

## Malloc Best Practice

There are multiple valid ways to use the `malloc` function with one of them being considered the best practice way. Often, the non-best practice way is taught first since it's a bit easier to understand. Below is a demonstration of the two methods showing why one is considered better. The best practice way is considered better because by not explicitly typecasting the return value of `malloc` and not explicitly writing in the data type with the `sizeof` operator then the data type of the memory being allocated can be changed and still be correct without other additional changes having to be made to prevent errors. This can also be extrapolated to `calloc` and `realloc`.

```
typedef struct s1 {  
    int data1;  
} S1;
```

```
typedef struct s2 {  
    int data1;  
    int data2;  
} S2;
```

Create an array of 10 integers and create an S1 structure. Later, change the types.

### Non-best practice way

- Typecast the return value of `malloc`
- Explicitly write in the type in the `sizeof` operator

```
int* a = (int*)malloc(sizeof(int) * 10);  
S1* b = (S1*)malloc(sizeof(S1));
```

// You decide to change a to an array of doubles and b to an S2 structure

```
double* a = (int*)malloc(sizeof(int) * 10);    // error  
S2* b = (S1*)malloc(sizeof(S1));              // error
```

```
double* a = (double*)malloc(sizeof(double) * 10);    // no error after additional changes  
S2* b = (S2*)malloc(sizeof(S2));                    // no error after additional changes
```

### Best practice way

- Don't explicitly typecast the return value of `malloc`
- Don't explicitly write in the type with the `sizeof` operator

```
int* a = malloc(sizeof(*a) * 10);  
S1* b = malloc(sizeof(*b));
```

// You decide to change a to an array of doubles and b to an S2 structure

```
double* a = malloc(sizeof(*a) * 10);    // no error without additional changes  
S2* b = malloc(sizeof(*b));              // no error without additional changes
```

### Realloc Best Practice

A common error made when using `realloc` is not storing its return value in a temporary pointer. If the pointer holding the address of the data being resized is also used to store the return value of `realloc` then if `realloc` fails all of the data gets lost and there's a memory leak because `realloc` returns `NULL` when it fails. This is avoided if a temporary pointer is used. The following example uses an array of integers to demonstrate this.

#### Non-best practice way

- Resize the array using `realloc` and store the return value in the array's pointer in this case named *a*. If `realloc` fails it returns `NULL` which in turn stores `NULL` in *a* resulting in *a* no longer storing the address of the array. The array can no longer be accessed resulting in a loss of data and a memory leak.

```
int size = 10;
int* a = malloc(sizeof(*a) * size);
a = realloc(a, sizeof(*a) * size * 2); // if realloc fails a is now NULL
```

#### Best practice way

- Resize the array using `realloc` and store the return value in another pointer in this case named *temp*. If `realloc` fails it returns `NULL` which in turn stores `NULL` in *temp* resulting in *a* still storing the address of the array. The array can still be accessed resulting in no loss of data and no memory leak.
- Also note how *a* needs to be assigned the value of *temp*. This is because `realloc` will attempt to take the existing array and increase its size without making a whole new copy of the array. In this case, the line of code is redundant. However, this is not guaranteed to happen. It's possible that `realloc` won't be able to do this and the resized array will be a copy of the old array (plus the new memory from the resize) somewhere else in memory. In this case, *a* needs to be assigned the new array's address.

```
int size = 10;
int* a = malloc(sizeof(*a) * size);
int* temp = realloc(a, sizeof(*a) * size * 2); // if realloc fails the old array is still preserved in a
a = temp;                                     // redundant unless realloc had to make a copy of a
```