Mutation as Return Values & Double Pointers

Rob Hackman

Winter 2025

University of Alberta

Table of Contents

Pass-by-Pointer

Using Double Pointers

Returning Multiple Values

Let's write a function `maxInfo` that takes in an array of floats and returns the maximal value and the index of the maximal value.

Returning Multiple Values

Let's write a function `maxInfo` that takes in an array of floats and returns the maximal value and the index of the maximal value.

But how do we return two values?

maxinfo — attempt one

```
1   ??? maxInfo(float *arr, size_t len) {
2     assert(len > 0);
3     float maxSoFar = arr[0];
4     size_t maxInd = 0;
5     for (size_t i = 0; i < len; ++i) {
6       if (arr[i] > maxSoFar) {
7         maxSoFar = arr[i];
8         maxInd = i;
9       }
10    }
11    // Technically valid statement but not
12    // for returning two values.
13    return (maxSoFar, maxInd);
14  }
```

Recall: mutation

We can't return two values in C. We've written code to return an array but that won't cut it when we wan't to return both a `float` and a `size_t` as the are different types so we can't stick them in the same array.

Is returning a value the only way we can get data out to the caller though?

Recall: mutation

We can't return two values in C. We've written code to return an array but that won't cut it when we wan't to return both a `float` and a `size_t` as the are different types so we can't stick them in the same array.

Is returning a value the only way we can get data out to the caller though?

Recall: if the caller gives us a pointer to their data, we can manipulate that data. We can "return" values by writing them directly to memory the caller gives us!

`maxInfo` using mutation and return

```
1  float maxInfo(float *arr, size_t len, size_t *ind) {
2    assert(len > 0);
3    float maxSoFar = arr[0];
4    size_t maxInd = 0;
5    for (size_t i = 0; i < len; ++i) {
6      if (arr[i] > maxSoFar) {
7        maxSoFar = arr[i];
8        maxInd = i;
9      }
10   }
11   *ind = maxInd;
12   return maxSoFar;
13 }
```

Calling `maxinfo`

As with any function the caller must understand how to use `maxInfo` properly.

```
1  int main() {
2    float arr[3] = {-1.25, 10.2, 3.7};
3    size_t maxInd;
4    float max = maxInfo(arr, 3, &maxInd);
5    printf("Maximal value: %f\n", max);
6    printf("Index of max: %lu\n", maxInd);
7  }
```

Calling `maxInfo`

As with any function the caller must understand how to use `maxInfo` properly.

```
1  int main() {
2    float arr[3] = {-1.25, 10.2, 3.7};
3    size_t maxInd;
4    // Passing address of maxInd so the
5    // maxInfo function can write directly
6    // to the memory of our maxInd variable.
7    float max = maxInfo(arr, 3, &maxInd);
8    printf("Maximal value: %f\n", max);
9    printf("Index of max: %lu\n", maxInd);
10 }
```

Returning multiple values with mutation

In this way we can simulate returning as many values as we like through adding pointer parameters to our function and simply writing the values we'd like to return directly to the caller's memory.

This is fairly common in both C and C++ and so we'll see it again.

`maxInfo` using mutation only

```
1  void maxInfo(float *arr, size_t len,
2               float *max, size_t *ind) {
3    assert(len > 0);
4    float maxSoFar = arr[0];
5    size_t maxInd = 0;
6    for (size_t i = 0; i < len; ++i) {
7      if (arr[i] > maxSoFar) {
8        maxSoFar = arr[i];
9        maxInd = i;
10     }
11   }
12   *ind = maxInd;
13   *max = maxSoFar;
14 }
```

Calling `maxInfo` using mutation only

```
1  int main() {
2    float arr[3] = {-1.25, 10.2, 3.7};
3    size_t maxInd;
4    float max;
5    maxInfo(arr, 3, &max, &maxInd);
6    printf("Maximal value: %f\n", max);
7    printf("Index of max: %lu\n", maxInd);
8  }
```

Table of Contents

Pass-by-Pointer

Using Double Pointers

The `insert` function we wrote previously for adding to our dynamic array didn't work if there wasn't enough space in the array for another element.

We wrote code to allocate a new larger array and copy over all the elements from the old array to the new in order to make a new larger copy.

Can `insert` do this itself so that the caller doesn't have to check if there's enough memory first?

Let's write a function `push` that appends a value to the end of a dynamic array, and if needed allocates more space.

Attempt at `push`

```
1  void push(int *arr, int val, size_t len, size_t cap) {
2    if (len == cap) {
3      int *newArr = malloc(sizeof(int)*cap*2);
4      for (size_t i = 0; i < len; ++i) {
5        newArr[i] = arr[i];
6      }
7      free(arr);
8      cap = cap*2;
9      arr = newArr;
10   }
11   arr[len] = val;
12   ++len;
13 }
```

Testing new `push`

Try this main with our new `push` function with different inputs.

```
int main() {
  int *arr = malloc(sizeof(int)*4);
  size_t len = 0, cap = 4;
  while (!feof(stdin)) {
    int x = 444;
    int rc = scanf("%d\n", &x);
    if (rc == 1) push(arr, x, len, cap);
  }
  printArray(arr, len);
  free(arr);
}
```

Testing new `push`

Try this main with our new `push` function with different inputs.

```
int main() {
  int *arr = malloc(sizeof(int)*4);
  size_t len = 0, cap = 4;
  while (!feof(stdin)) {
    int x = 444;
    int rc = scanf("%d\n", &x);
    if (rc == 1) push(arr, x, len, cap);
  }
  printArray(arr, len);
  free(arr);
}
```

Nothing prints — why?

`push` problem

```c
int main() {
  int *arr = malloc(sizeof(int)*4);
  size_t len = 0, cap = 4;
  while (!feof(stdin)) {
    int x = 444;
    int rc = scanf("%d\n", &x);
    if (rc == 1) push(arr, x, len, cap);
  }
  printArray(arr, len);
  free(arr);
}
```

Nothing prints because `len` is still 0. We only ever updated the local parameter `len` in `push`, which had no affect on `main`'s variable.

`push` problem

```
1  int main() {
2    int *arr = malloc(sizeof(int)*4);
3    size_t len = 0, cap = 4;
4    while (!feof(stdin)) {
5      int x = 444;
6      int rc = scanf("%d\n", &x);
7      if (rc == 1) push(arr, x, len, cap);
8    }
9    printArray(arr, len);
10   free(arr);
11 }
```

If we want `push` to change `main`'s `len` we'll need to provide its address to `push` so it can manipulate our memory directory.

push take two

```
1  void push(int *arr, int val, size_t *len, size_t cap) {
2    if (*len == cap) {
3      int *newArr = malloc(sizeof(int)*cap*2);
4      for (size_t i = 0; i < *len; ++i) {
5        newArr[i] = arr[i];
6      }
7      free(arr);
8      cap = cap*2;
9      arr = newArr;
10   }
11   arr[*len] = val;
12   ++*len;
13 }
```

`push` problem

```
1  int main() {
2    int *arr = malloc(sizeof(int)*4);
3    size_t len = 0, cap = 4;
4    while (!feof(stdin)) {
5      int x = 444;
6      int rc = scanf("%d\n", &x);
7      if (rc == 1) push(arr, x, &len, cap);
8    }
9    printArray(arr, len);
10   free(arr);
11 }
```

We update main to pass the address of `len` and try again...

`push` problem

```
1  int main() {
2    int *arr = malloc(sizeof(int)*4);
3    size_t len = 0, cap = 4;
4    while (!feof(stdin)) {
5      int x = 444;
6      int rc = scanf("%d\n", &x);
7      if (rc == 1) push(arr, x, &len, cap);
8    }
9    printArray(arr, len);
10   free(arr);
11 }
```

And our program exhibits strange behaviour printing incorrect values and crashing. We note that the cap variable is also not being updated in `main`

`push` take three

```
1  void push(int *arr, int val, size_t *len, size_t *cap) {
2    if (*len == *cap) {
3      int *newArr = malloc(*cap*sizeof(int)*2);
4      for (size_t i = 0; i < *len; ++i) {
5        newArr[i] = arr[i];
6      }
7      free(arr);
8      *cap = *cap*2;
9      arr = newArr;
10   }
11   arr[*len] = val;
12   ++*len;
13 }
```

Take 3 — problem

We update our main accordingly again, this time our program crashes before reaching EOF.

What's happening? Let's consider the error[1]:

```
free(): double free detected in tcache 2
Aborted (core dumped)
```

The error means exactly that — we're calling `free` on the same pointer twice.

[1] *Likely* error, this is undefined behaviour so not guaranteed.

Take 3 — problem

We update our main accordingly again, this time our program crashes before reaching EOF.

What's happening? Let's consider the error[1]:

```
free(): double free detected in tcache 2
Aborted (core dumped)
```

The error means exactly that — we're calling `free` on the same pointer twice.

But how?

[1] *Likely* error, this is undefined behaviour so not guaranteed.

Take 3 — problem

Once again we forgot to propogate all changes `push` was making out to `main`.

`push` was also allocating a new array, somewhere else on the heap, and assigning its *local* variable `arr` to point at it.

We the pointer `arr` in `main` was still pointing at the original memory address — it's now a dangling pointer since `push` freed it when $size = cap = 4$. That same instance the new array `push` allocated was leaked as the pointer to that new memory was lost when `push` returned.

In order for `push` to mutate the pointer `arr` that `main` has, we have to provide `push` with `arr's` memory address — a double pointer.

push **fixed**

```
1  void push(int **arr, int val, size_t *len, size_t *cap) {
2    if (*len == *cap) {
3      int *newArr = malloc(*cap*sizeof(int)*2);
4      for (size_t i = 0; i < *len; ++i) {
5        newArr[i] = (*arr)[i];
6      }
7      free(*arr);
8      *cap = *cap*2;
9      *arr = newArr;
10   }
11   (*arr)[*len] = val;
12   ++*len;
13 }
```

push fixed

```
1  void push(int **arr, int val, size_t *len, size_t *cap) {
2    if (*len == *cap) {
3      int *newArr = malloc(*cap*sizeof(int)*2);
4      for (size_t i = 0; i < *len; ++i) {
5        newArr[i] = (*arr)[i];
6      }
7      free(*arr);
8      *cap = *cap*2;
9      *arr = newArr;
10   }
11   (*arr)[*len] = val;
12   ++*len;
13 }
```

main for fixed push

```
1  int main() {
2    int *arr = malloc(sizeof(int)*4);
3    size_t len = 0, cap = 4;
4    while (!feof(stdin)) {
5      int x = 444;
6      int rc = scanf("%d\n", &x);
7      if (rc == 1) push(&arr, x, &len, &cap);
8    }
9    printArray(arr, len);
10   free(arr);
11 }
```

Practice Problem

Write a function `pop` that takes in an array of integers and *removes* the last item from the array. The function should have a `void` return type and any changes it needs to make to the callers data should be done through mutation.

Lots of data representing one thing

That's becoming a lot of parameters to pass to the `push` function and keep straight to maintain the state of our dynamically growing array.

But all of these variables are just parts of our dynamic array. Our length and capacity are meaningless except with respect to our array.

So what can we do?