

Dynamic Allocation 1: Malloc and Free

☰ Tags	
🕒 Created time	@October 29, 2024 11:07 AM
☑ Reviewed	<input type="checkbox"/>

Use Memory as you go

- In production, the majority of programs in C use pointers to some extent.
- C also has a further facility called dynamic memory allocation that depends on the concept of a pointer and provides a strong incentive to use pointers in your code.
- Dynamic memory allocation allows memory for storing data to be allocated dynamically when your program executes.
- Allocating memory dynamically is possible only because you have pointers available.
- Think back to one of the exercise you have done, that calculated the average scores for a group of students. At the moment, it works for a fixed number of students (e.g. we have assumed the program can handle up to 50 students, so we declared an array `int grade[50]`).
- Ideally, the program should work for any number of students without knowing the number of students in the class in advance and without using any more memory than necessary for the number of student scores specified.
- Dynamic memory allocation allows you to do just that. You can create arrays at runtime that are just large enough to hold the amount of data you require for the task.

The `malloc()` Function

From the `stdlib.h` library.

When you use the `malloc()` function, you specify the number of bytes of memory that you want allocated.

The function returns the address of the first byte of memory that it allocated in the response to your request. Because you get an address, a pointer is the only place to put it.

A template for using the malloc function is:

```
<type> *<pointerName> = (<type>*)malloc(<number_of_bytes_allocated>);  
  
# Example  
  
int *pNumber = (int*)malloc(200);
```

In this example we have requested 200 bytes of memory and assigned the address of this block of memory to the pointer `*pNumber`

The pointer `*pNumber` will point to the first int location at the beginning of the 200 bytes that were allocated.

This whole block of memory can hold 50 `int` values on my computer, because each `int` require 4 bytes. You can think about the same principle when working with other types (e.g. `char`, `float`, `double` etc.).

If you don't want to remember the number of bytes needed for each data types, there is a better way to allocate the memory as follows:

If you don't want to remember the number of bytes needed for each data types, you can allocate memory as follows:

```
int *pNumber = (int*)malloc(50*sizeof(int));
```

In the above expression, the argument to `malloc()` is clearly indicating that sufficient bytes for accommodating 50 values of type `int` should be made available.

- If the memory that you request can't be allocated for any reason, `malloc()` returns a pointer with the value `NULL`.
- It's always a good idea to check any dynamic memory request immediately using an `if` statement to make sure the memory is actually there before you try to use it. For example, it is possible to write this:

```
int *pNumber = (int*)malloc(50*sizeof(int));
if(!pNumber)
{
    //... Code to deal with memory allocation failure, for e
    printf("Failed to allocate memory!");
}
```

Releasing Dynamically Allocated Memory

When you allocate memory, you should always release it when you no longer need it.

Memory that you allocate will be automatically released when your program ends, but it is better to explicitly release the memory when you are done, even if it is just before you exit from the program.

```
#pNumber points to the original allocation
free(pNumber)
pNumber = NULL;
```

Examples



Write a program accepts `n` integer numbers and prints the summary of the elements.

```
#include<stdio.h>
#include<stdlib.h>

int sumOfArray(int* pNumbers, int length);
int productOfArray(int *pNumbers, int length);

int main(int argc, char*argv[])
{
    int length = argc - 1;
    int *pNumbers = NULL;
    int *pResult = NULL;

    pNumbers = (int*)malloc(length*(sizeof(int)));
    if(!pNumbers)
    {
        //... Code to deal with memory allocation failure, for e
        printf("Failed to allocate memory!");
        return 0;
    }

    for(int i = 0; i < length; ++i)
    {
        *(pNumbers+i) = atoi(argv[i+1]);
    }

    /*Calculate sum */
    pResult = (int*)malloc(1*(sizeof(int)));
    *pResult = sumOfArray(pNumbers, length);
    printf("Sum of the array is: %d\n", *pResult);
    free(pResult); //Free the memory
```

```

    pResult = NULL;

    printf("Now the programing is calculating the product of the\n");

    /* Calculate the product */
    pResult = (int*)malloc(1*(sizeof(int)));
    *pResult = productOfArray(pNumbers, length);
    printf("Product of the array is: %d\n", *pResult);
    free(pResult); //Free the memory
    free(pNumbers);
    pResult = NULL;
    pNumbers = NULL;

    return 0;
}

int sumOfArray(int* pNumbers, int length)
{
    int sum = 0;
    for(int i = 0; i < length; ++i)
    {
        sum += *(pNumbers + i);
    }
    return sum;
}

int productOfArray(int *pNumbers, int length)
{
    int product = 1;
    for(int i = 0; i < length; ++i)
    {
        product *= *(pNumbers + i);
    }
    return product;
}

```