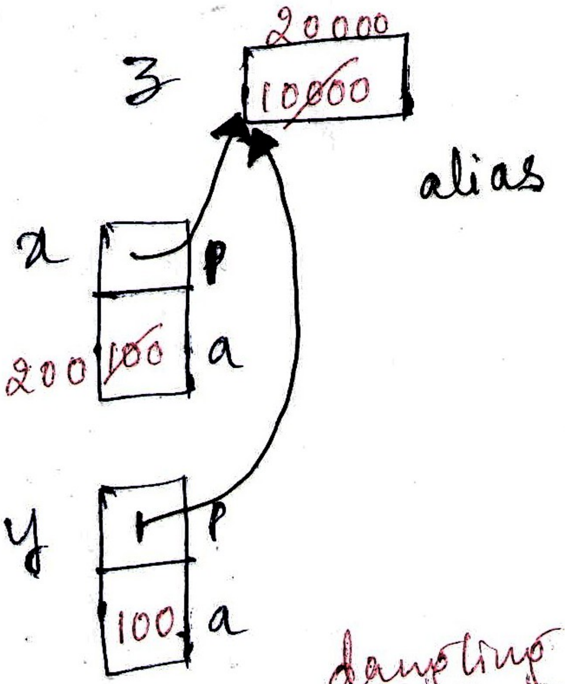
In case 2, the pointers in both x and y point to a location on the heap. Freeing through one of these pointers, make all these pointers dangling.

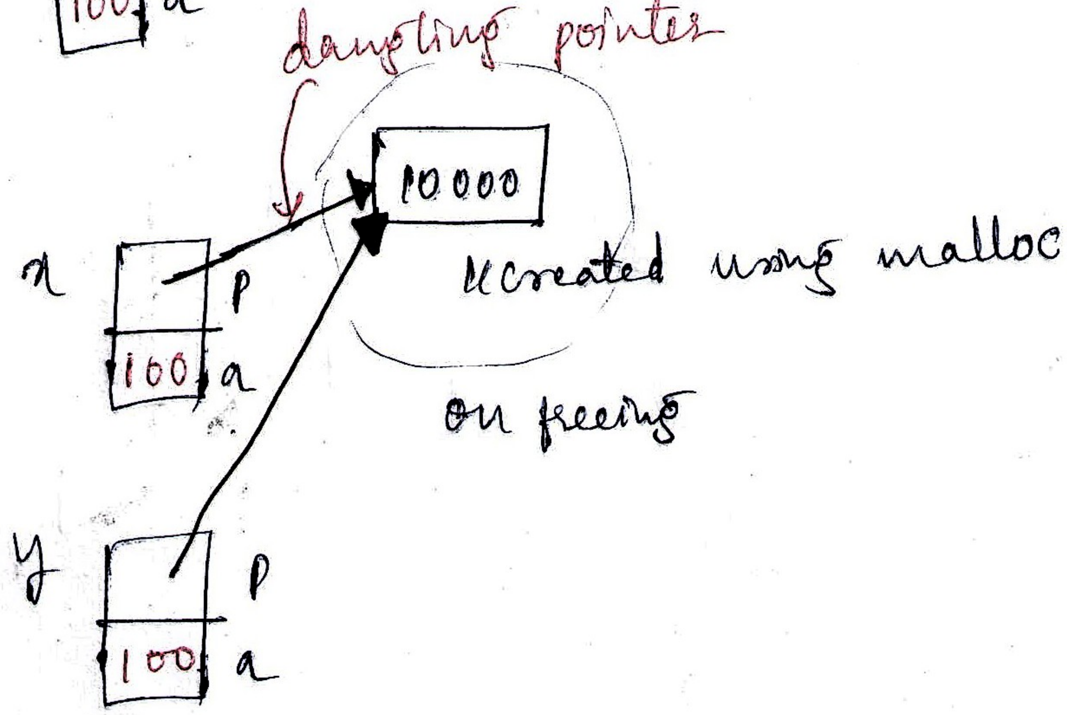
In case 3, after the assignment of x to y, both the pointers point to the same location on the heap. When y.p is assigned a new pointer value, the aliasing is broken. Both x.p and y.p point to two different locations all together – perfectly safe.

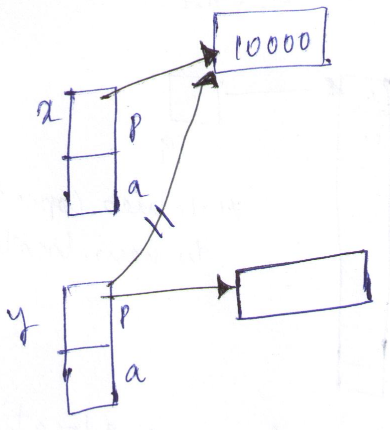
In case 4, after assignment of x to y, both the pointers point to the same location on the heap. When x.p is assigned a new pointer value, the aliasing is broken. Both x.p and y.p point to two different locations all together – perfectly safe.

In case 5, the pointer is grounded resulting in loss of access to location on the heap – causing garbage and therefore memory leak in ‘C’.

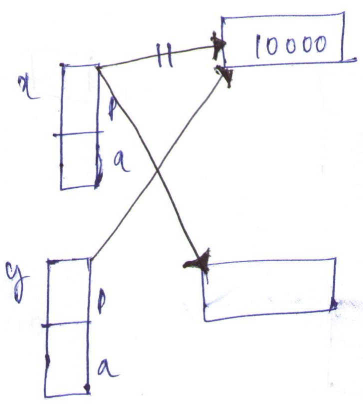

11
22
p
q


20000
3
~~10000~~
alias
x
p
200 ~~100~~
a
y
p
100
a

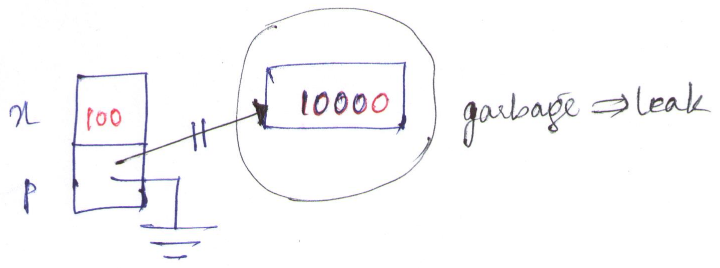

dangling pointer
10000
x
p
100
a
//created using malloc
on freeing
y
p
100
a



10000
x
p
a
y
p
a



10000
x
p
a
y
p
a



x
100
p
10000
garbage ⇒ leak