**Dynamic Memory Allocation**

This is the process by which memory is allocated as needed during run-time. The purpose is full/efficient utilization of memory.

void \*malloc(size\_t numbytes);

C

This function is used to allocate memory.

size\_t – unsigned integer

successful – returns pointer to start of allocated memory

unsuccessful – null pointer

void free(void \*ptr);

C

This function is used to free memory that has already been allocated. If the memory is not freed, reallocating memory to the same pointer would cause more memory to be allocated to the already allocated memory, but make the previously allocated memory to become inaccessible. This would eventually lead to memory shortages.

#include<stdio.h>  
#include<stdlib.h>  
int main()  
{  
 char \*p; *//can hold 80 characters* p = malloc(80); *//80 bytes of memory assigned* if(!p) *//if(p == NULL)* {  
 printf("Allocation Error\n");  
 exit(1);  
 }  
 puts("Enter a string:\n");  
 gets(p);  
 printf(p);  
 free(p); *//frees the memory that was allocated*}

C

void \*calloc(size\_t size, size\_t num);

C

This function also allocates memory. It takes the number of variables and the size of each variable in bytes.

char \*p;  
p = calloc(80, 4);

C

This allocates (80 variables) \* (4 bytes each) = 320 bytes of memory.

Depending on the processor, the number of variables that can be stored may differ. For 32-bit processors, integers have a size of 4 bytes so storing 80 integers would take 320 bytes of memory. For 15-bit processors, integers have a size of 2 byes, so storing 80 integers would take only 160 bytes of memory. Using calloc(80, 4) thus would waste 160 bytes of memory.

This problem can be solved with another function, sizeof. This returns the size of a particular variable in the system.

int a = sizeof int;  
*//this returns the size of integers in the system char \*p;*p = calloc(sizeof int, 80);  
*//this will allocate the required memory for 80 integers*

C

void realloc(void \*ptr, size\_t size);

C

This function frees the memory allocated to a pointer and reallocates memory to the pointer. It is essentially a combination of the free and malloc functions.

If the new memory is smaller than the old memory, the extra space is just freed. If the new memory is larger, the old memory is freed and the entire thing is allocated again. If there was information stored in the old memory, it is moved to the new memory. This is done to avoid overwriting data that may exist just after the old memory.

char \*p;  
p = calloc(sizeof int, 20); *//assigns 80 bytes of memory*p = realloc(sizeof int, 40); *//assigns 160 bytes of memory*

C

Old memory: 80 bytes (p)

int a

New memory: free space

int a

160 bytes(p)

The sizeof Function:

int main()  
{  
 int a;  
 printf("%d", sizeof a);  
}

C

OUTPUT: 4

int main()  
{  
 char a[10];  
 printf("%d", sizeof a);  
}

C

OUTPUT: 10

struct a  
{  
 int i;  
 char a[10];  
}test;  
int main()  
{  
 printf("%d", sizeof(test));  
}

C

OUTPUT: 16

Here the output should be 14, but it is 16. This is because memory can only be accessed as ‘words’. Each word has a certain size (in our case it is 4 bytes since we have 4-bit addressing). So only multiples of 4 bytes of memory can be accessed.

In the above example, 2 bytes of memory that are not actually used are also displayed as results. The unused memory is known as padding.

Previously, when finding the size of just a character or a character array, this problem did not come into play. That is because memory is handled differently when considering characters.

union a  
{  
 int i;  
 char ch[10];  
}test;  
int main()  
{  
 printf("%d", sizeof(test));  
}

C

OUTPUT: 10

In this case also, memory is allocated in consideration of the character array since that is the larger data type.

struct a  
{  
 unsigned u1 : 2;  
 unsigned u2 : 1; *//unsinged variables*}test;  
int main()  
{  
 printf("%d", sizeof test);  
}

C

OUTPUT: 4

Here, the output should have been 3, but it is 4. Padding is occurring here as well.