Tirgul3 - Agenda

- malloc & realloc
- more about memory management
- visibility, duration and linkage

Dynamic array

- In static arrays we need to know the array size at compilation time:

```
int arr[5] = {1,2,3,4,5};
```

- But, this size may not be known in advance, or might be changed during the program execution.

- **The solution: use dynamic array:**

```
int *arr = (int*)malloc(sizeof(int)*arraySize);
```

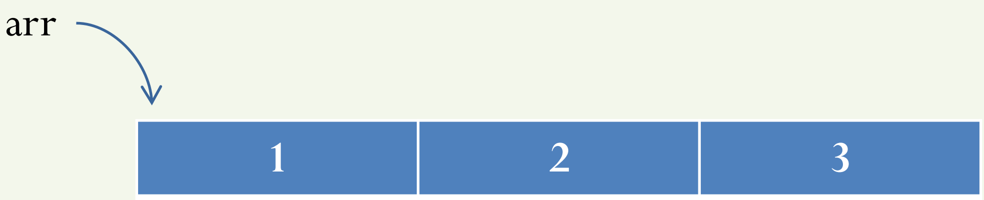
Array reallocation – **wrong** solution

```
int *arr = (int*)malloc(sizeof(int)*arraySize);
//..put some values in arr
int* newArr =
  (int*)malloc(sizeof(int)*(arraySize+1));
//..copying values from arr to newArr
arr = newArr;
```

// BAD: lost address of first allocation!

Array reallocation

- Allocate array on the heap and assign a pointer to it

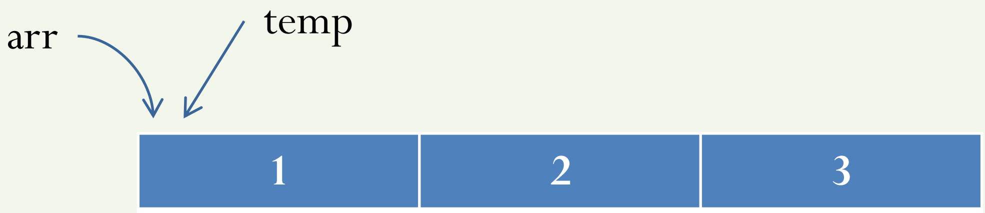


arr

1	2	3

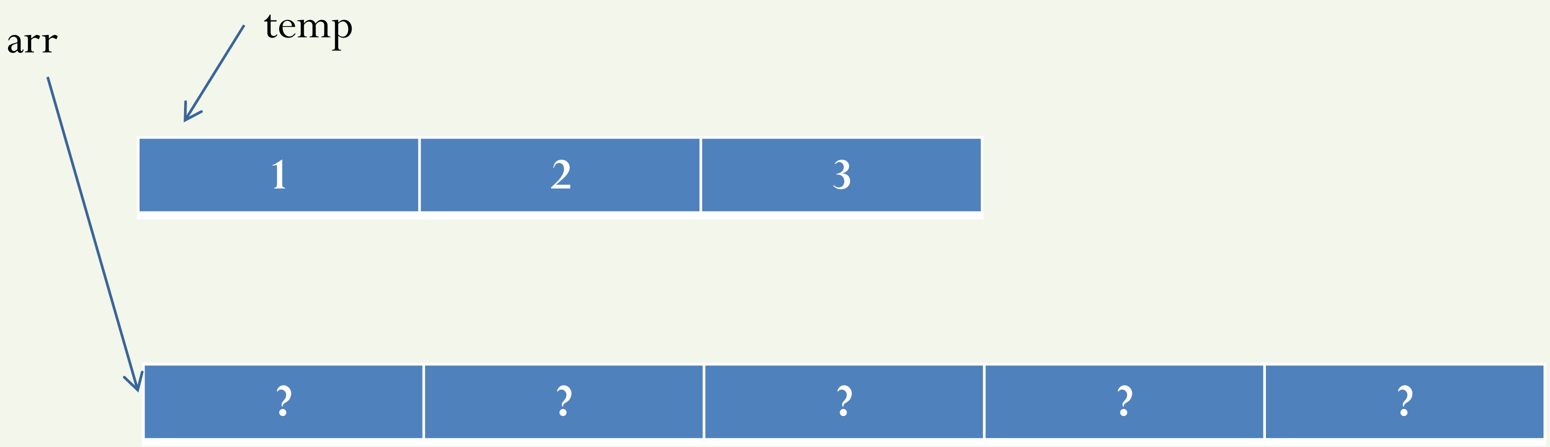
Array reallocation

- Use temporary pointer to point to the old array.



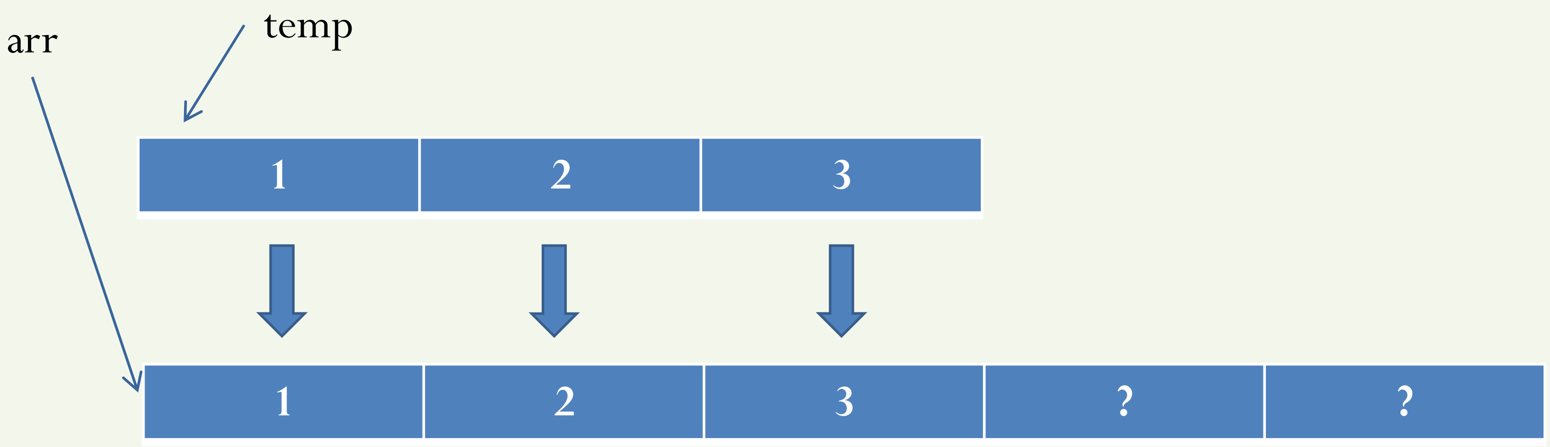
Array reallocation

- Use temporary pointer to point to the old array.
- Reallocate new array.



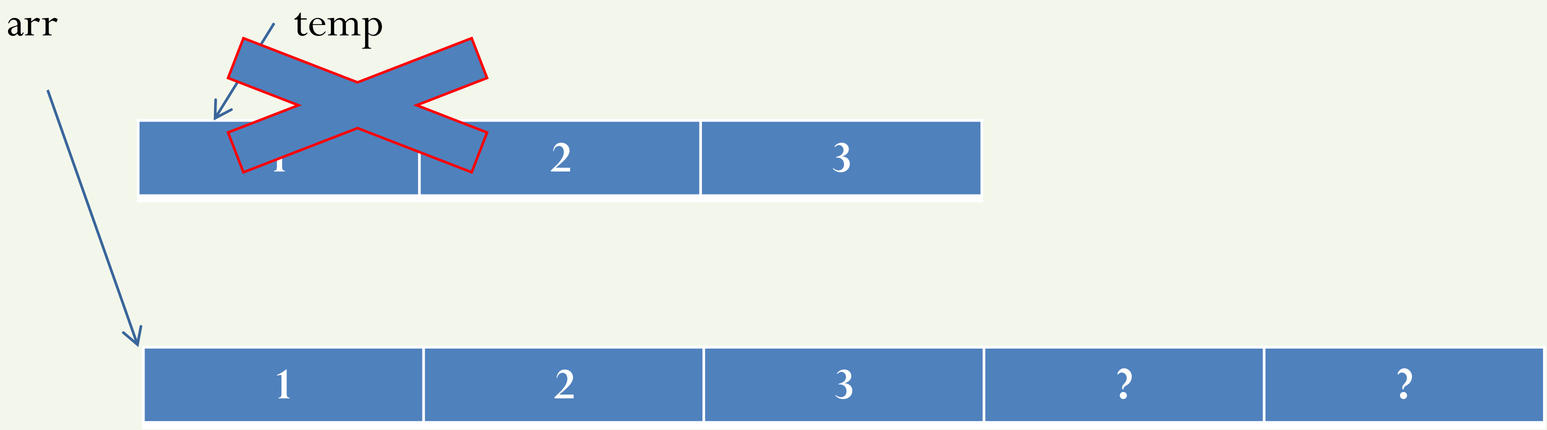
Array reallocation

- Use temporary pointer to point to the old array.
- Reallocate new array.
- Copy values from the old array to the new one.



Array reallocation

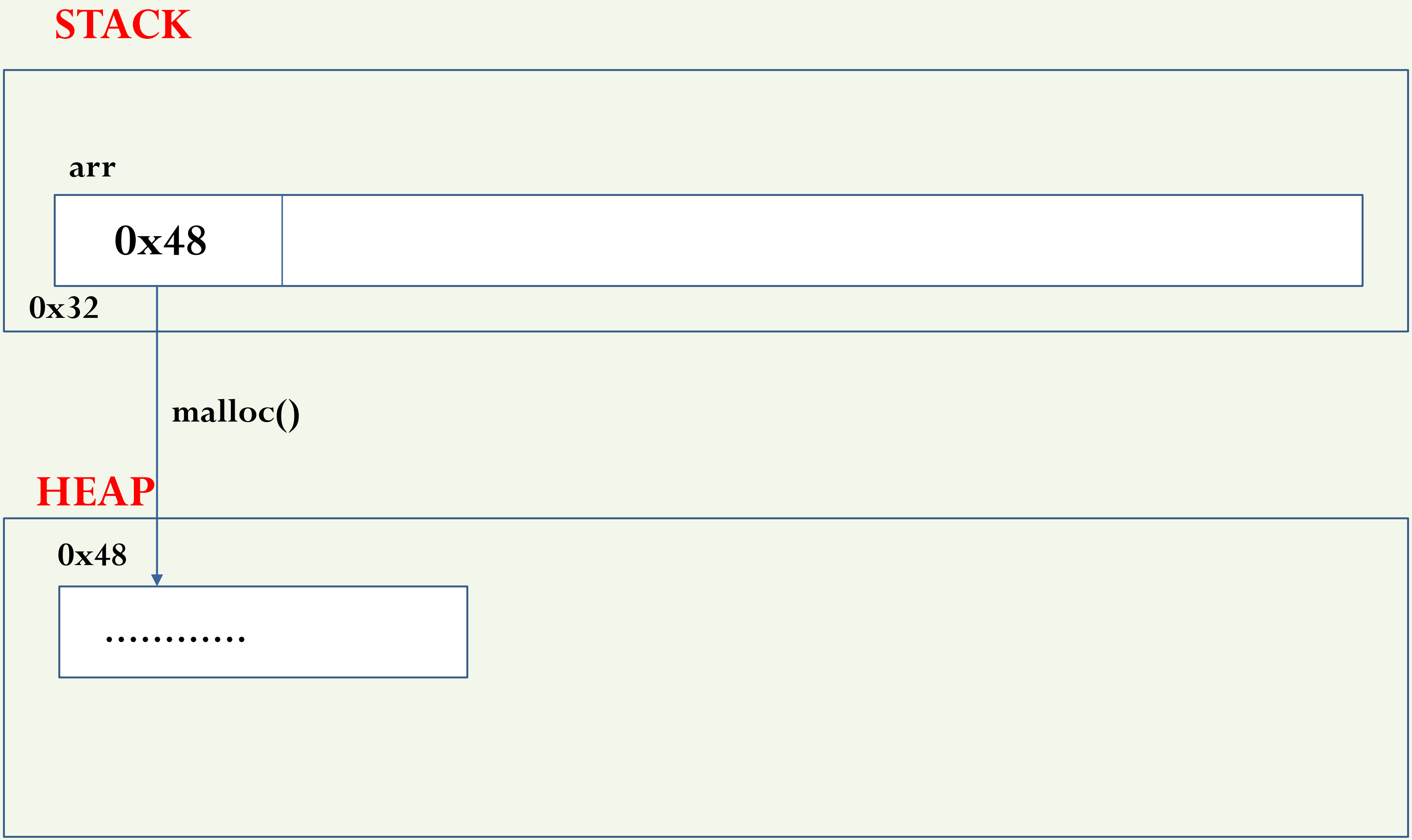
- Use temporary pointer to point to the old array.
- Reallocate new array.
- Copy values from the old array to the new one.
- Free the old array.



Array reallocation – version 1

```
int* reallocateMemory(int *arr, unsigned int oldSize, unsigned int newSize)
   { //Partial example due to space limitations
      int *temp = arr;
      arr = (int*)malloc(sizeof(int)*newSize);
      int i = 0;
      while( i < oldSize)
      {
         arr[i] = temp[i];
         i++;
      }
      free(temp);
      return arr;
   }
int main()
   {
      int *arr = (int*)malloc(sizeof(int)*arrSize);
      //do some stuff…
      arr = reallocateMemory(arr,arrSize,newSize);
      free(arr);
```

Short look of the memory…



STACK

arr

0x48

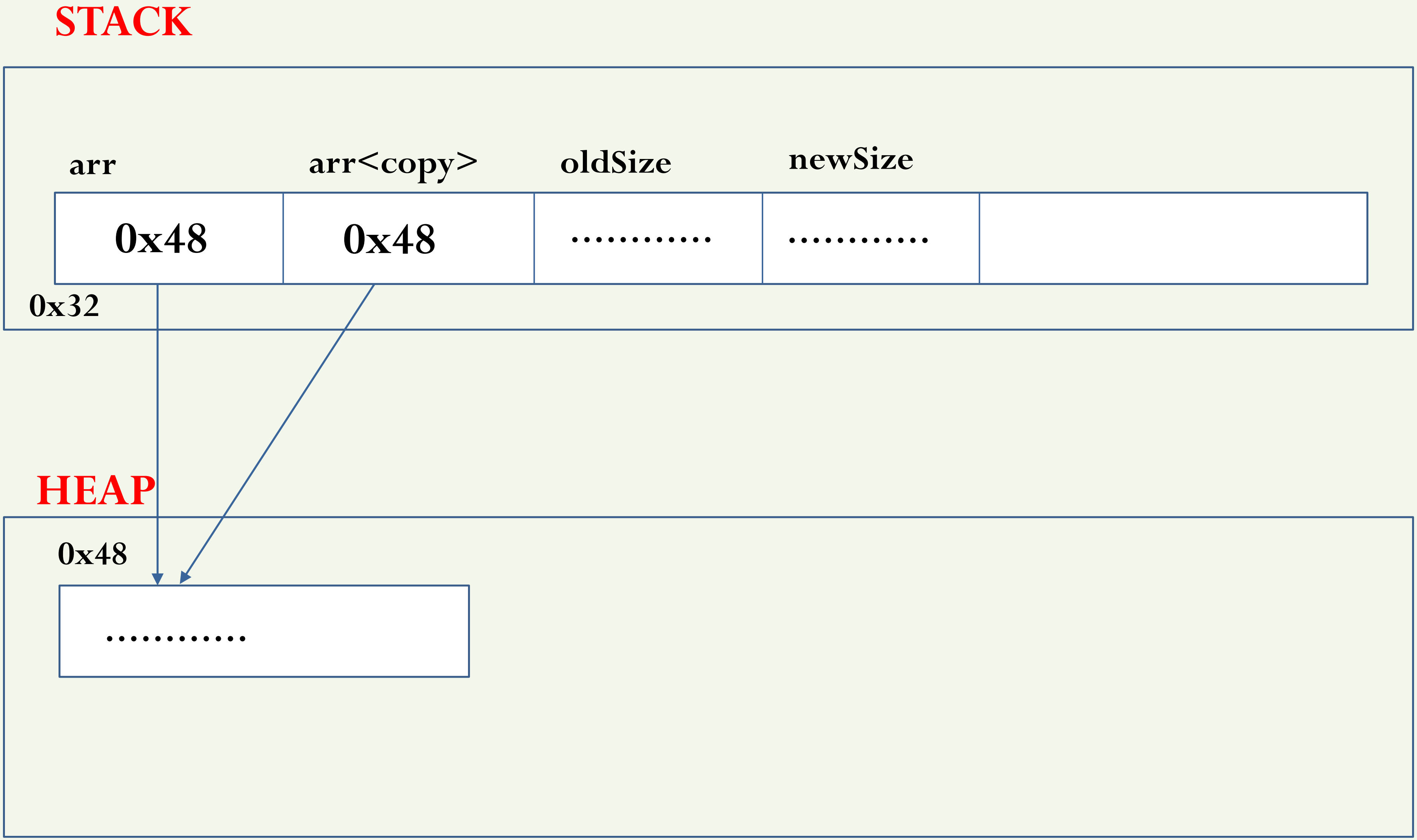
0x32

malloc()

HEAP

0x48

…………



STACK

arr

arr<copy>

oldSize

newSize

0x48

0x48

....

....

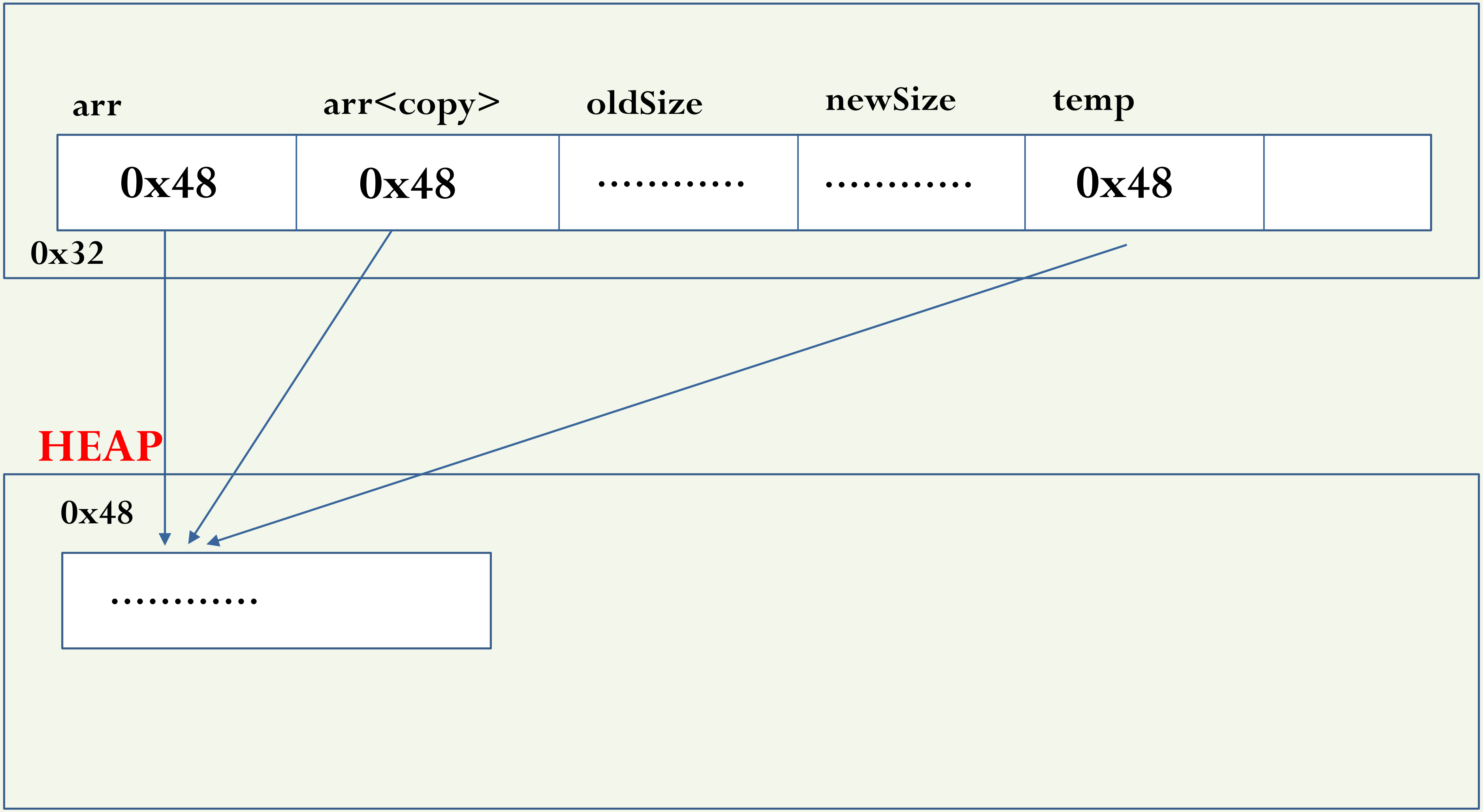
0x32

HEAP

0x48

....

STACK



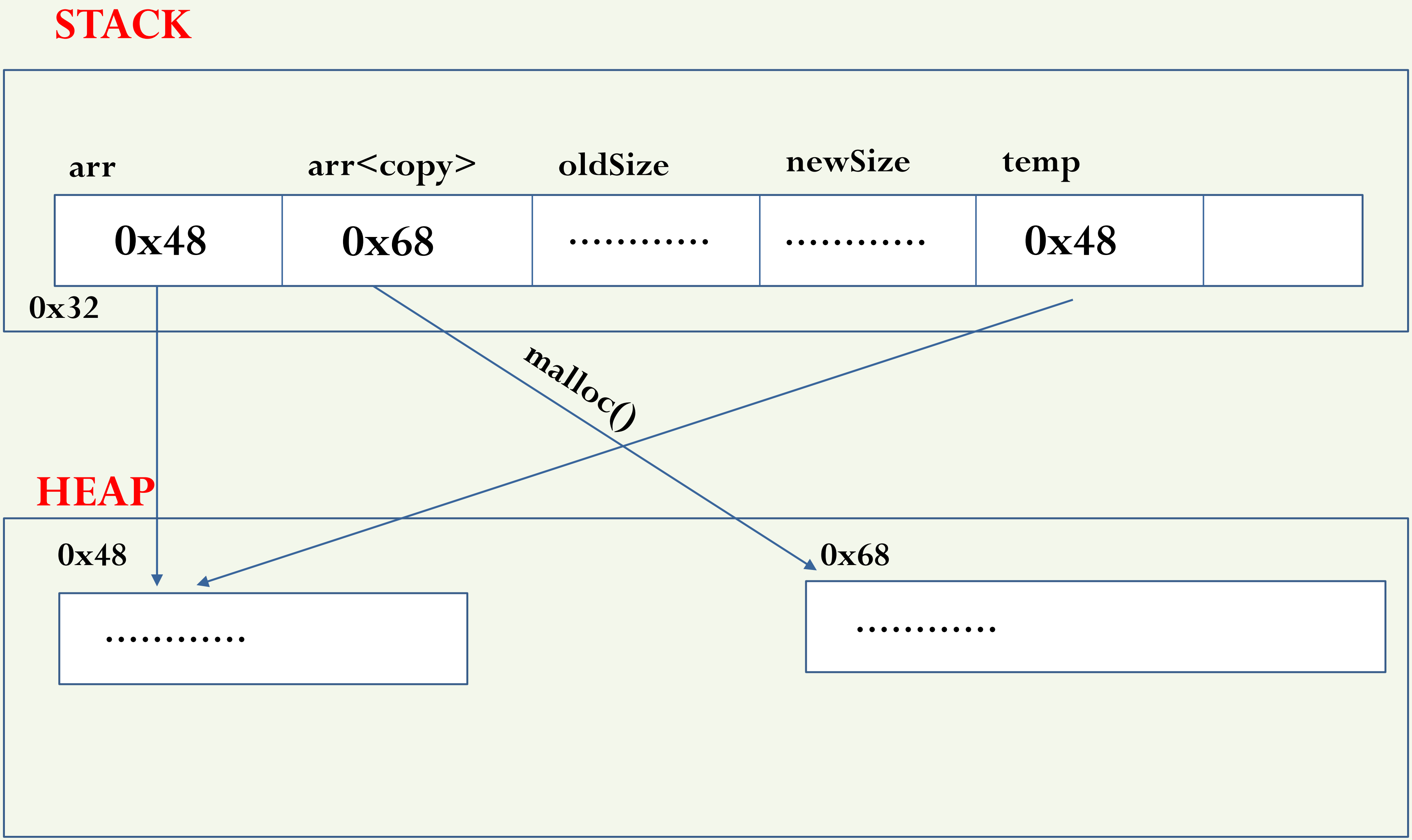
arr	arr<copy>	oldSize	newSize	temp	
0x48	0x48	....	....	0x48	

0x32

HEAP

0x48

....



STACK

arr arr<copy> oldSize newSize temp

0x48 0x68 ………… ………… 0x48

0x32

malloc()

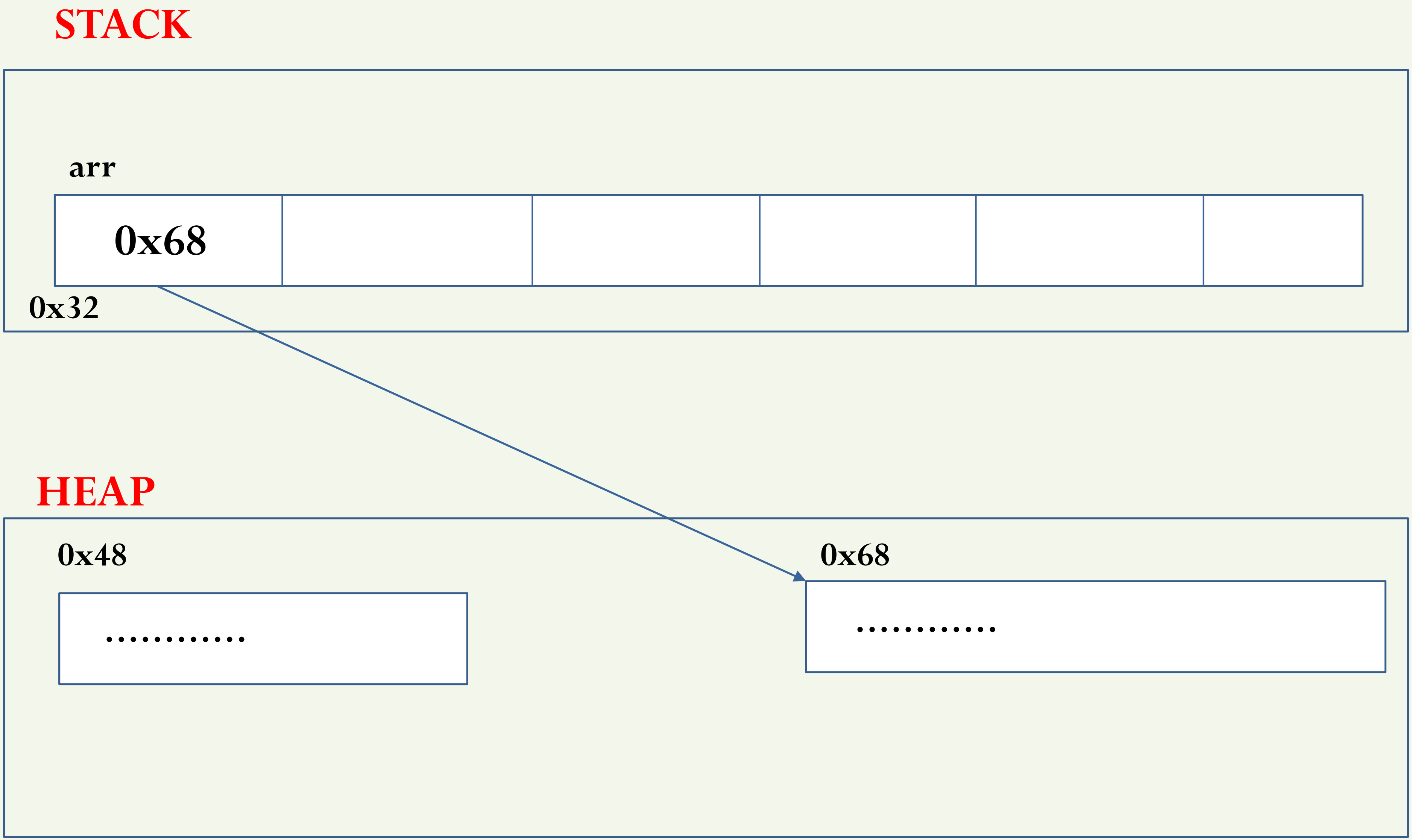
HEAP

0x48

…………

0x68

…………



STACK

arr

0x68

0x32

HEAP

0x48

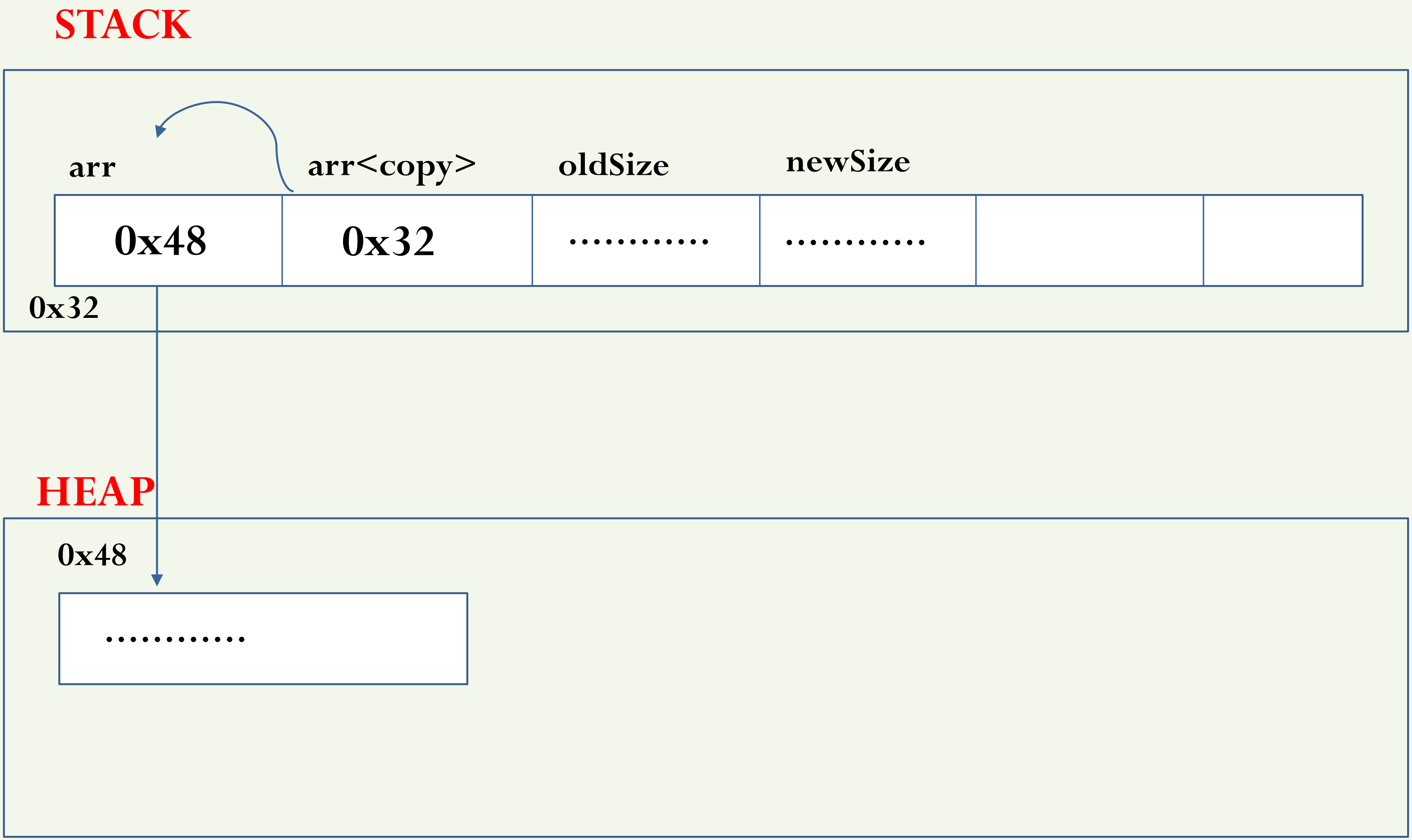
....

0x68

....

Array reallocation – version 2

```
void reallocateMemory(int **arr, unsigned int oldSize, unsigned int newSize)
   { //partial example (checking alloc…)
      int *temp = *arr;
      *arr = (int*)malloc(sizeof(int)*newSize);
      int i = 0;
      while( i < oldSize)
      {
         (*arr)[i] = temp[i];
         i++;
      }

      free(temp);

   }

   int main()

   {

      int *arr = (int*)malloc(sizeof(int)*arrSize);

      //do some stuff …

      reallocateMemory(&arr,arrSize,newSize);

      free(arr)
```



STACK

arr

arr<copy>

oldSize

newSize

0x48

0x32

…………

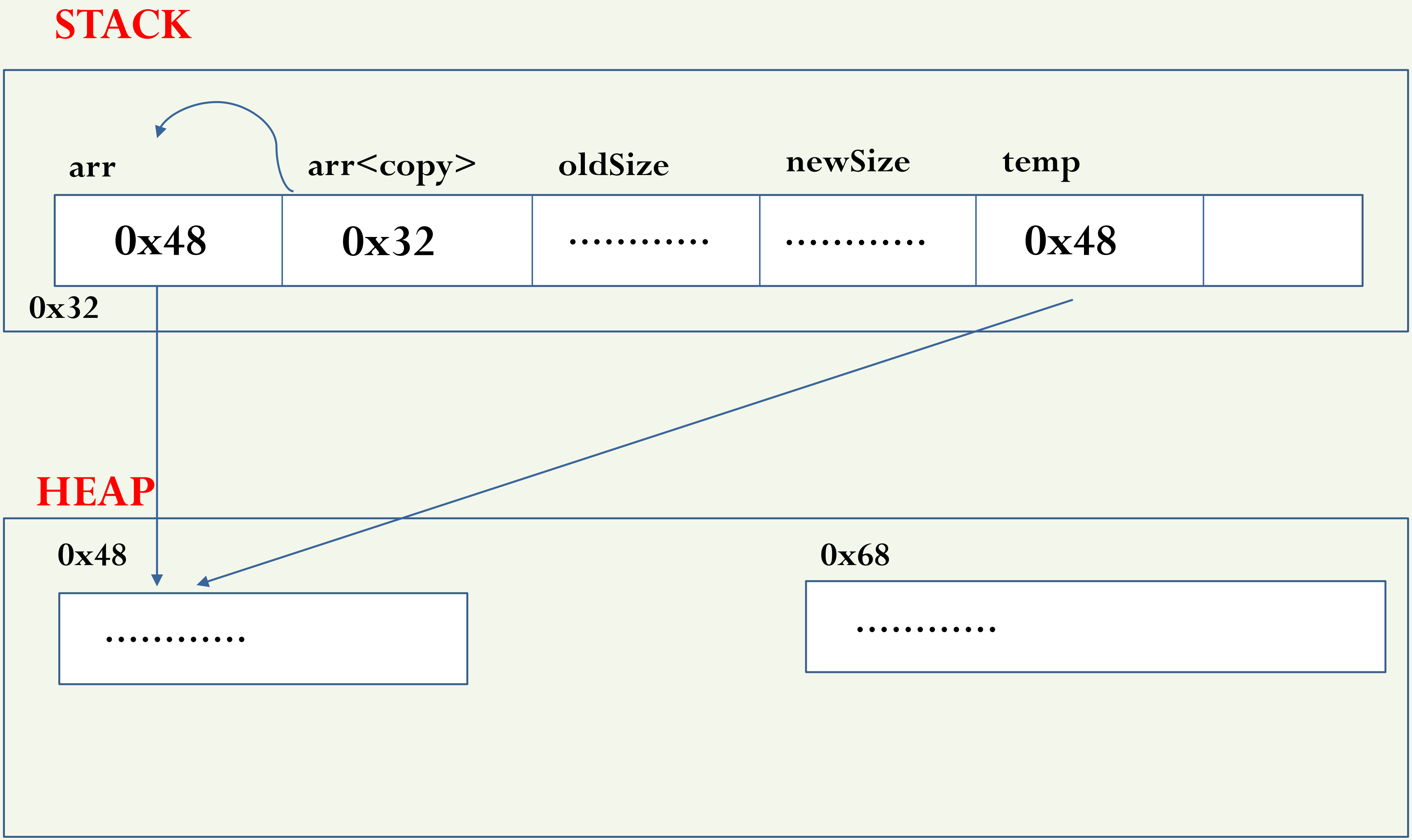
…………

0x32

HEAP

0x48

…………



STACK

arr

arr<copy>

oldSize

newSize

temp

0x48

0x32

....

....

0x48

0x32

HEAP

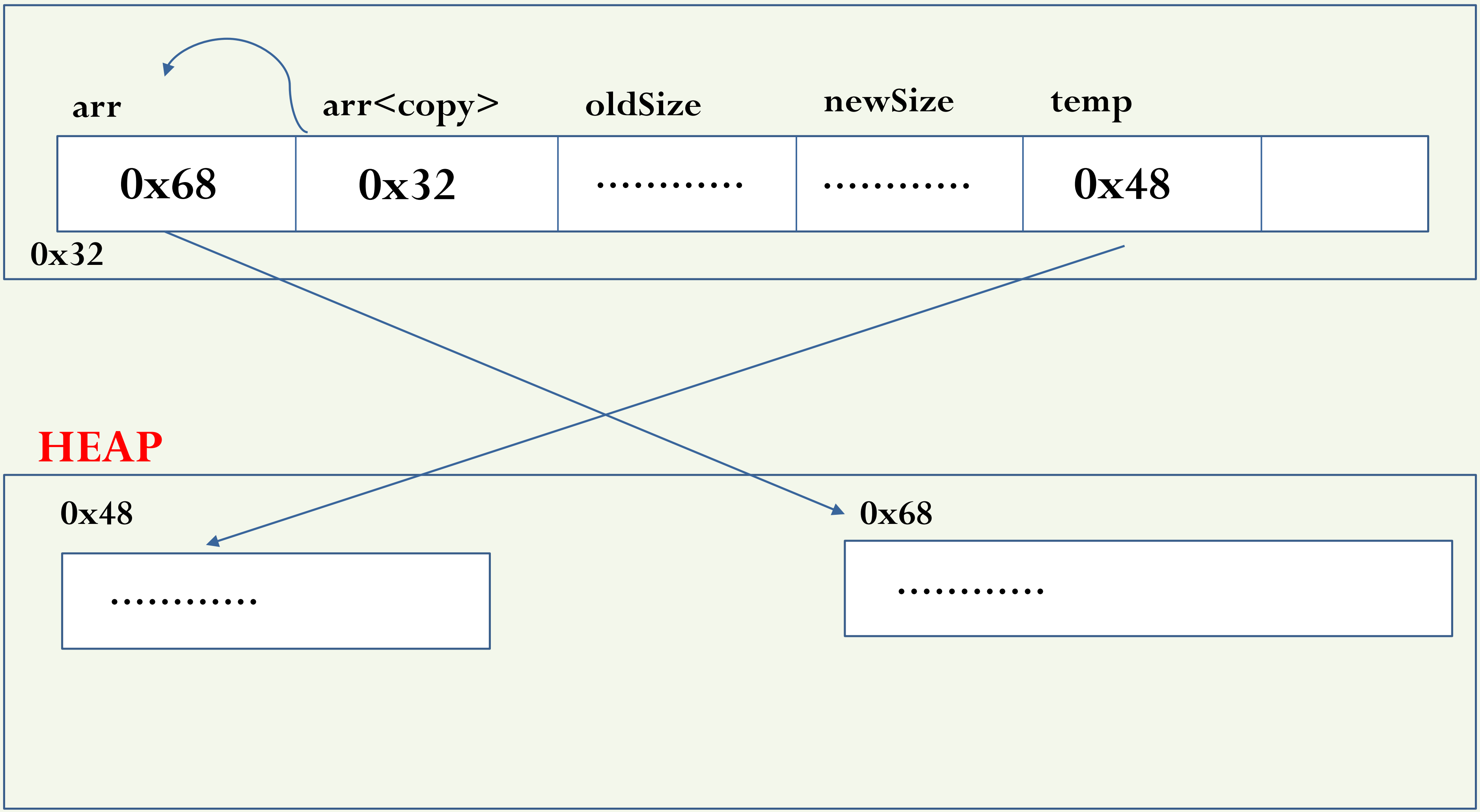
0x48

....

0x68

....

STACK



arr	arr<copy>	oldSize	newSize	temp	
0x68	0x32	…………	…………	0x48	

HEAP

Array reallocation – using realloc

```
int * arr =
  (int*)malloc(sizeof(int)*oldSize);

arr = (int*)realloc(arr,sizeof(int)*newSize);
```

- realloc tries to reallocate the new memory in place, if fails, tries elsewhere
- The old data is preserved
- The new cells contents is undefined
- If `arr`=NULL, behaves like malloc

Rule 1: Do not return an address of local variable!!!!!

```
int *foo()
{
        int a = 5;
        ….


        return &a;
}
```

```
int *goo()
{
        int arr[10] = {0};
        int *p = arr;
        ….

        return p;
}
```

Rule 1: Do not return an address of local variable!!!!!

```
int *foo()
{
        int a = 5;
        ….


        return &a;
}
```

```
int *goo()
{
        int arr[10] = {0};
        int *p = arr;
        ….

        return p;
}
```

Rule 2: Always check pointer!=NULL

```
char *str = (char*)malloc(5*sizeof(char));
if (str == NULL)
{
   // print error message or perform
   // other relevant operation
   exit(1); // we don’t have to exit
}
```

Rule 3: after free() operation do: pointer = NULL. (not must but recommended)

```
int* iptr =
     (int*) malloc(sizeof(int));


...
free(iptr);
iptr=NULL;
```

How to copy a string?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
   const char *p1 = "hi mom“;
   char *p2 = (char*)malloc(strlen(p1) + 1);
   strcpy(p2,p1);
   printf("%s\n",p2);
   return 0;
}
```

Rule 4

Rule 5: each malloc() should have corresponding free()

Good design:

the function which allocate memory should free it or document that the user must do that!

To find memory bugs

- Use valgrind!
  - Self study
  - See the tutorial at the course website (under TA lectures).

Static variables in a function: visibility vs. duration

- Static variables duration is the entire program running time.
- Static variables in a function keep their value for the next call to the function
- Memory is allocated on global space (called: static heap)

```
int getUniqueID()
{
   static int id=0;
   id++;
   return id;
}
int main()
{
   int i = getUniqueID();
   int j = getUniqueID();
}
```

Static variables in a function: visibility vs. duration

- Static variables duration is the entire program running time.
- Static variables in a function keep their value for the next call to the function
- Memory is allocated on global space (called: static heap)

```
int getUniqueID()
{
   static int id=0;
   id++;
   return id;
}
int main()
{
   int i = getUniqueID(); //i=1
   int j = getUniqueID(); //j=2
}
```

Understanding “extern”

1. Declaration can be done any number of times but definition only once.
2. When “extern” is used with a variable, it’s only declared not defined.

BUT

3. When an “extern” variable is declared with initialization, it is taken as definition of the variable as well.

Static and extern variables, cont.

- “static” variable on the global scope
  - Available only in the current module
- “extern” variable
  - May be defined outside the module

file1.c

```c
int y;
static int x;
int z;
int myFunc1()
{
   x = 3;
}
```

file2.c

```c
extern int y; // y should be imported (from file1.c )
extern int x; // x should be imported (from file1.c)
int myFunc2()
{
   extern int z; // z from file1.c
   y = 5;
   x = 3; //linker error
}
```

Static functions.

- “static” function - available only in the current module.