MODULE 1 / UNIT 7

STACK, HEAP, AND MEMORY MANAGEMENT

Today

- **(Probably) the last unit on C++ ‘programming’**
  - .. before moving back to algorithm

- **Memory management in C++:**
  - Arguably the most confusing part in C/C++ language
  - Most ”challenging” errors occurs during memory management in C++.

```cpp
#include <iostream>
#include <string>
using namespace std;

string hello_str() {
  string s("hello");
  return s;
}

const char* hello_char_arr() {
  char s[] = "Hello";
  return s;
}

int main(int argc, char** argv) {
  cout << hello_str() << endl;
  cout << hello_char_arr() << endl;
  return 0;
}
```

A simple C++ function

- **Example output:**

```
hello
F??
```

Why?

Python and R implementations

```python
def hello():
    s = "hello"
    return(s)

print(hello())
```

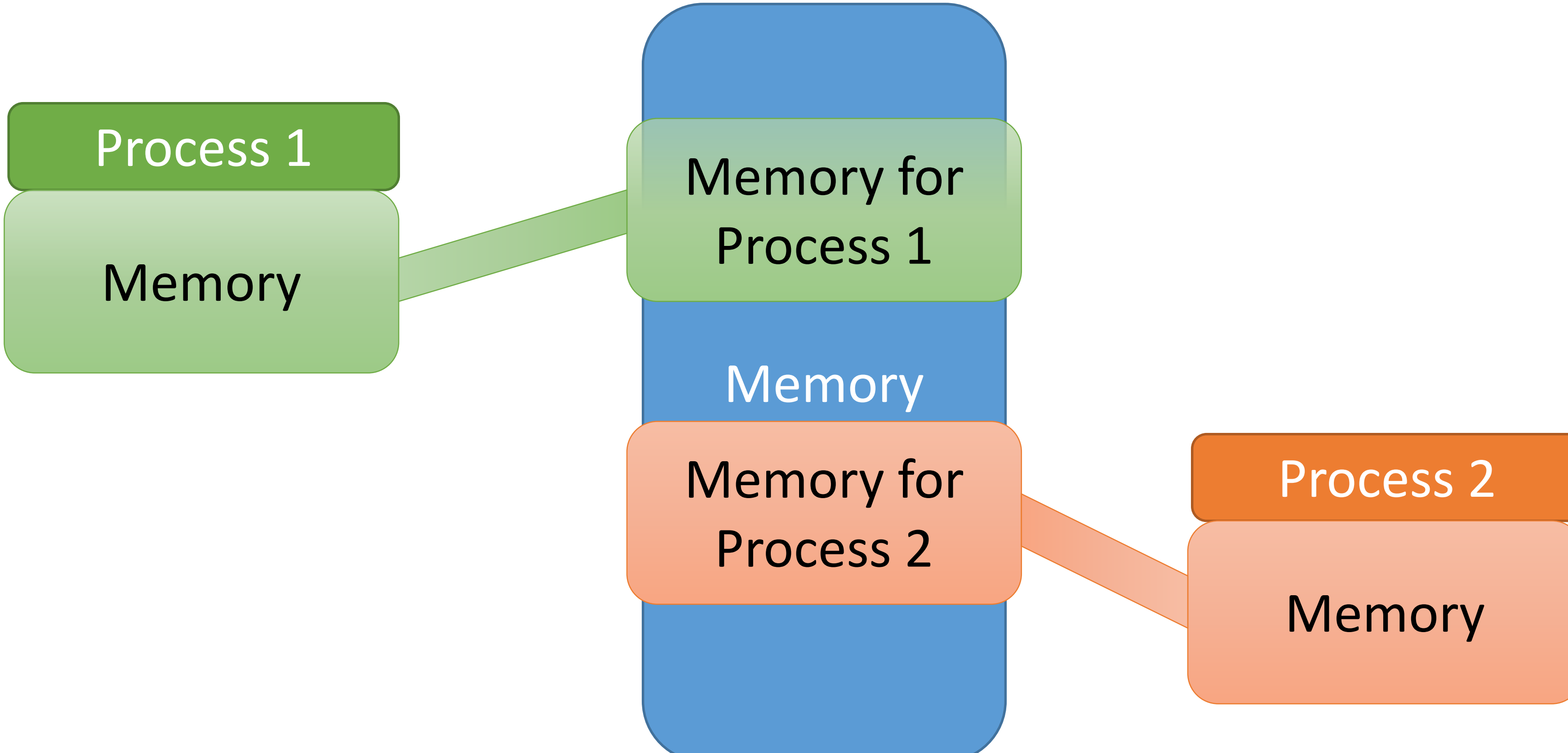
```r
hello <- function() {
  s <- "hello"
}

cat(hello())
cat("\n")
```

```
hello
```

```
hello
```

Each process have their own “protected” memory

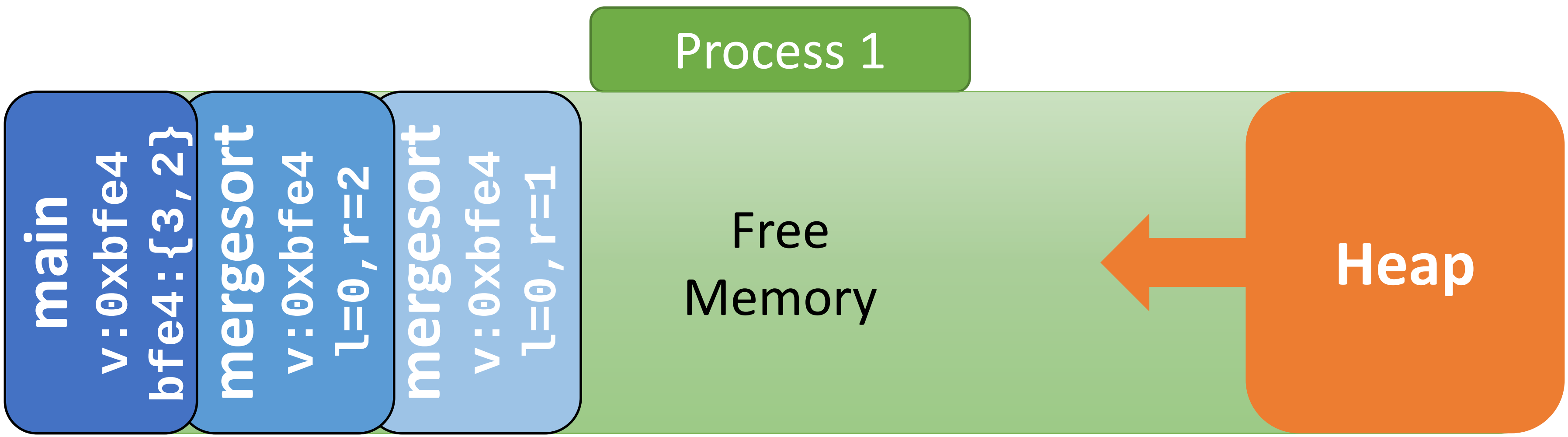


Each function has two+ types of accessible memory

Stack stores variables within functions

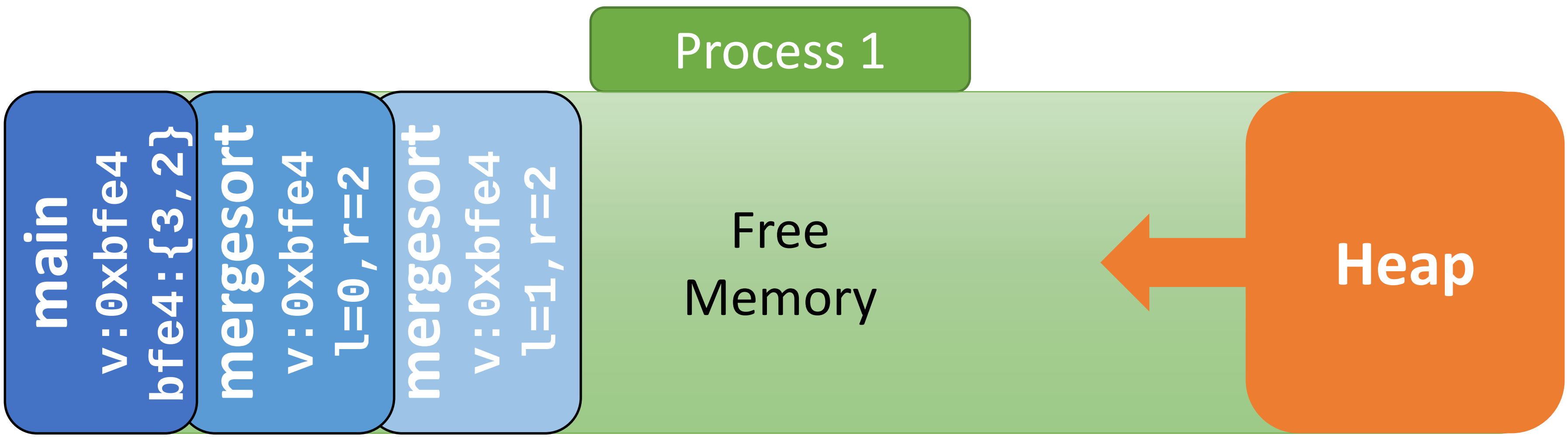
Stack stores variables within functions

Stack stores variables within functions



Stack stores variables within functions

Stack stores variables within functions



Stack stores variables within functions

Stack stores variables within functions

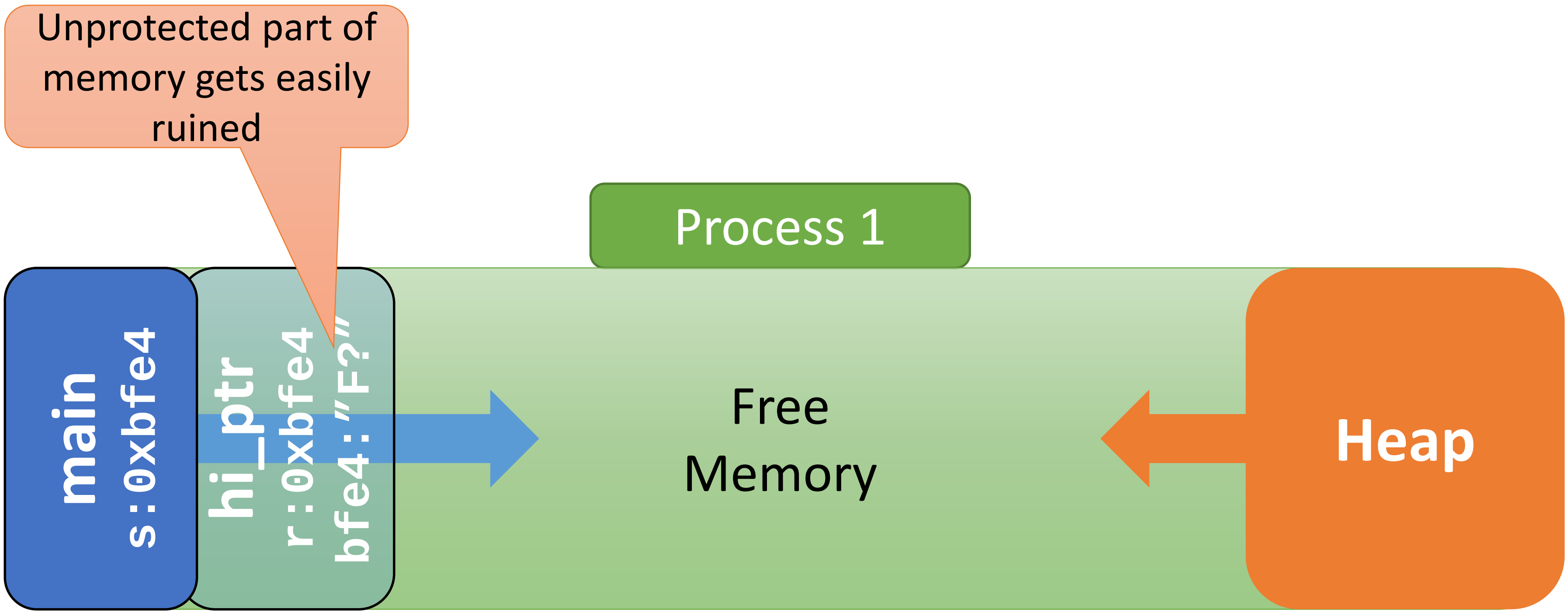
Stack stores variables within functions

In the case of `hello_char_arr()`

In the case of hello_char_arr()

In the case of `hello_char_arr()`

In the case of `hello_char_arr()`



What if a function has to return new objects?

```cpp
#include <iostream>
#include <string>
using namespace std;

int* make_array() {
  int A[3] = {3,2,1};
  return A;
}

int main() {
  int* B = make_array();
  cout << B[0] << "\t" << B[1] << "\t" << B[2] << endl;
  return 0;
}
```

Compiler warns that something went wrong

```
kang2015:615_1_7 hmkang$ g++ local_return.cpp
local_return.cpp:7:10: warning: address of stack memory associated with local variable
      'A' returned [-Wreturn-stack-address]
  return A;
         ^
1 warning generated.
```

.. and the results are not as expected either.

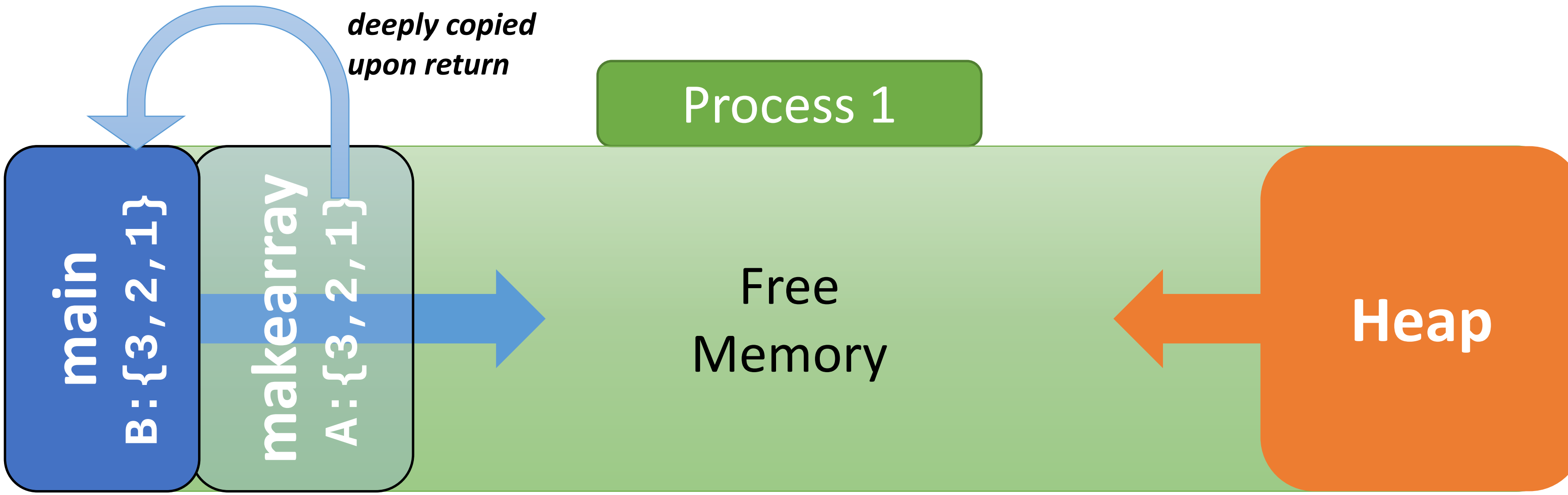
```
kang2015:615_1_7 hmkang$ ./a.out
3       143572844       1
```

When a function returns a complex object...

If shallow-copied, the original object can be ruined

One solution is to make a deep-copy happen

A and B are `vector<int>`



```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

vector<int> make_vector() {
  vector<int> A(3);
  A[0] = 3; A[1] = 2; A[2] = 1;
  return A;
}

int main() {
  vector<int> B = make_vector();
  cout << B[0] << "\t" << B[1] << "\t" << B[2] << endl;
  return 0;
}
```

Caveats of return-by-deep-copy

- **Deep copy is costly.**
  - Requires additional consumption of memory and CPU while copying
  - Problematic especially when returning a large object

- **Returning multiple objects by deep copy is not easy in C++.**
  - If more than one object has to be returned, a special implementation is required

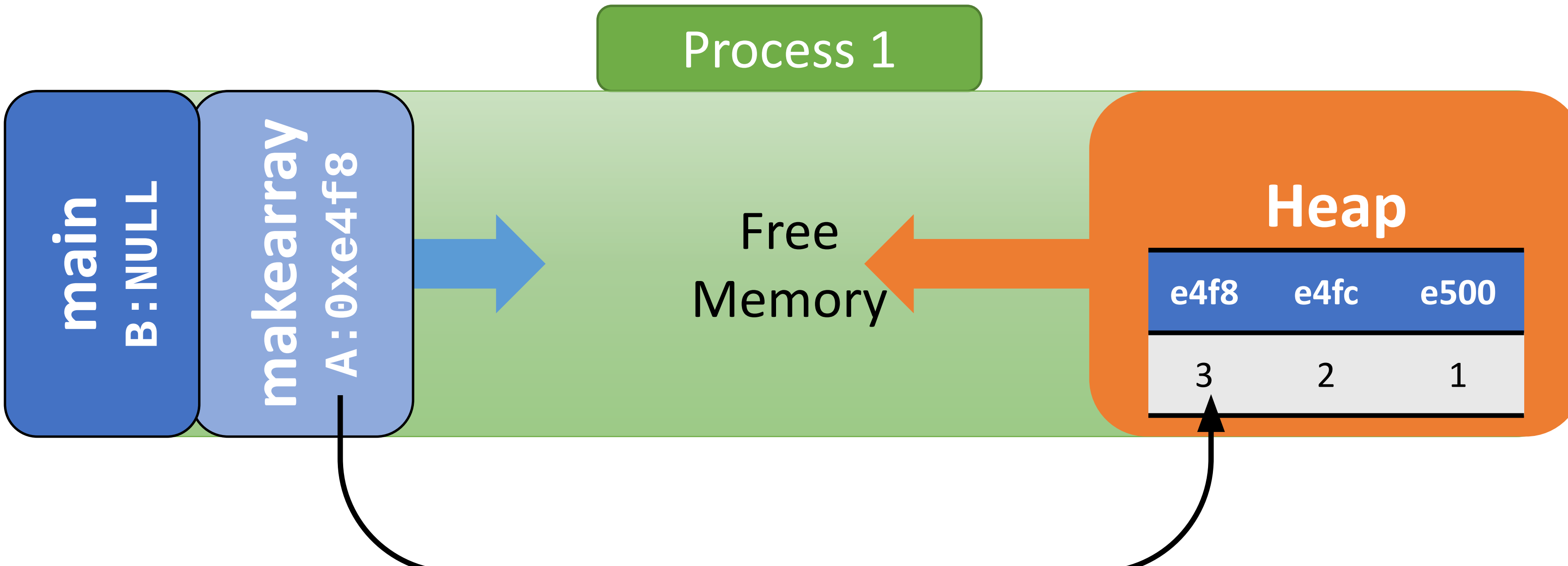
Can we do something like this?

```
int* make_array() {
   int* A = // something
// this function creates an array of {3,2,1}
// and return the array using a shallow-copy (via a pointer)
// but I want somehow magically to protect
// the array created in this function
// even after the function finishes
   return A;
}
```

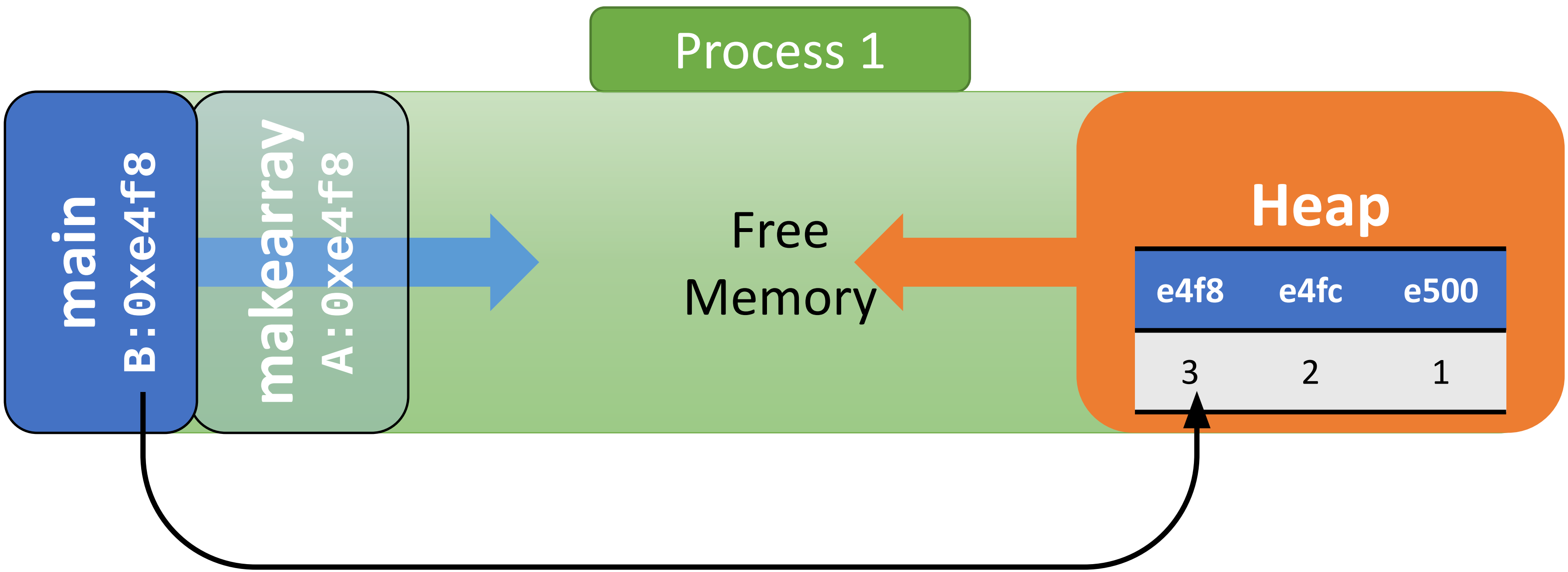
B is NULL initially

A is NULL initially, too

A new array is created in heap



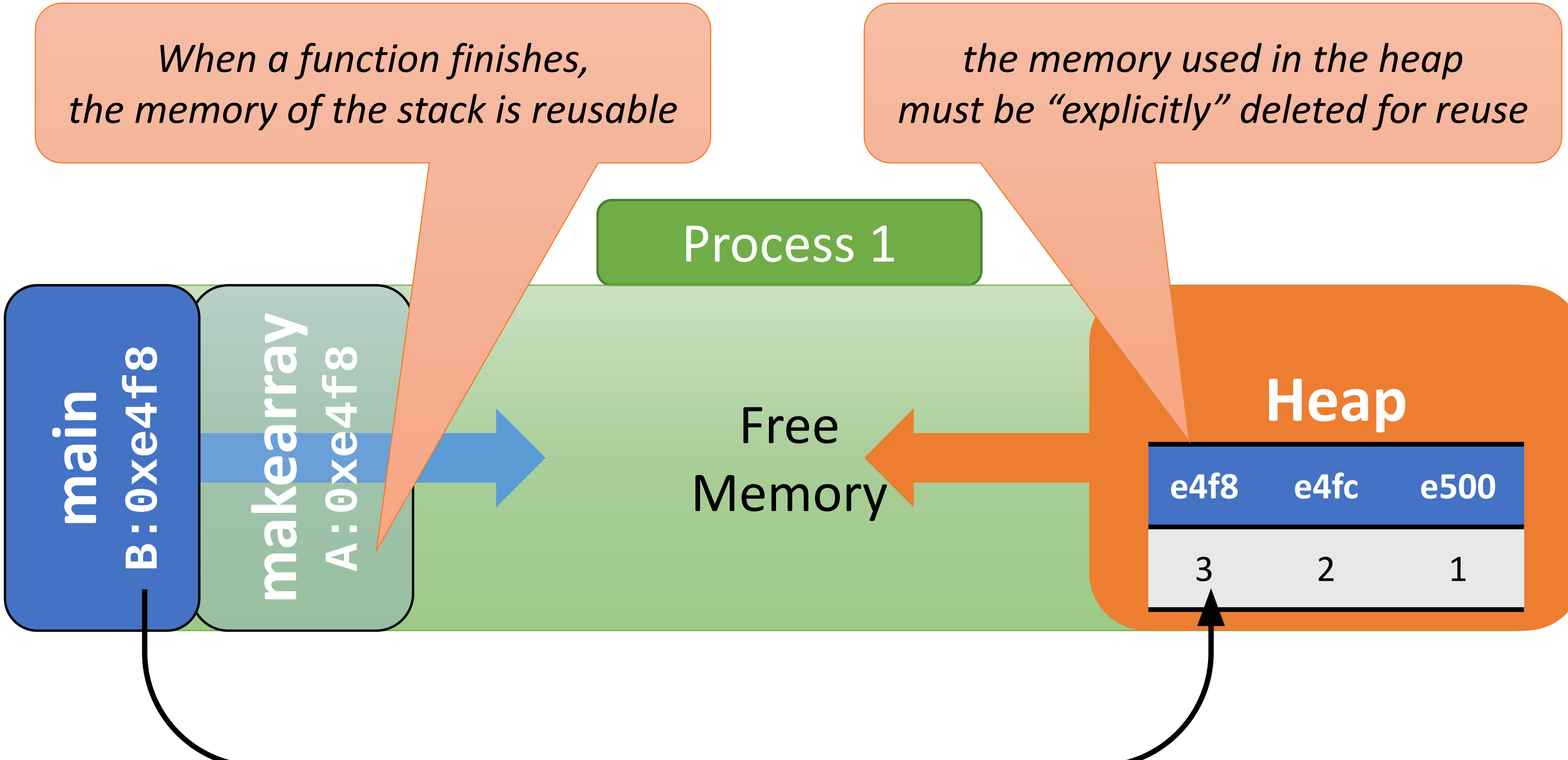
Even if the function finishes, allocated memory is still protected



The actual C++ implementation

```
// the function returns the pointer to an array
// where the array is allocated in heap
int* make_array() {
   // new[] operator allows creating an array in heap
   int* A = new int[3];
   A[0] = 3; A[1] = 2; A[0] = 1; // assign values
   return A; // returning A will only copy the pointer
}
int main() {
   int* B = make_array();
   ...
```

When the object is no longer needed..



Modified implementation with `delete[]`

```
int* make_array() {
    // new[] operator allows creating an array in heap
    int* A = new int[3];
    A[0] = 3; A[1] = 2; A[0] = 1; // assign values
    return A; // returning A will only copy the pointer
}
int main() {
    int* B = make_array();
    // do something with B
    delete[] B; // reclaim the memory space
    ...
}
```

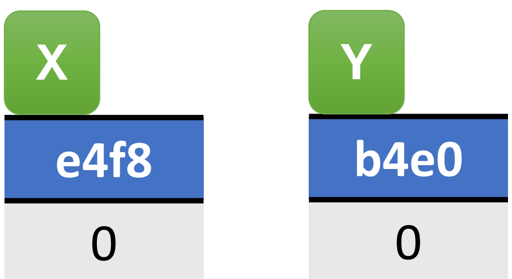
What if we want to return multiple things?

```
void make_arrays(int* A, int* B) {
// this function creates two arrays
// but do not return anytyhing.
// Instead, it modifies the value or A and B
// so that it the newly created arrays can be replaced to
// their original values
}

// Would this way work?
```

Would this way work?

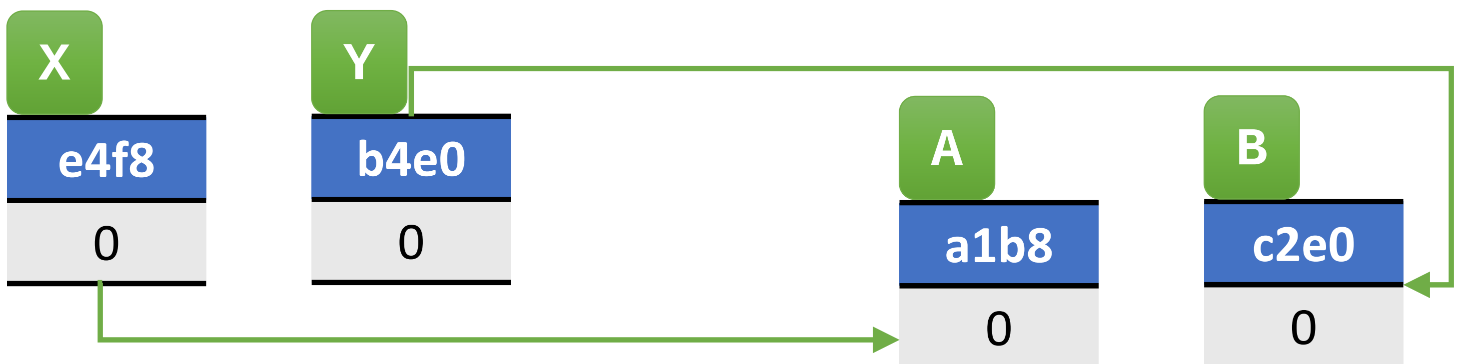
```
main()
int* X = NULL;
int* Y = NULL;
```



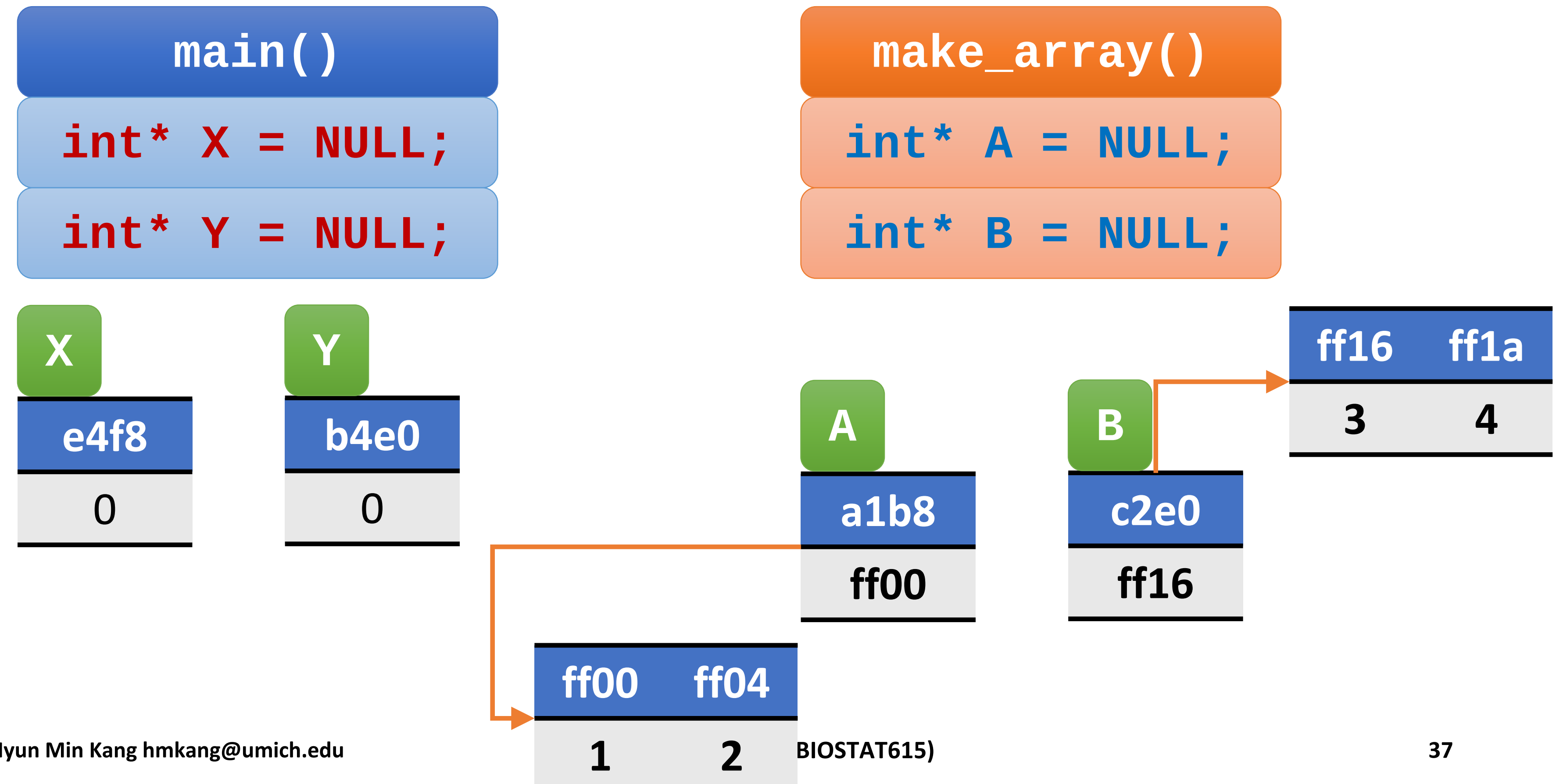
Would this way work?

```
main()
int* X = NULL;
int* Y = NULL;
```

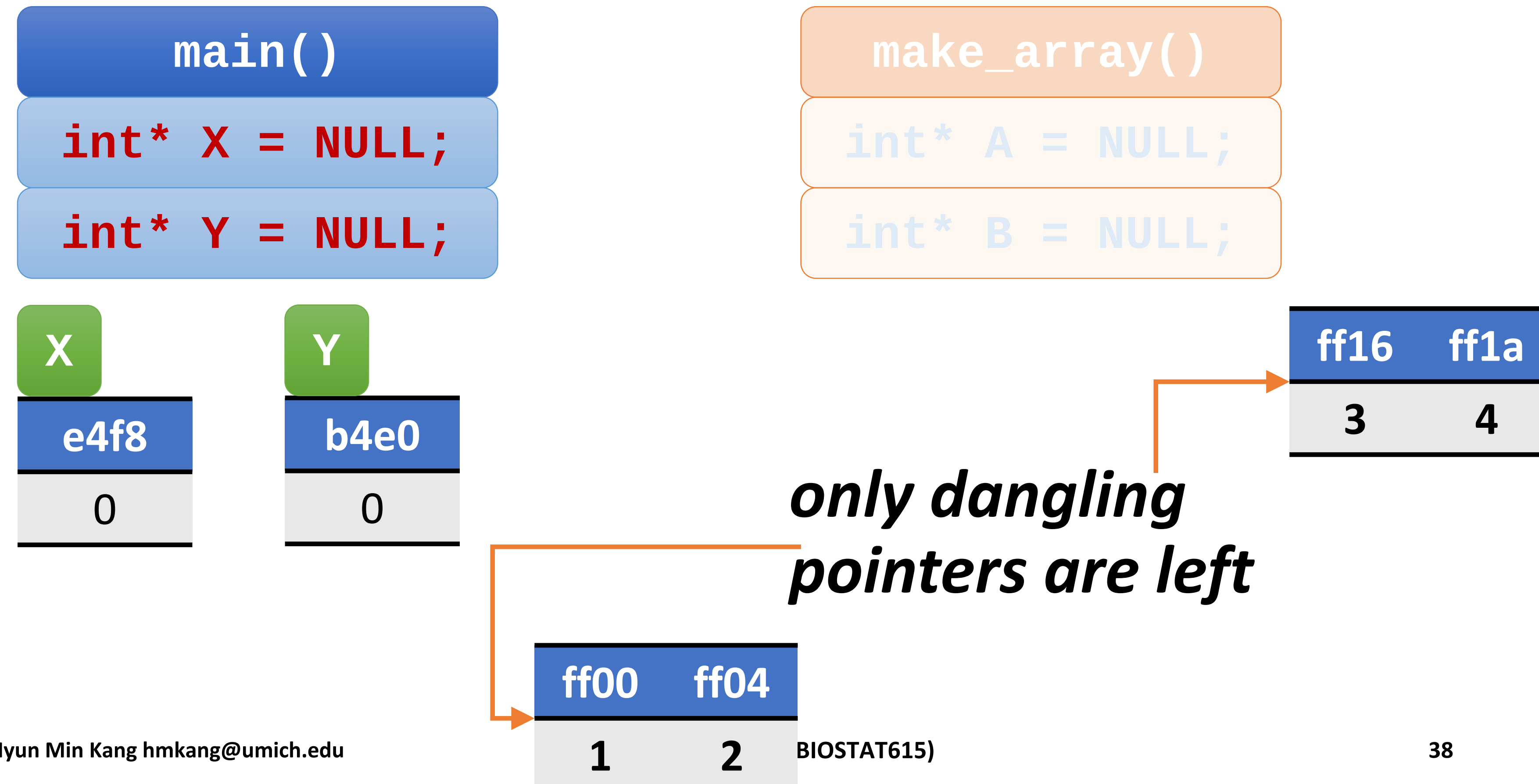
```
make_array()
int* A = NULL;
int* B = NULL;
```



After creating new objects in the heap..

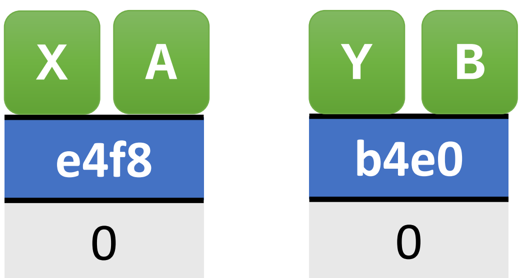


If the function finishes,



A right way to fix : pass by reference

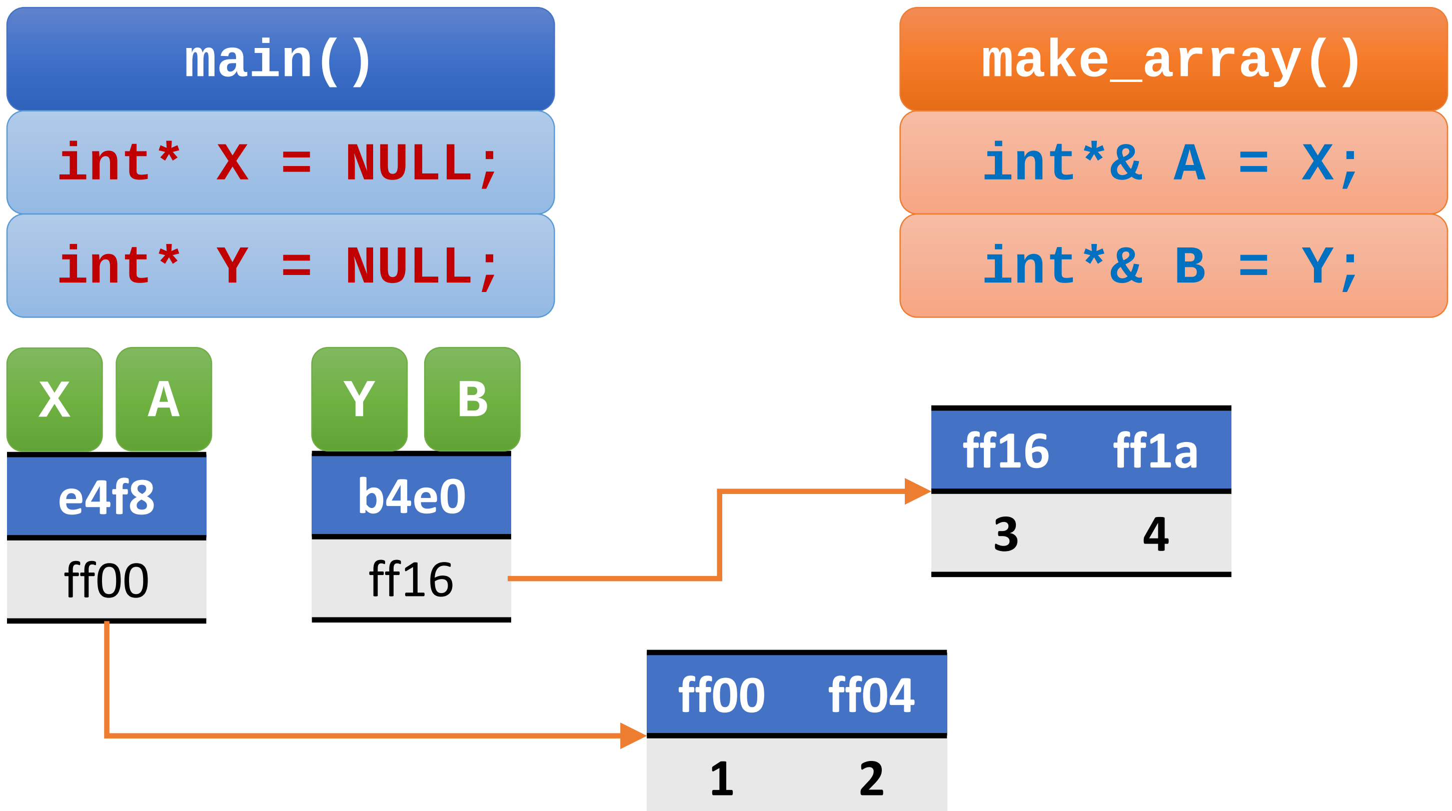
```
main()
int* X = NULL;
int* Y = NULL;
```



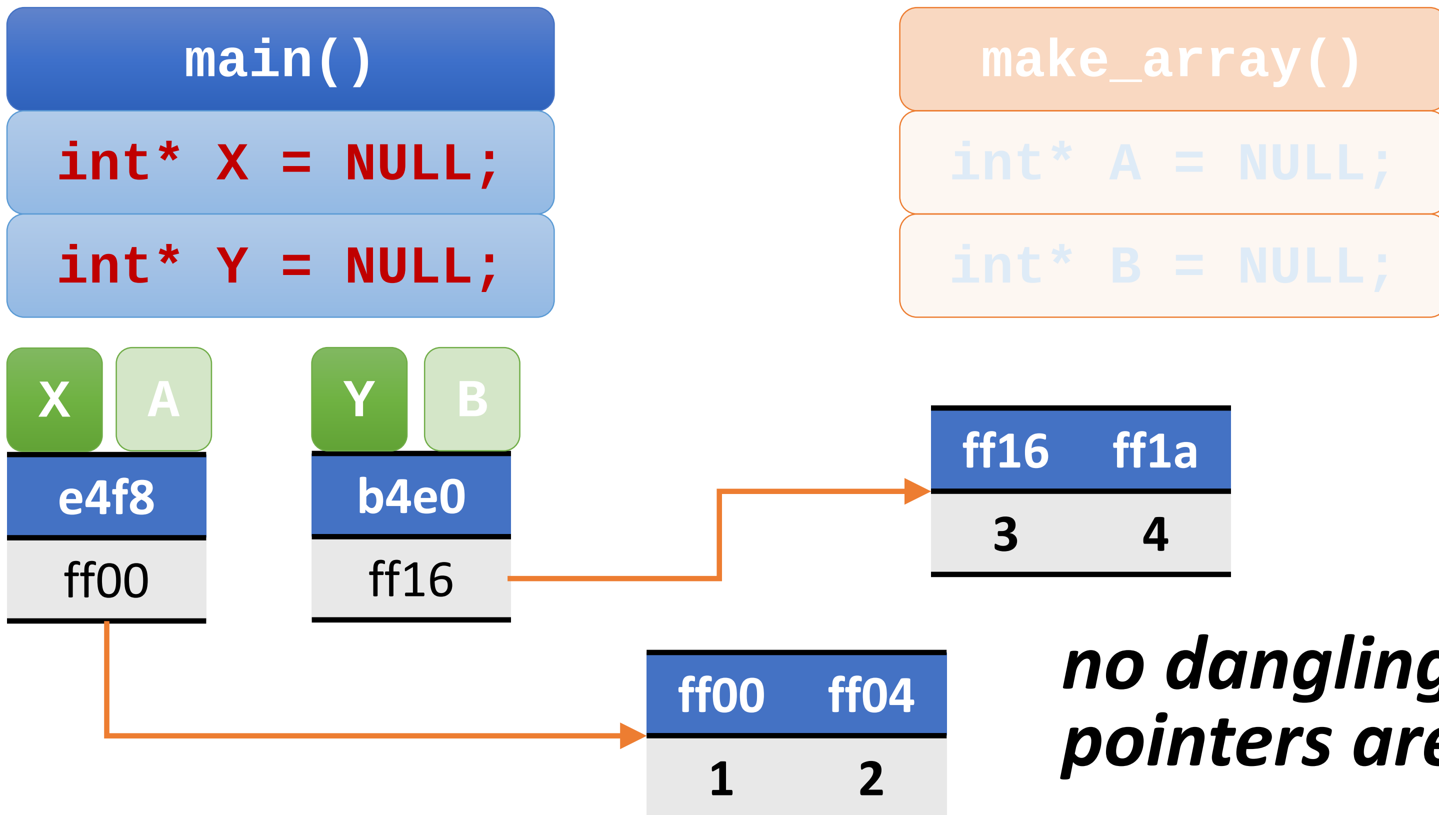

```
make_array()
int*& A = X;
int*& B = Y;
```

Pass X and Y by reference, instead of value

Heap allocations are pointed by multiple variables



After the function finishes..



no dangling pointers are left

Implementing this idea to a C++ code

```
#include <iostream>
using namespace std;
void make_arrays(int*& A, int*& B) {
  A = new int[2]; A[0] = 1; A[1] = 2; // create a new array in heap
  B = new int[2]; B[0] = 3; B[1] = 4; // create a new array in heap
}
int main() {  // main function – does not need arguments
  int *X, *Y; // declare pointer to be modified soon
  make_arrays(X, Y); // X, Y now point newly allocated variables
  cout << “X : “ << X[0] << “ “ << X[1] << endl;
  cout << “Y : “ << Y[0] << “ “ << Y[1] << endl;
  // because X and Y are allocated in heap,
  // make sure to delete them explicitly after use
  delete [] X;
  delete [] Y;
  return 0; // returning zero means normal termination
}
```

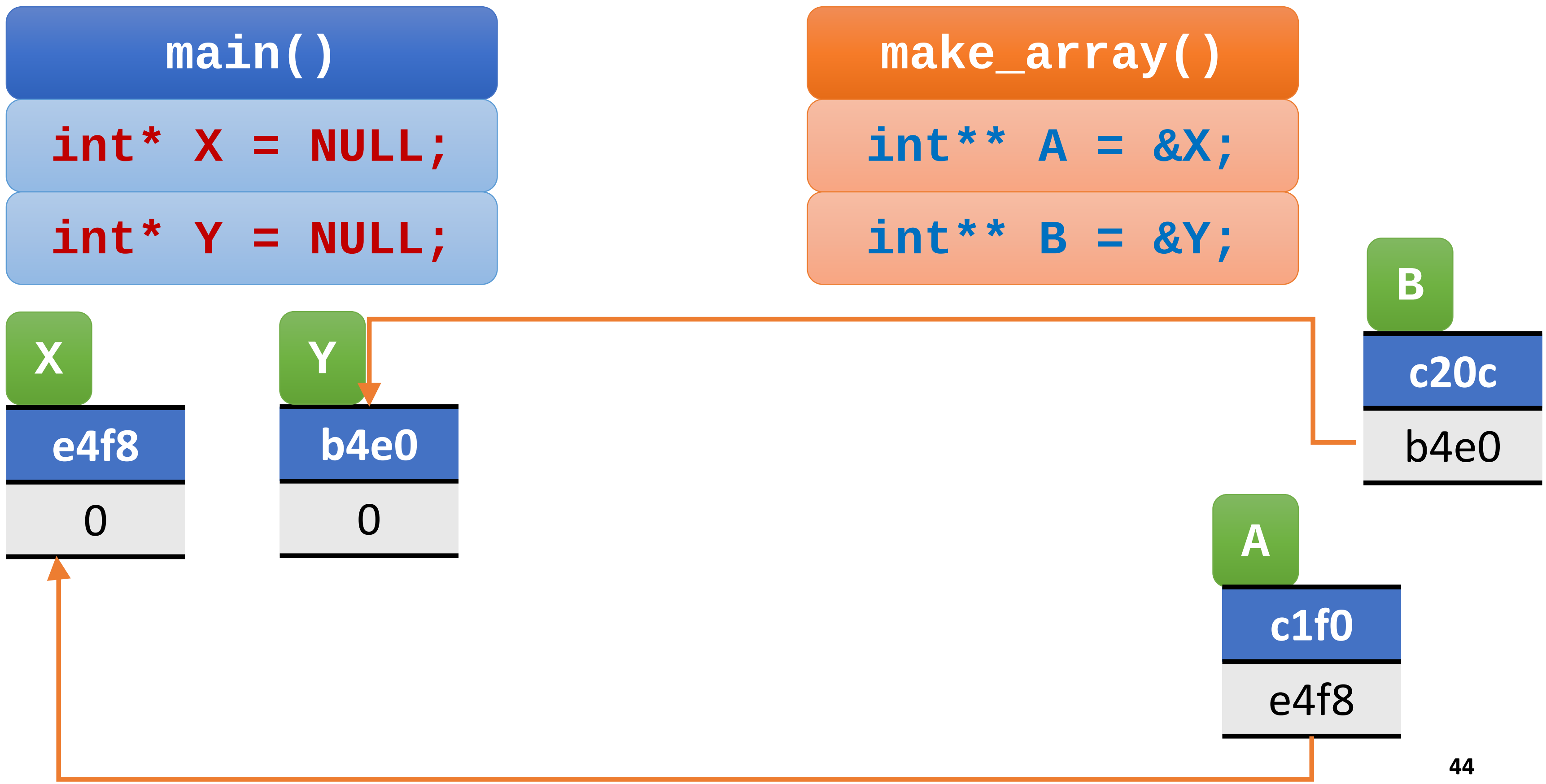
Another right way to fix : pass by pointer

main()

```
int* X = NULL;
int* Y = NULL;
```

X	Y
e4f8	b4e0
0	0

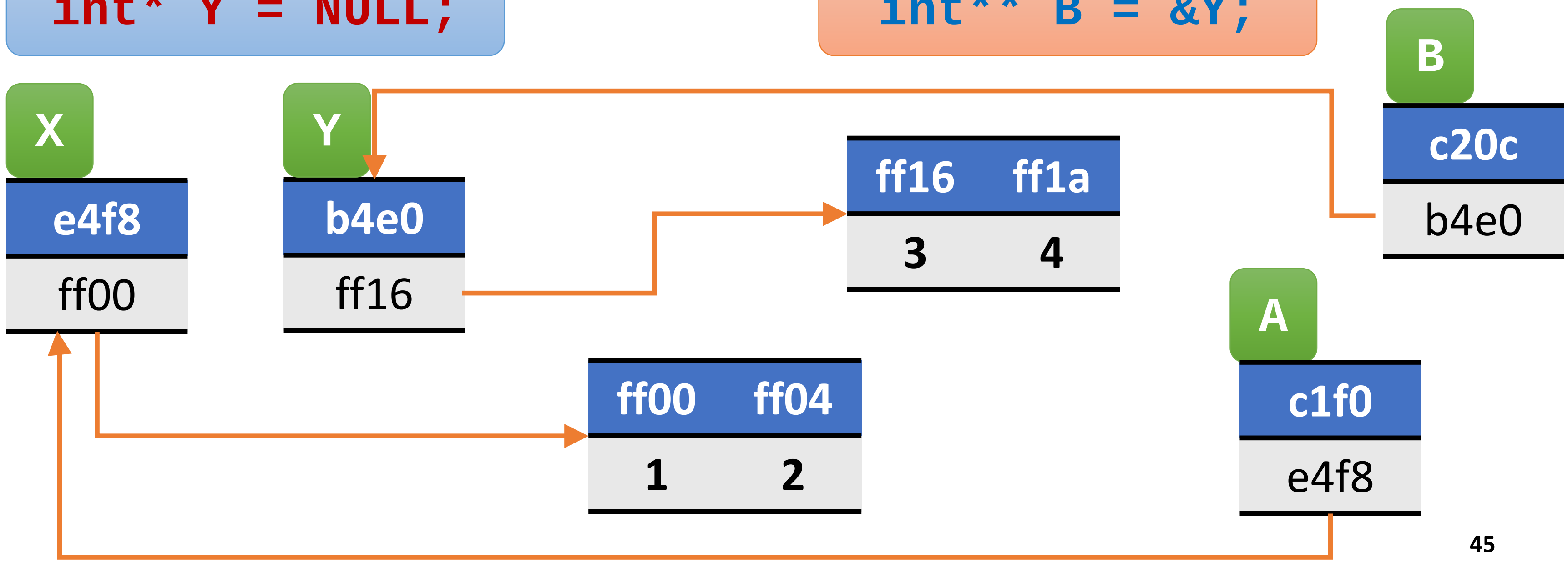
A and B are now pointing to an empty array



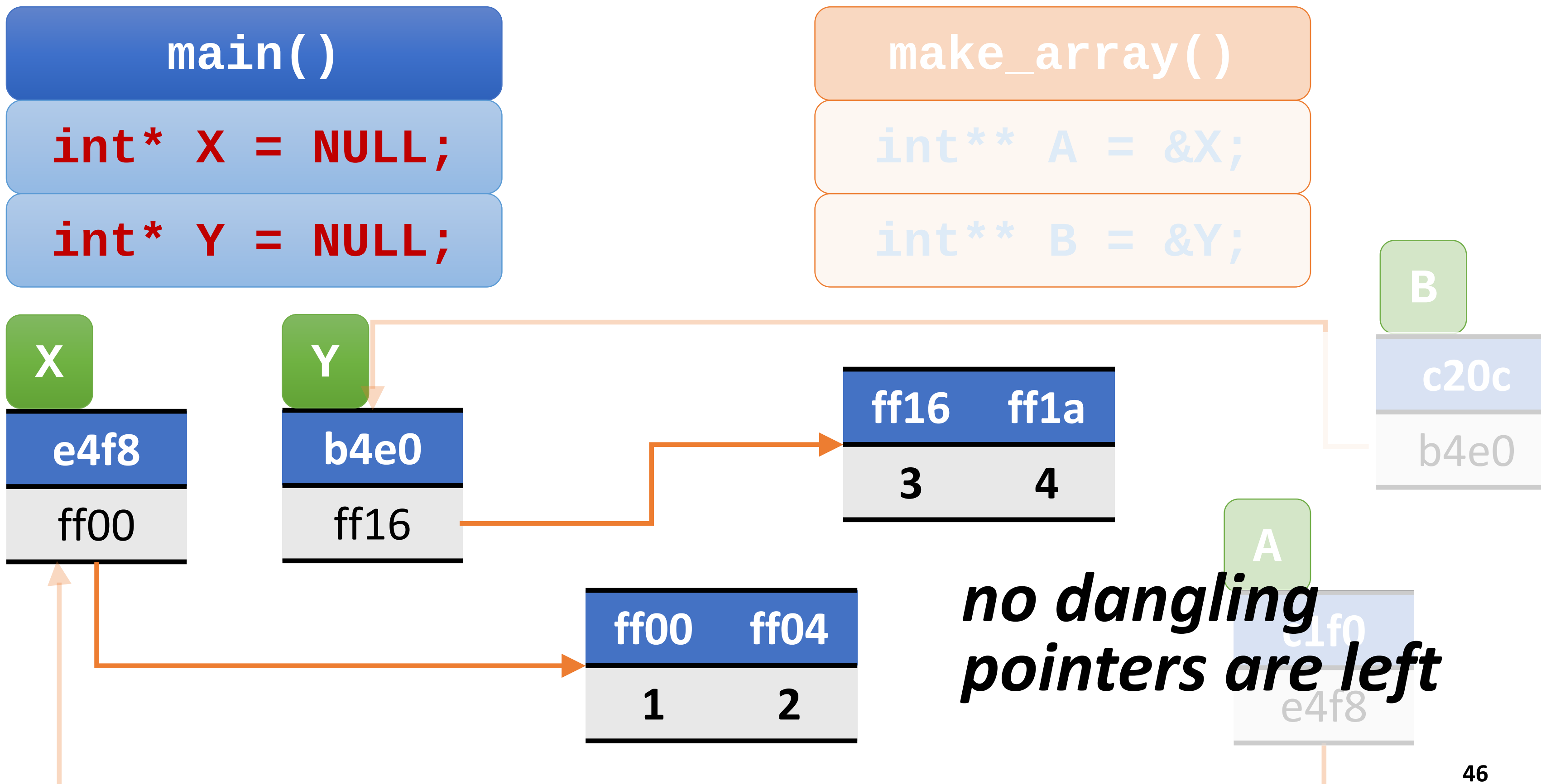
New arrays are allocated and their addresses are stored to the pointees of A, B

```
main()
int* X = NULL;
int* Y = NULL;
```

```
make_array()
int** A = &X;
int** B = &Y;
```



No leaks in heap after the function finishes.



no dangling pointers are left

The code is slightly more complicated..

```
#include <iostream>
using namespace std;
void make_arrays(int** A, int** B) {
  *A = new int[2]; (*A)[0] = 1; (*A)[1] = 2;
  *B = new int[2]; (*B)[0] = 3; (*B)[1] = 4;
}
int main() {  // main function – does not need arguments
  int *X, *Y; // declare pointer to be modified soon
  make_arrays(&X, &Y); // X, Y now point newly allocated variables
  cout << “X : “ << X[0] << “ “ << X[1] << endl;
  cout << “Y : “ << Y[0] << “ “ << Y[1] << endl;
  // because X and Y are allocated in heap,
  // make sure to delete them explicitly after use
  delete [] X;
  delete [] Y;
  return 0; // returning zero means normal termination
}
```

To summarize so far..

- **Variables defined in a function typically are stored in stack.**
  - And they are destroyed after the function finishes.
  - This sometimes may produce dangling pointers (especially when shallow-copied).
- **Variables allocated with “new” uses the heap space**
  - And they’re never destroyed until told to be.
  - This allows child functions to create something and pass to their parents.
  - Explicit management of memory is necessary (Use delete or delete[] to explicitly destroy)

Garbage collection in python and R

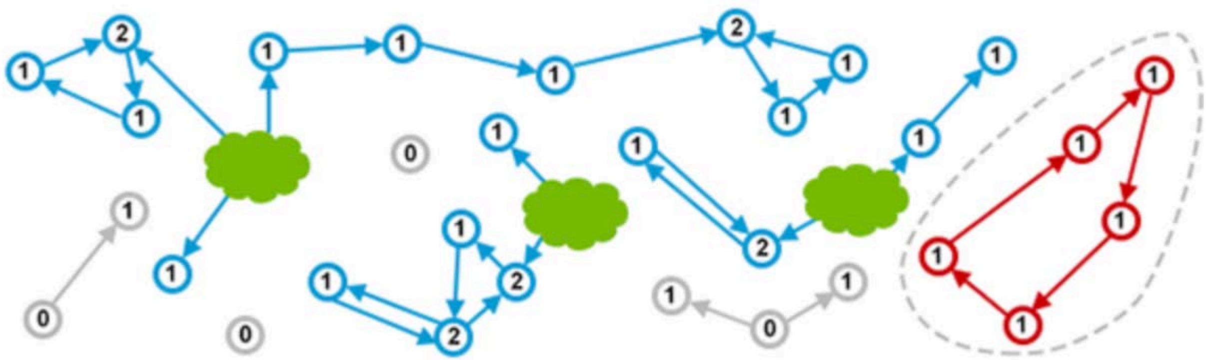
- **As you know, in python and R, you don't need to care about these memory management stuffs.**

- **As the language interpreter is supposed to "take care of" reclaiming the memory space of unused objects**
  - This is called "garbage collection"

- **Compared to explicit memory management, this approach is more convenient, but not more efficient**

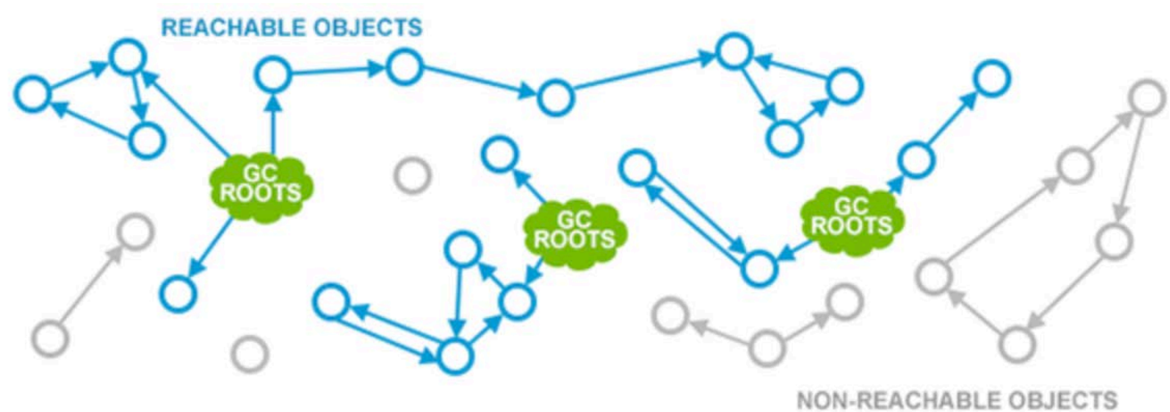
A useful overview on garbage collection

https://plumbr.eu/blog/garbage-collection/what-is-garbage-collection



Reference counting

Mark-and-sweep



Hyun Min Kang hmkang@umich.edu

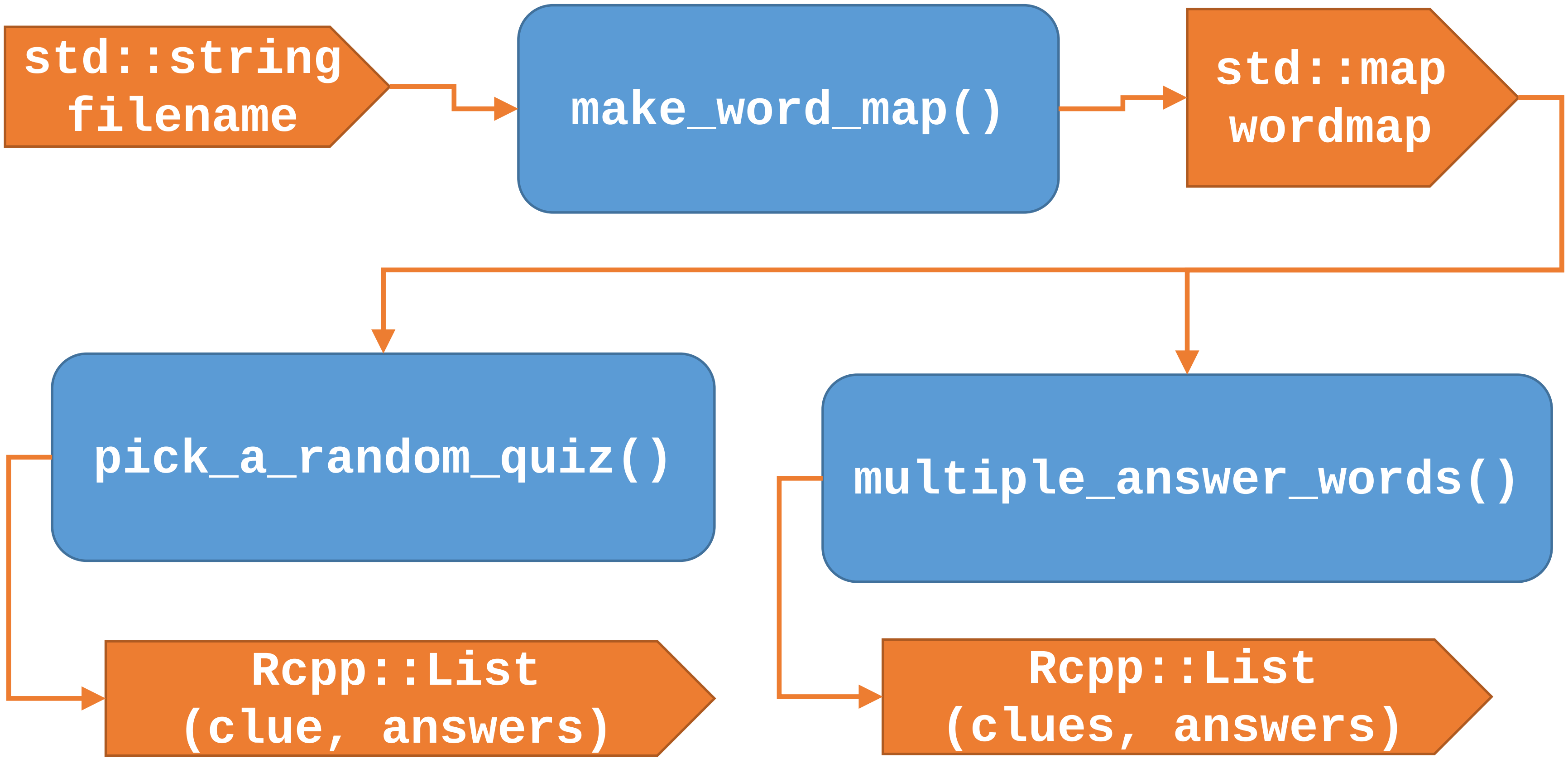
R/C++ communication with heap

- Typically function with `[[Rcpp::export]]` should return a data type recognized by R, such as
  - `NumericVector`
  - `NumericMatrix`
  - `StringVector`
  - `List`
  - `...`
- What if I want to return something that are not of R-compatible data type, such as `std::map`, `graph`, or other user-defined classes?

Some Rcpp functions for word unscrambler

- `make_word_map()`
  - Given : Name of file storing list of words
  - Return : STL map for word unscrambler
- `pick_a_random_quiz()`
  - Given : The STL map created from `make_word_map()`
  - Return : A list with randomly picked word (as clue) and all possible answer (as ans)
- `multiple_answer_words()`
  - Given : The STL map created from `make_word_map()`
    A threshold of the number of possible answers
  - Return : List of all possible (key / answers) that has multiple ways to unscramble words above the threshold

The structure between functions



Starting the implementation..

```
#include <Rcpp.h>
#include <string>
#include <vector>
#include <fstream>
#include <algorithm>
#include <map>

using namespace Rcpp;
using namespace std;

typedef map< string,vector<string> > s2vs_t;
typedef map< string,vector<string> >::iterator s2vs_it_t;
```

Use **`typedef`** to define a (short) nickname of a (long) data type

To return a pointer to a C++ object allocated in the heap, use Rcpp::XPtr<T>

```
// [[Rcpp::export]]
XPtr<s2vs_t> make_word_map(string filename) {
  ifstream ifs(filename.c_str());
  string s;
  s2vs_t* p = new s2vs_t;
  while( ifs >> s ) {
    string q = s;
    sort(q.begin(), q.end()); // lexicographical ordering
    (*p)[q].push_back(s);
  }
  return XPtr<s2vs_t>(p);
}
```

Returns the pointer of newly created object

create a new object

Same as before except that **p** had to be dereferenced

Need to use XPtr to returning an external pointer

To utilize the pointer for another function..

Pass the external pointer by value (Note that pass-by-ref with Rcpp does not work)

```
// [[Rcpp::export]]
List pick_a_random_quiz(XPtr<s2vs_t> p) {
  s2vs_it_t it = p->begin();
  advance(it, rand() % p->size());
  StringVector sv(it->second.size()+1);
  string q = it->first;
  random_shuffle(q.begin(), q.end());
  return List::create(Named("clue")=q,Named("ans")=it->second);
}
```

An inefficient way to sample from map randomly

Returning List is an easy way to return (key,value)-like data

Example Output

```
ptr <- make_word_map("nltk.235886.words.txt")
pick_a_random_quiz(ptr)
```

```
$clue
[1] "itecetar"

$ans
[1] "ceratite"
```

```
pick_a_random_quiz(ptr)
```

```
$clue
[1] "bndaa"

$ans
[1] "badan" "banda"
```

Another function to find words with many possible answers

```
// [[Rcpp::export]]
List multiple_answer_words(XPtr<s2vs_t> p, int num_ties) {
  List ret;
  for(s2vs_it_t it = p->begin(); it != p->end(); ++it) {
    if ( it->second.size() >= num_ties )
        ret.push_back(List::create(Named("key")=it->first,
                                   Named("words")=it->second));
  }
  return ret;
}
```

The list to return contains many lists

Example output

```
multiple_answer_words(ptr,8)
```

```
[[1]]
[[1]]$key
[1] "acert"

[[1]]$words
[1] "caret" "carte" "cater" "crate" "creat" "creta" "react" "recta" "trace"


[[2]]
[[2]]$key
[1] "aelpt"

[[2]]$words
[1] "leapt" "palet" "patel" "pelta" "petal" "plate" "pleat" "tepal"
```

Summary – Heap usage with Rcpp

- A new object can be allocated within the heap space, and its pointer can be returned using `Rcpp::XPtr<T>`
- The object may not be directly used in R function, but can be passed onto any C++ function through Rcpp.
- The garbage collection algorithm will automatically destroy the object when the object is not in use, but it is also possible to explicitly destroy the object before garbage collection happens

Reading Material

- [RK pp. 42-49] Using arrays and pointers
- [RK pp. 49-54] Functions