Heap-Allocated Arrays

Computer Science Fundamentals II

Jim Fix

Fall 2020, Lecture 03-2

Lab 03

Lab 03 looked at stack-allocated structs and arrays.

➡ (Demo solutions in terminal.)

➡ I'll post my solutions on-line

Today: we'll look at heap-allocated arrays.

Recall: Array Syntax

➡ Declaration of an array variable, with stack allocation.

statement ::=

***type var-name*[*int-literal*];**
***type var-name*[*int-literal*] = {*initializer-list*};**
***type var-name*[] = {*initializer-list*};**

➡ Assignment of an element:

statement ::= ***var-name*[*int-expression*] = *expression*;**

➡ Access to an element's value:

expression ::= ***var-name*[*int-expression*]**

➡ Passing as a parameter to a function/procedure:

proc-func-defn ::=

***type-or-void name*(..., *type** *var-name*, ...) {*block*}**

Example: stats.cc

```
double meanValue(double* array, int length) { ... }
double minValue(double* array, int length) { ... }

int main() {
  double data[10];
  int maxSize = 10;
  int size;
  do {
    std::cout << "What's the size of your data set? ";
    std::cin >> size;
  } while (size < 1 || size > maxSize);
  for (int i=0; i < size; i++) {
    std::cout << "Enter entry number "<< i << ": ";
    std::cin >> data[i];
  }
  std::cout << "Here is what you entered:\n";
  for (int i=0; i < size; i++) {
    std::cout << "Entry " << i << ": " << data[i] << "\n";
  }
  std::cout << "The mean value is " << meanValue(data,size) << ".\n";
  std::cout << "The minimum is " << minValue(data,size) << ".\n";
}
```

Example: appendInto.cc

```
int seq1[10];
int seq2[10];
int seq3[20];
int sz1,sz2,sz3;

// Request the first sequence.
std::cout << "Enter the size of the first: ";
std::cin >> sz1;
  std::cout << "Enter its " << sz1 << " items:\n";
  for (int i=0; i<sz1; i++) {
    std::cin >> seq1[i];
  }
  // Request the second sequence.
...
  appendInto(seq1,sz1,seq2,sz2,seq3);
  sz3 = sz1+sz2;
...
  output(seq3,sz3);
```

Example: appendInto.cc

```
void appendInto(int* a1, int n1,
                int* a2, int n2,
                int* a) {
  int i =  0;
  // Copy the first into the front of `a`.
  for (int i1=0; i1<n1; i1++) {
    a[i] = a1[i1];
    i++;
  }
  // Copy the second into the end of `a`.
  for (int i2=0; i2<n2; i2++) {
    a[i] = a2[i2];
    i++;
  }
}
```

Example: appendInto.cc

```
void output(int* a, int n) {
  std::cout << "[" << a[0];
  for (int i=1; i<n; i++) {
    std::cout << "," << a[i];
  }
  std::cout << "]" << std::endl;
}
```

Stack Memory

Recall that:

- When a function is called:
  - ➡ A stack frame is allocated. That frame holds the locations for its parameters and local variables.
- When that function returns:
  - ➡ Its stack frame is taken down. "De-allocated."

So far, we've allocated arrays and structs on the stack.

- Variable declarations like

```
int array[100];
CS2Student s;
```

  create storage on the stack to hold their data.
- That storage is implicitly deallocated upon **`return`**.

➡ **The lifetime of stack-allocated data is the same as the function's.**

Heap Memory

There is a separate region of memory called the *heap*.

➡ Like the stack, it is managed by the C++ runtime system.

You can reserve chunks of memory in the heap to use for your data structures. Keyword is **`new`**.

➡ You can request a chunk to hold an array of data.

➡ You can request a chunk to hold a struct's data.

- Use the keyword **`new`** to allocate data on the heap.

You determine the lifetime of these reserved chunks.

➡ You explicitly release them back to the heap.

- Use the keyword **`delete`** to de-allocate heap data.

Heap Allocation

Rather than allocate arrays on the stack, we can ***allocate them on the heap***.

➡ We use the keyword **`new`**, like so:

```
double* data = new double[size];
int* seq3 = new int[sz1+sz2];
```

- This requests a chunk of bytes large enough to use that many double values and int values. The *heap* reserves this space for your use.
- It gives us back a pointer (type **`double*`** and **`int*`**) to that place in memory.
- The *lifetime* of each can be beyond the lifetime of its local variable, i.e. beyond the lifetime of that function's stack frame.

We explicitly ***de-allocate*** the memory using **`delete`**, like so:

```
delete [] data; // note the brackets
delete [] seq3;
```

This *releases*/*frees* that chunk of memory so it can be used freely by the heap in future uses of **`new`**.

Example: stats.cc

```
double meanValue(double* array, int length) { ... }
double minValue(double* array, int length) { ... }

int main() {
  int size;
  std::cout << "What's the size of your data set? ";
  std::cin >> size;
  double* data = new double[size];

  for (int i=0; i < size; i++) {
    std::cout << "Enter entry number "<< i << ": ";
    std::cin >> data[i];
  }
  std::cout << "Here is what you entered:\n";
  for (int i=0; i < size; i++) {
    std::cout << "Entry " << i << ": " << data[i] << "\n";
  }

  std::cout << "The mean value is " << meanValue(data,size) << ".\n";
  std::cout << "The minimum is " << minValue(data,size) << ".\n";
  delete [] data;
}
```

Example: stats.cc

```
double meanValue(double* array, int length) { ... }
double minValue(double* array, int length) { ... }

int main() {
  int size;
  std::cout << "What's the size of your data set? ";
  std::cin >> size;
  double* data = new double[size];

  for (int i=0; i < size; i++) {
    std::cout << "Enter entry number "<< i << ": ";
    std::cin >> data[i];
  }
  std::cout << "Here is what you entered:\n";
  for (int i=0; i < size; i++) {
    std::cout << "Entry " << i << ": " << data[i] << "\n";
  }

  std::cout << "The mean value is " << meanValue(data,size) << ".\n";
  std::cout << "The minimum is " << minValue(data,size) << ".\n";
  delete [] data;
}
```

Example: append.cc

```
// Request the first sequence.
std::cout << "Enter the size of the first: ";
int sz1;
std::cin >> sz1;
std::cout << "Enter its " << sz1 << " items:\n";
int* seq1 = new int[sz1];
for (int i=0; i<sz1; i++) {
  std::cin >> seq1[i];
  }
  // Request the second sequence.
...
  int* seq3 = append(seq1,sz1,seq2,sz2);
  int sz3 = sz1+sz2;
...
  output(seq3,sz3);
```

Example: append.cc

```
// Request the first sequence.
std::cout << "Enter the size of the first: ";
int sz1;
std::cin >> sz1;
std::cout << "Enter its " << sz1 << " items:\n";
int* seq1 = new int[sz1];
for (int i=0; i<sz1; i++) {
  std::cin >> seq1[i];
  }
  // Request the second sequence.
...
  int* seq3 = append(seq1,sz1,seq2,sz2);
  int sz3 = sz1+sz2;
...
  output(seq3,sz3);
```

Example: append.cc

```
int* append(int* a1,int n1,int* a2,int n2) {
  int* a = new int[n1+n2];

  int i = 0;
  // Copy the first into the front of `a`.
  for (int i1=0; i1<n1; i1++) {
    a[i] = a1[i1];
    i++;
  }
  // Copy the second into the end of `a`.
  for (int i2=0; i2<n2; i2++) {
    a[i] = a2[i2];
    i++;
  }
  return a;
}
```

Example: append.cc

```
int* append(int* a1,int n1,int* a2,int n2) {
  int* a = new int[n1+n2];

  int i = 0;
  // Copy the first into the front of `a`.
  for (int i1=0; i1<n1; i1++) {
    a[i] = a1[i1];
    i++;
  }
  // Copy the second into the end of `a`.
  for (int i2=0; i2<n2; i2++) {
    a[i] = a2[i2];
    i++;
  }
  return a;
}
```

Example: append.cc

```
int* append(int* a1,int n1,int* a2,int n2) {
  int* a = new int[n1+n2];

  int i = 0;
  // Copy the first into the front of `a`.
  for (int i1=0; i1<n1; i1++) {
    a[i] = a1[i1];
    i++;
  }
  // Copy the second into the end of `a`.
  for (int i2=0; i2<n2; i2++) {
    a[i] = a2[i2];
    i++;
  }
  return a;
}
```

Example: append.cc

```
// Request the first sequence.
std::cout << "Enter the size of the first: ";
int sz1;
std::cin >> sz1;
std::cout << "Enter its " << sz1 << " items:\n";
int* seq1 = new int[sz1];
for (int i=0; i<sz1; i++) {
  std::cin >> seq1[i];
  }
  // Request the second sequence.
...
  int* seq3 = append(seq1,sz1,seq2,sz2);
  int sz3 = sz1+sz2;
...
  output(seq3,sz3);
```

Example: append.cc

```
// Request the first sequence.
std::cout << "Enter the size of the first: ";
int sz1;
std::cin >> sz1;
std::cout << "Enter its " << sz1 << " items:\n";
int* seq1 = new int[sz1];
for (int i=0; i<sz1; i++) {
  std::cin >> seq1[i];
}
// Request the second sequence.
...
  int *seq3 = append(seq1,sz1,seq2,sz2);
  int sz3 = sz1+sz2;
...
  output(seq3,sz3);
```

Example: append.cc

```
// Request the first sequence.
std::cout << "Enter the size of the first: ";
int sz1;
std::cin >> sz1;

int* seq1 = requestElements(sz1);

  // Request the second sequence.
...
  int* seq3 = append(seq1,sz1,seq2,sz2);
  int sz3 = sz1+sz2;
...
  output(seq3,sz3);
```

Example: append.cc

```
int* requestElements(int sz) {
  std::cout << "Enter its " << sz << " items:\n";
  int* seq = new int[sz];
  for (int i=0; i<sz; i++) {
    std::cin >> seq[i];
  }
  return seq;
}
```

Revised append.cc

```
...
... // definitions of requestElements, append
...
int main(void) {
  std::cout << "Enter the size of the first: ";
  int sz1;
  std::cin >> sz1;
  int* seq1 = requestElements(sz1);
  std::cout << "Enter the size of the second: ";
  int sz2;
  std::cin >> sz2;
  int* seq2 = requestElements(sz2);
  int* seq3 = append(seq1,sz1,seq2,sz2);
  int sz3 = sz1+sz2;
  output(seq3,sz3);
  delete [] seq1;
  delete [] seq2;
  delete [] seq3;
}
```

Instrumented append.cc

```
...
... // definitions of requestElements, append
...
int main(void) {
  std::cout << "Enter the size of the first: ";
  int sz1;
  std::cin >> sz1;
  int* seq1 = requestElements(sz1);
  std::cout << "Enter the size of the second: ";
  int sz2;
  std::cin >> sz2;
  int* seq2 = requestElements(sz2);
  int* seq3 = append(seq1,sz1,seq2,sz2);
  int sz3 = sz1+sz2;
  output(seq3,sz3);
  // print the hexadecimal code of their locations
  std::cout << seq1 << std::endl;
  std::cout << seq2 << std::endl;
  std::cout << seq3 << std::endl;
  delete [] seq1; delete [] seq2; delete [] seq3;
}
```

main

seq1: 0x??????????

sz1: ??

seq2: 0x??????????

sz2: ??

seq3: 0x??????????

sz3: ??

the call stack

the heap

main

seq1:	0x??????????
sz1:	4
seq2:	0x??????????
sz2:	??
seq3:	0x??????????
sz3:	??

the call stack

the heap

requestElements

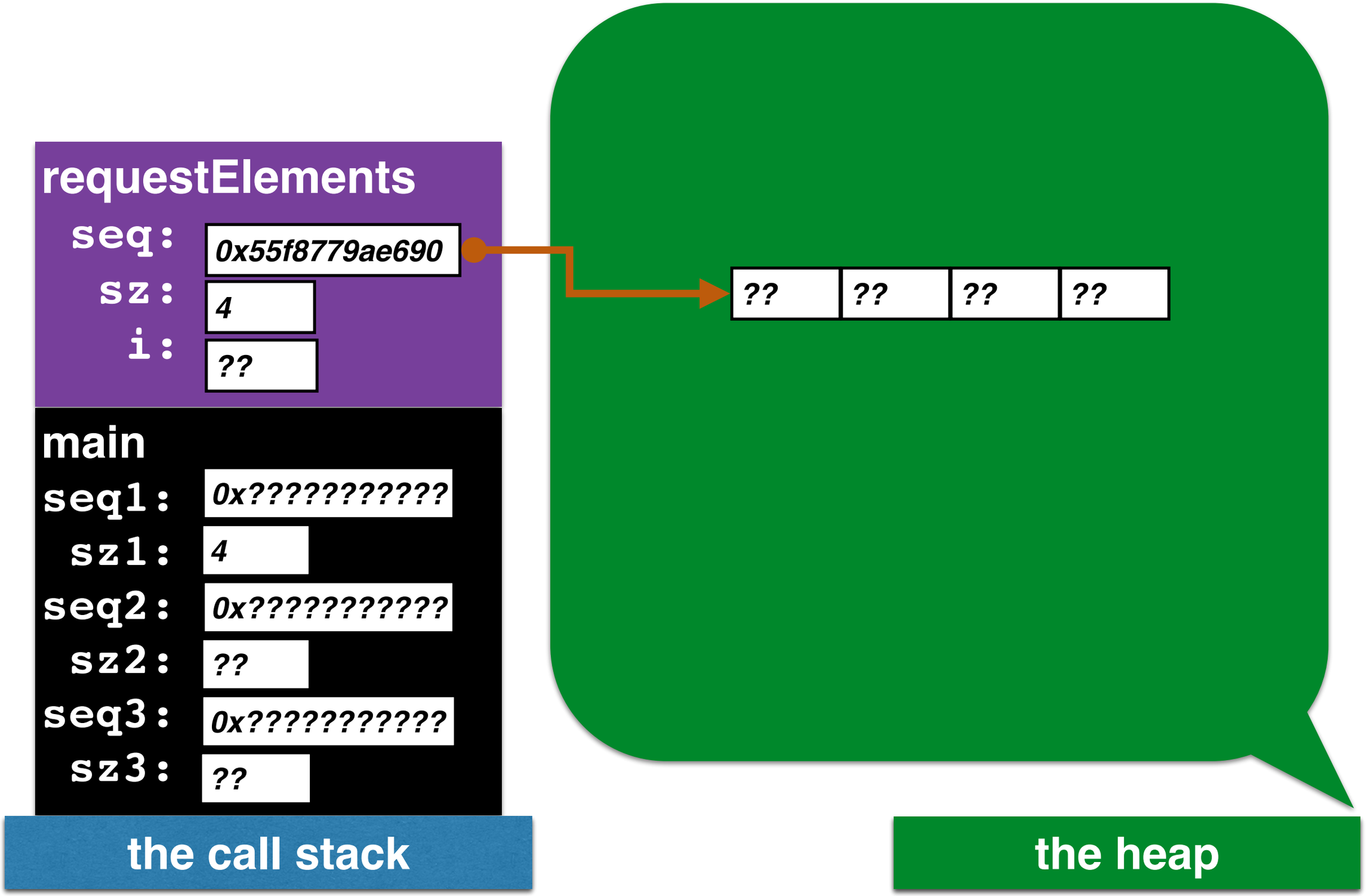
seq:	*0x??????????*
sz:	*4*
i:	*??*

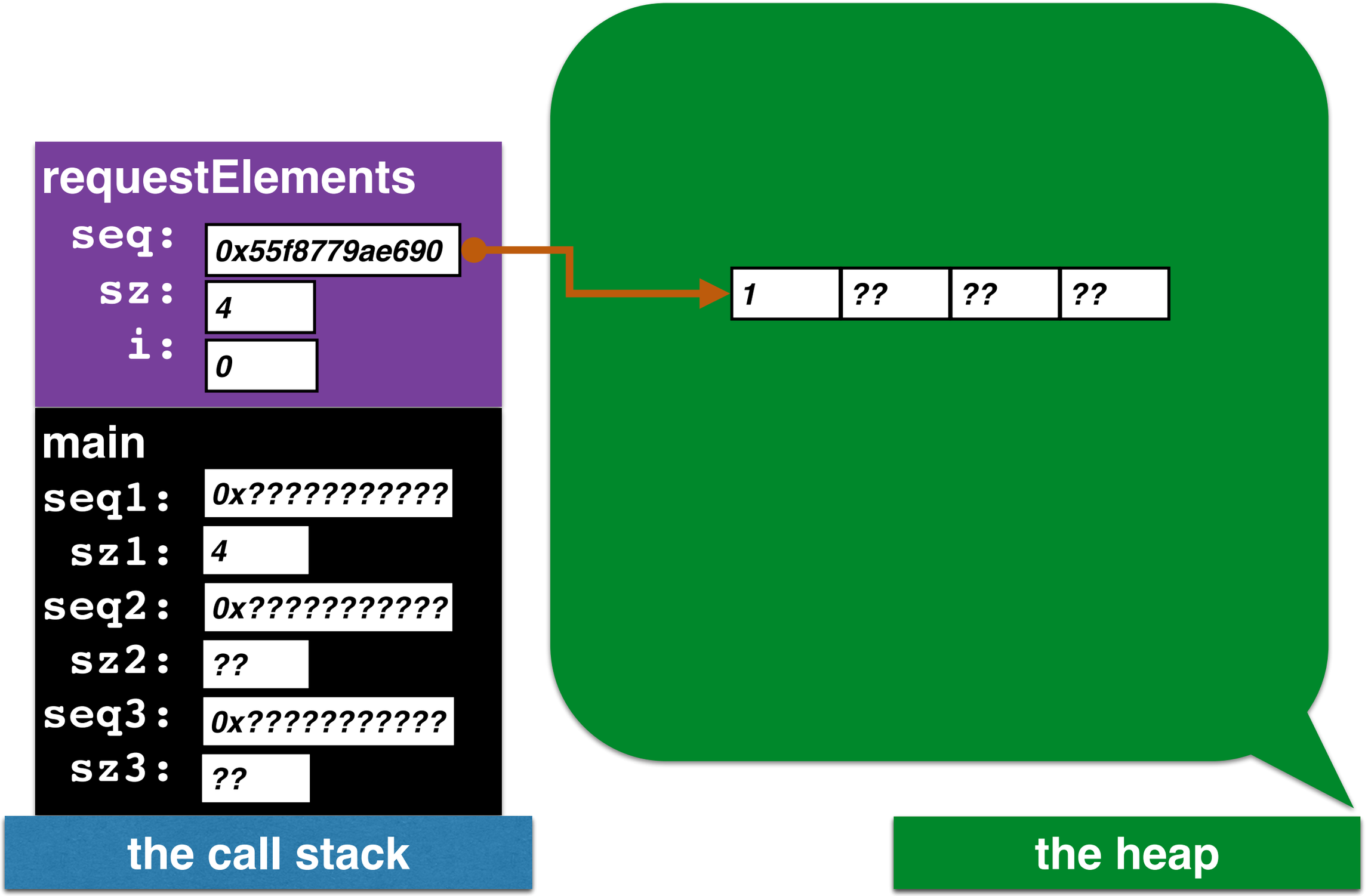
main

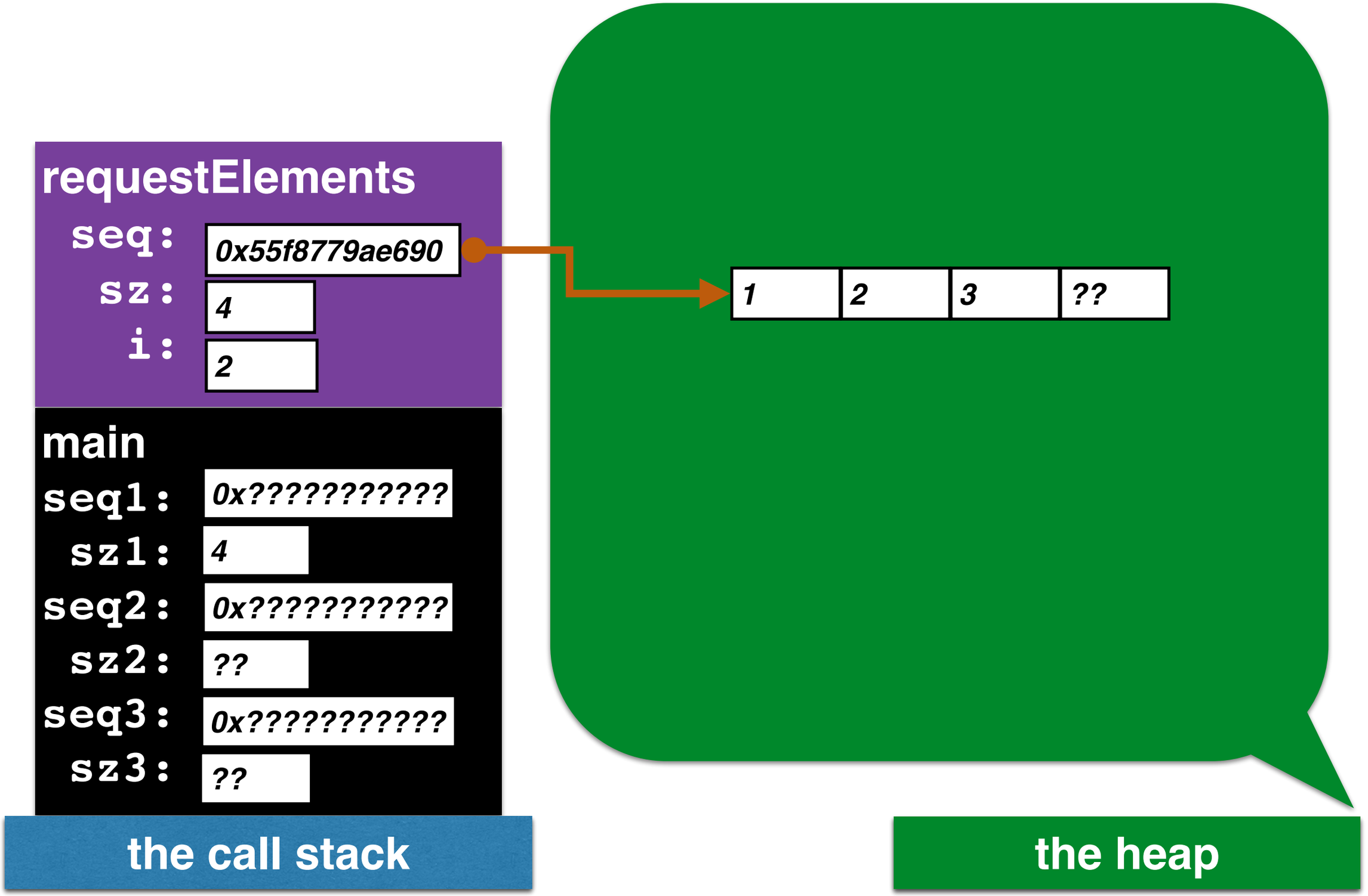
seq1:	*0x??????????*
sz1:	*4*
seq2:	*0x??????????*
sz2:	*??*
seq3:	*0x??????????*
sz3:	*??*

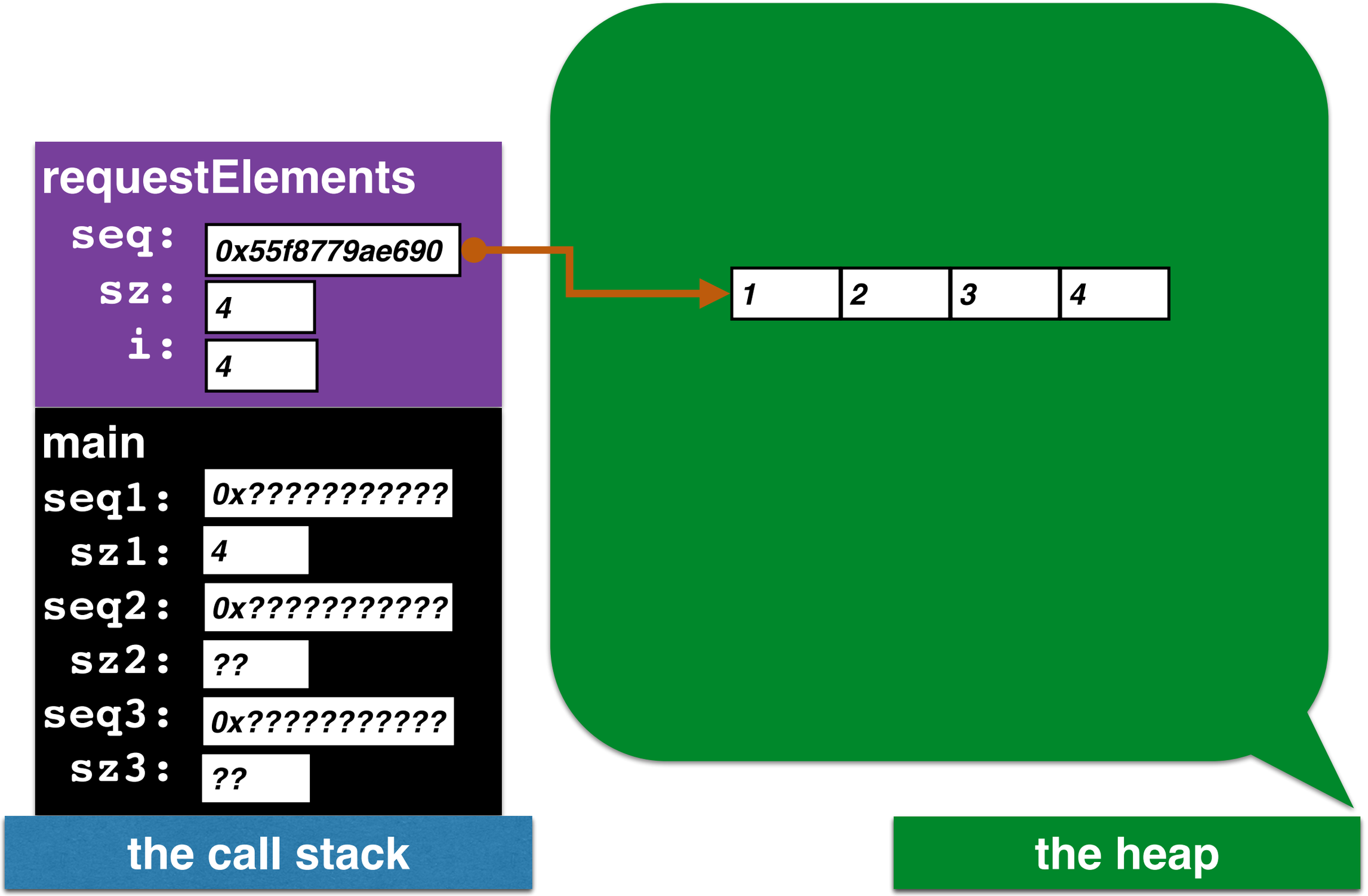
the call stack

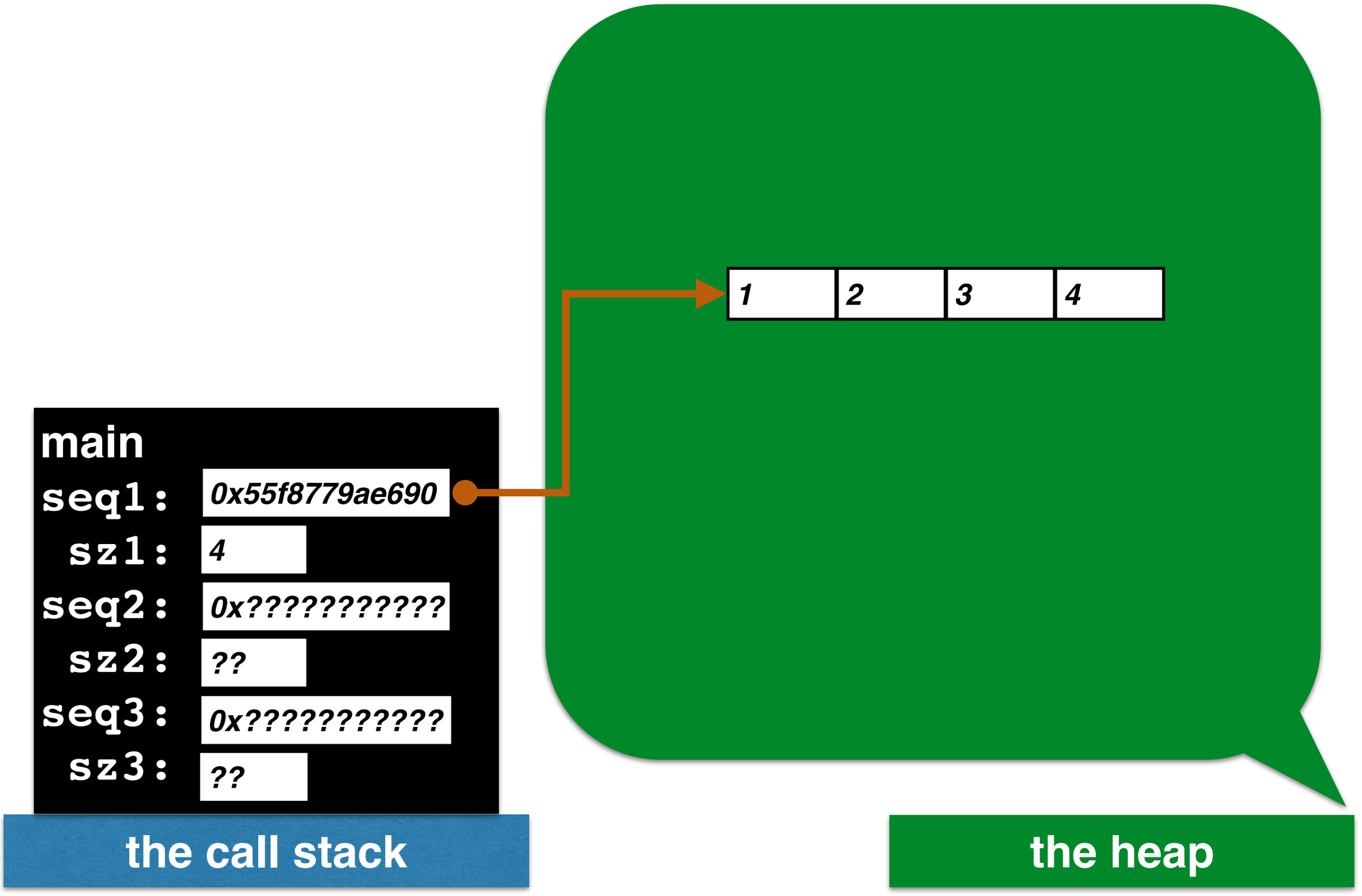
the heap

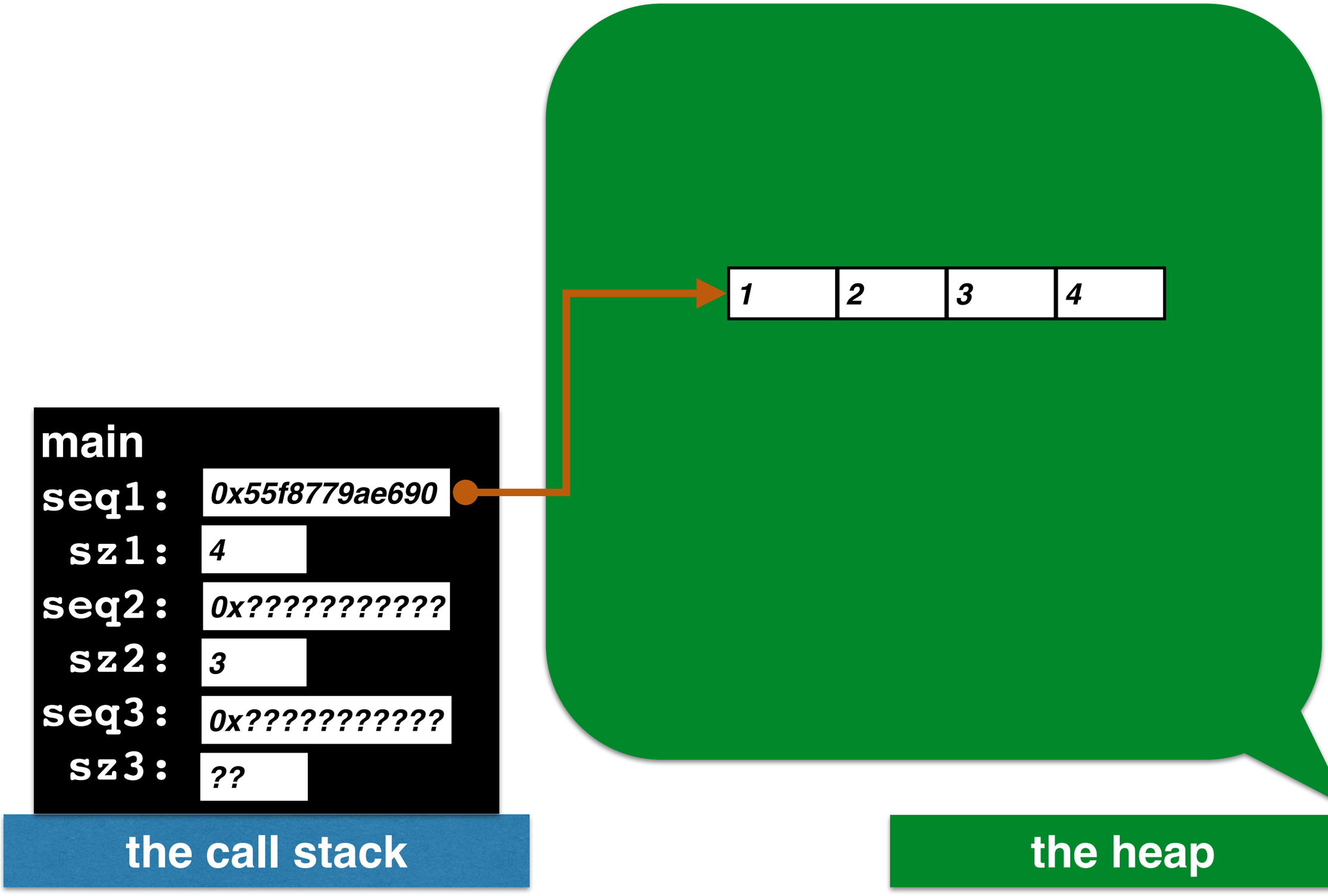

requestElements
seq:
0x55f8779ae690
sz:
4
i:
??
main
seq1:
0x??????????
sz1:
4
seq2:
0x??????????
sz2:
??
seq3:
0x??????????
sz3:
??
??
??
??
??
the call stack
the heap

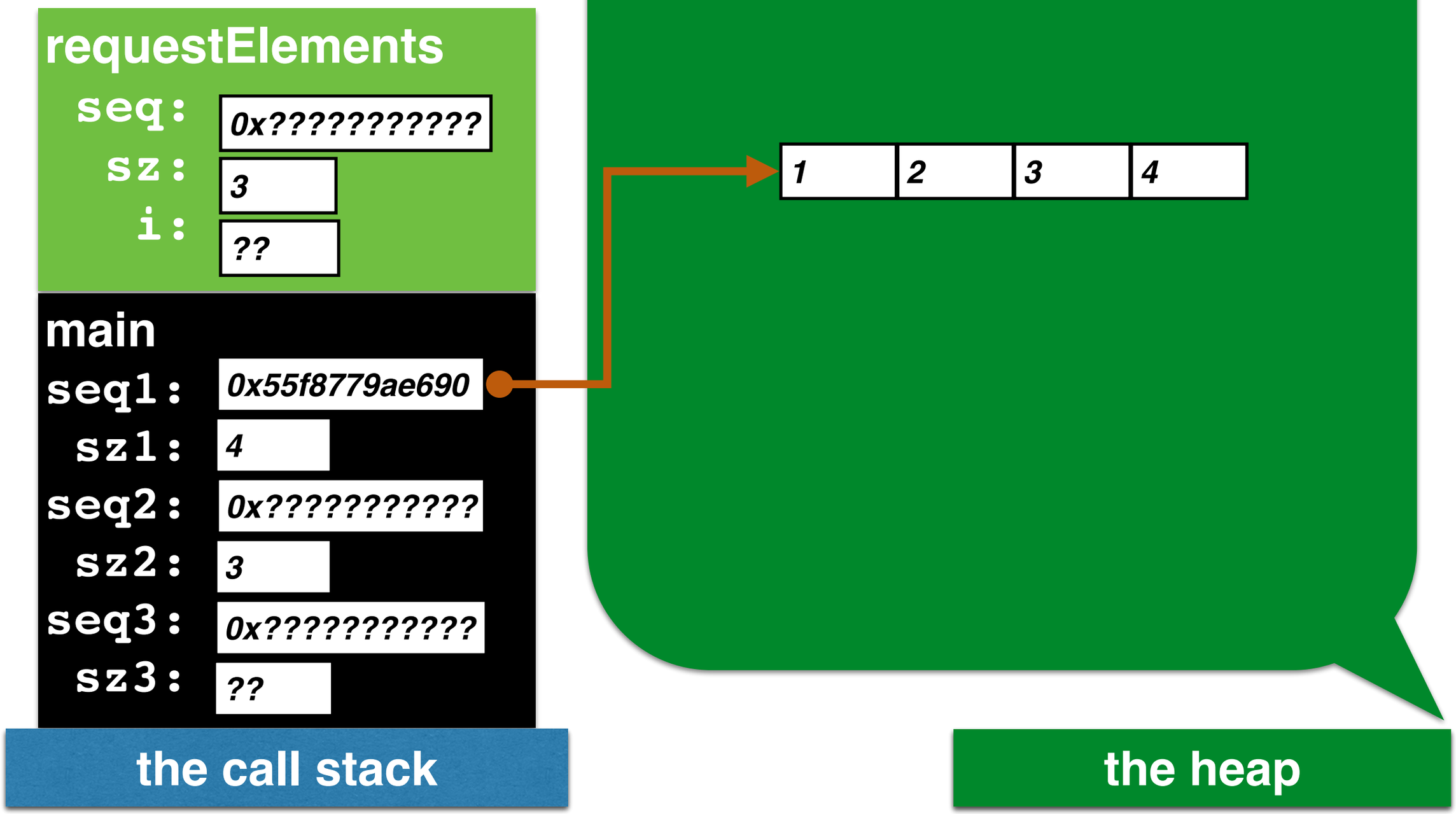

requestElements
seq:
0x55f8779ae690
sz:
4
i:
0
main
seq1:
0x??????????
sz1:
4
seq2:
0x??????????
sz2:
??
seq3:
0x??????????
sz3:
??
1
??
??
??
the call stack
the heap

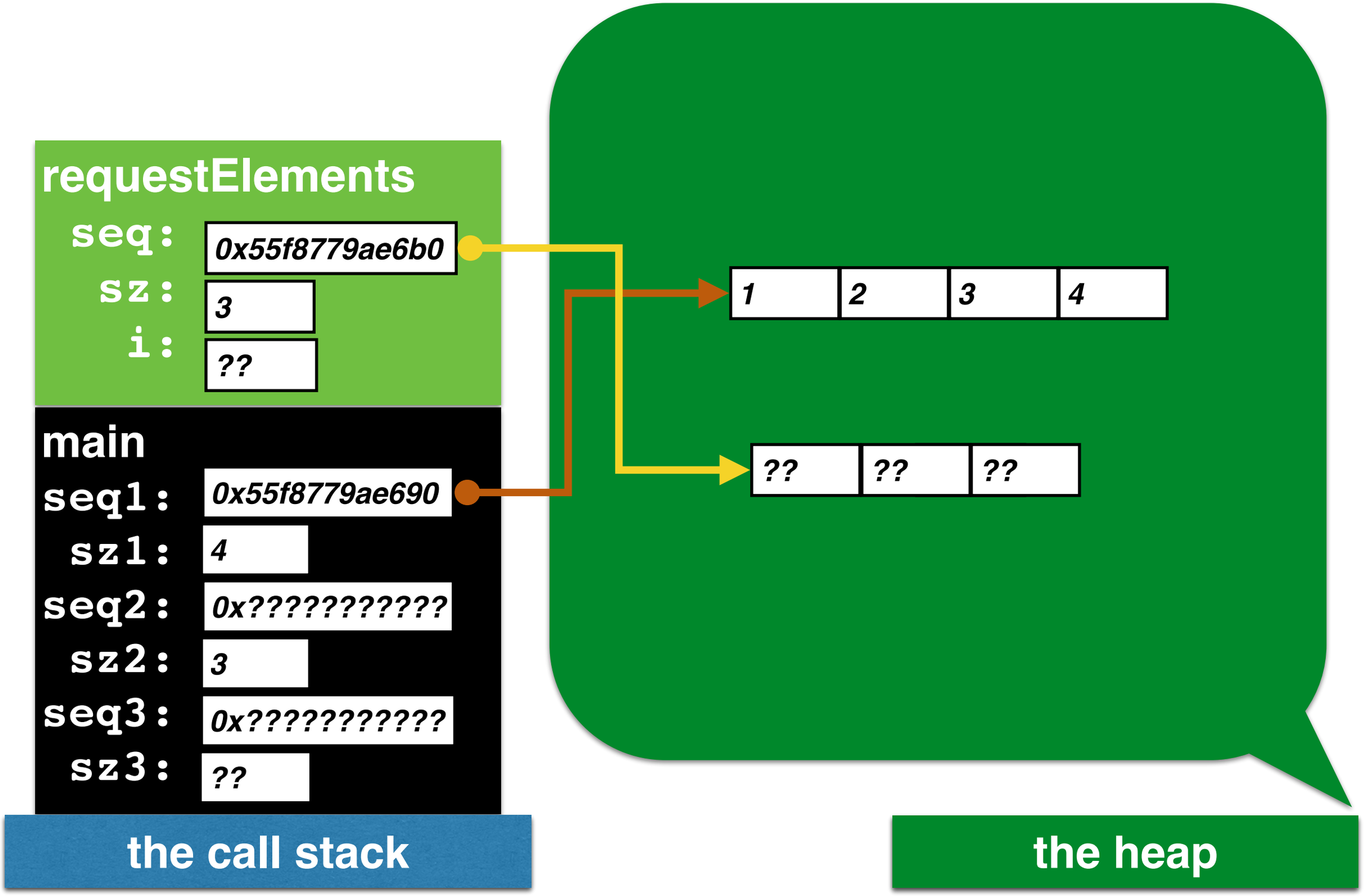

requestElements
seq:
0x55f8779ae690
sz:
4
i:
2
main
seq1:
0x???????????
sz1:
4
seq2:
0x???????????
sz2:
??
seq3:
0x???????????
sz3:
??
1
2
3
??
the call stack
the heap

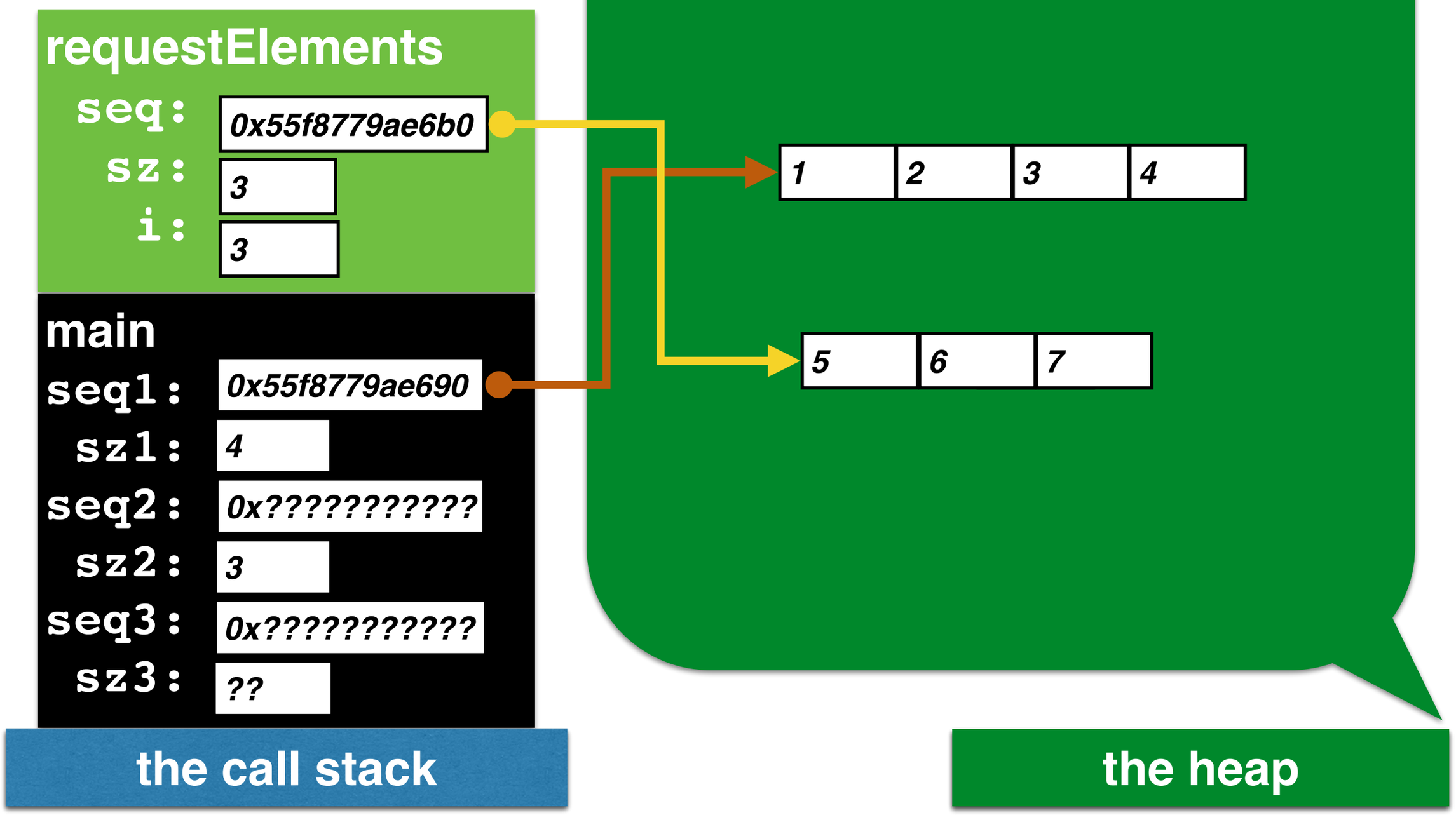

requestElements
seq:
0x55f8779ae690
sz:
4
i:
4
main
seq1:
0x???????????
sz1:
4
seq2:
0x???????????
sz2:
??
seq3:
0x???????????
sz3:
??
1
2
3
4
the call stack
the heap

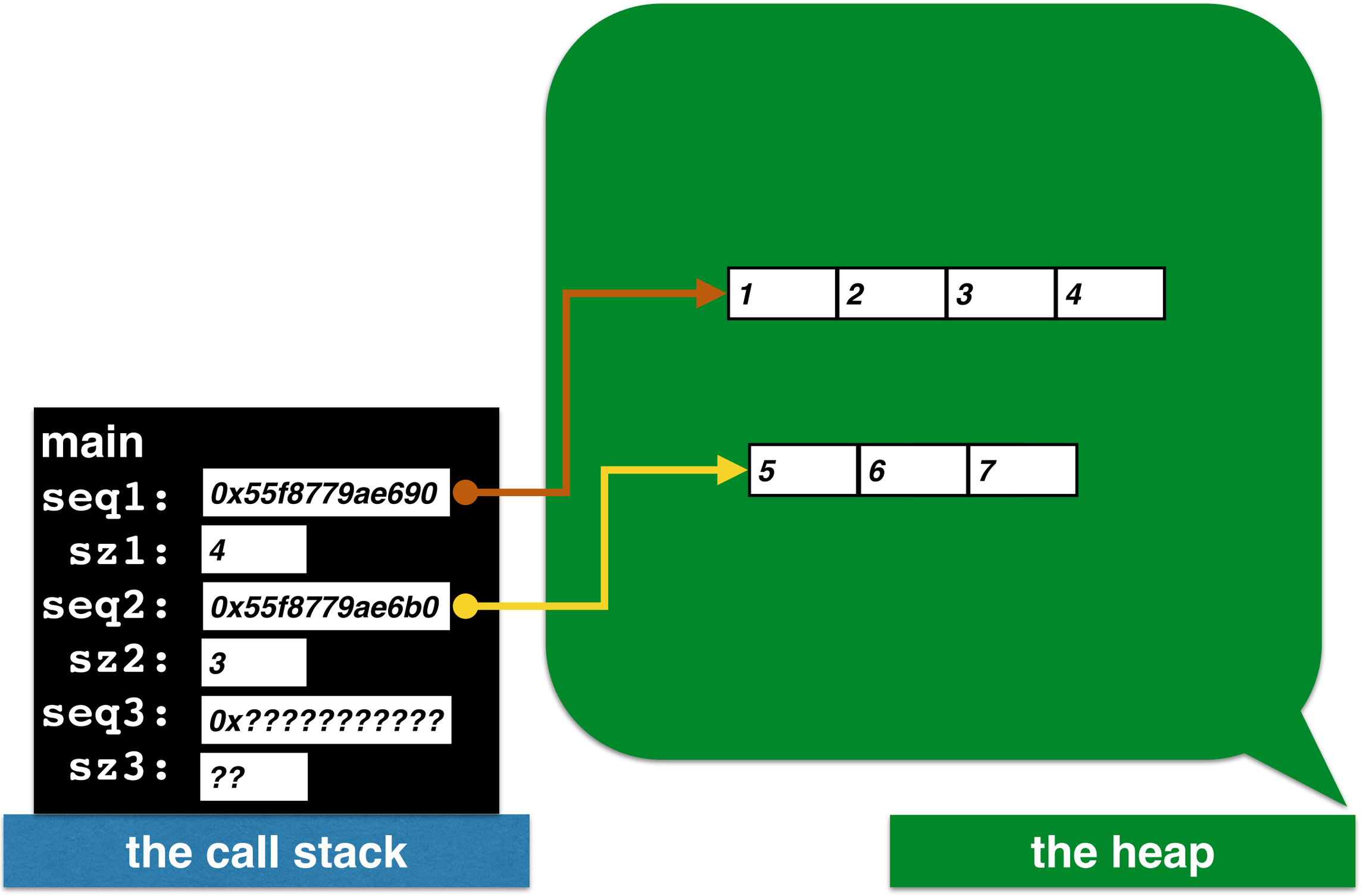

main
seq1:
0x55f8779ae690
sz1:
4
seq2:
0x???????????
sz2:
??
seq3:
0x???????????
sz3:
??
1
2
3
4
the call stack
the heap


main
seq1:
0x55f8779ae690
sz1:
4
seq2:
0x???????????
sz2:
3
seq3:
0x???????????
sz3:
??
1
2
3
4
the call stack
the heap


requestElements
seq:
0x???????????
sz:
3
i:
??
main
seq1:
0x55f8779ae690
sz1:
4
seq2:
0x???????????
sz2:
3
seq3:
0x???????????
sz3:
??
1
2
3
4
the call stack
the heap


requestElements
seq:
0x55f8779ae6b0
sz:
3
i:
??
main
seq1:
0x55f8779ae690
sz1:
4
seq2:
0x???????????
sz2:
3
seq3:
0x???????????
sz3:
??
1
2
3
4
??
??
??
the call stack
the heap


requestElements
seq:
0x55f8779ae6b0
sz:
3
i:
3
main
seq1:
0x55f8779ae690
sz1:
4
seq2:
0x???????????
sz2:
3
seq3:
0x???????????
sz3:
??
1
2
3
4
5
6
7
the call stack
the heap


main
seq1:
0x55f8779ae690
sz1:
4
seq2:
0x55f8779ae6b0
sz2:
3
seq3:
0x???????????
sz3:
??
1
2
3
4
5
6
7
the call stack
the heap

append

a1: 0x55f8779ae690

n1: 4

a2: 0x55f8779ae6b0

n2: 3

a: 0x???????????

main

seq1: 0x55f8779ae690

sz1: 4

seq2: 0x55f8779ae6b0

sz2: 3

seq3: 0x???????????

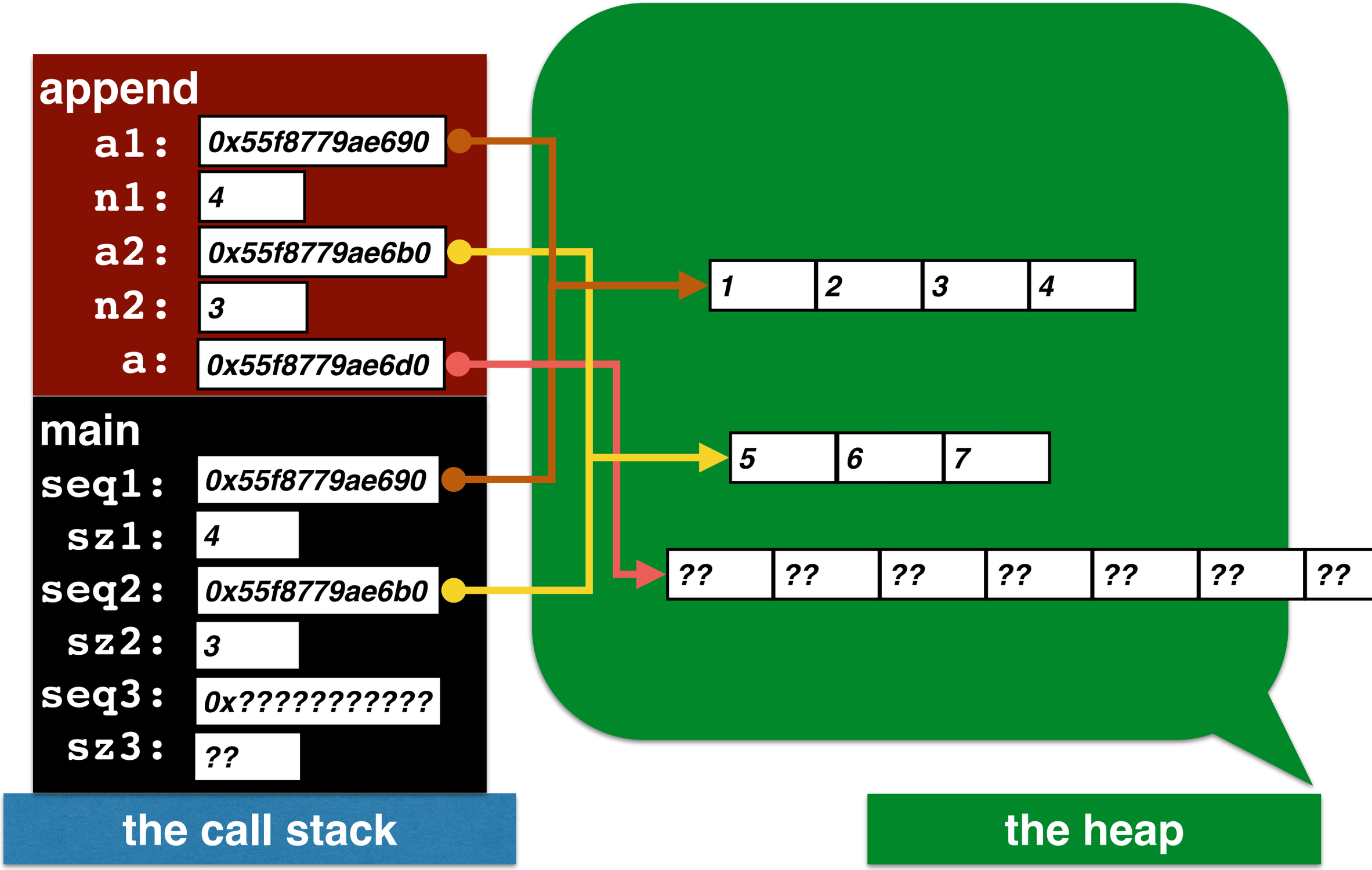
sz3: ??

the call stack

1	2	3	4

5	6	7

the heap


append
a1:
0x55f8779ae690
n1:
4
a2:
0x55f8779ae6b0
n2:
3
a:
0x55f8779ae6d0
main
seq1:
0x55f8779ae690
sz1:
4
seq2:
0x55f8779ae6b0
sz2:
3
seq3:
0x???????????
sz3:
??
1
2
3
4
5
6
7
??
??
??
??
??
??
??
the call stack
the heap

append

a1: 0x55f8779ae690

n1: 4

a2: 0x55f8779ae6b0

n2: 3

a: 0x55f8779ae6d0

main

seq1: 0x55f8779ae690

sz1: 4

seq2: 0x55f8779ae6b0

sz2: 3

seq3: 0x???????????

sz3: ??

the call stack

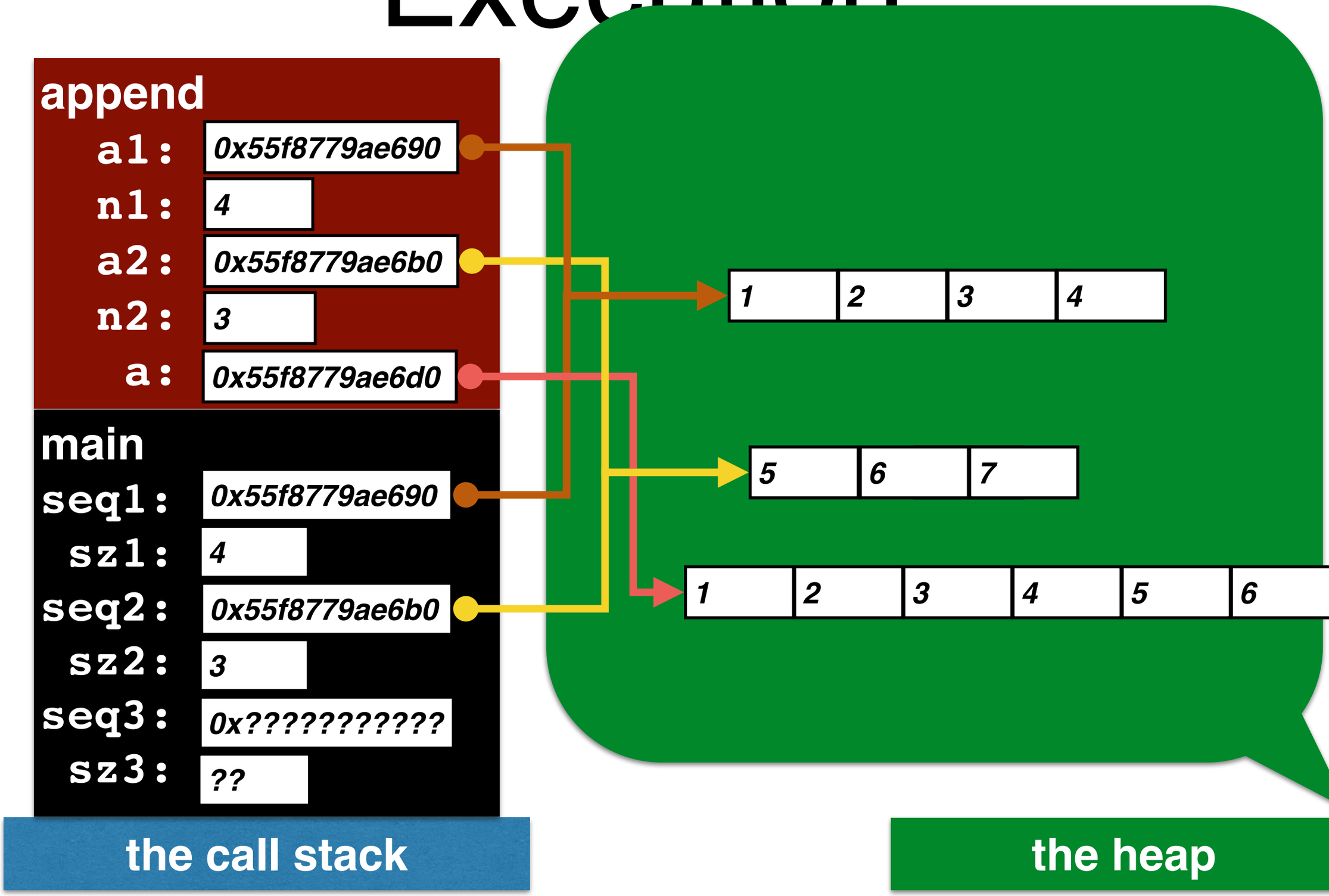
1 | 2 | 3 | 4

5 | 6 | 7

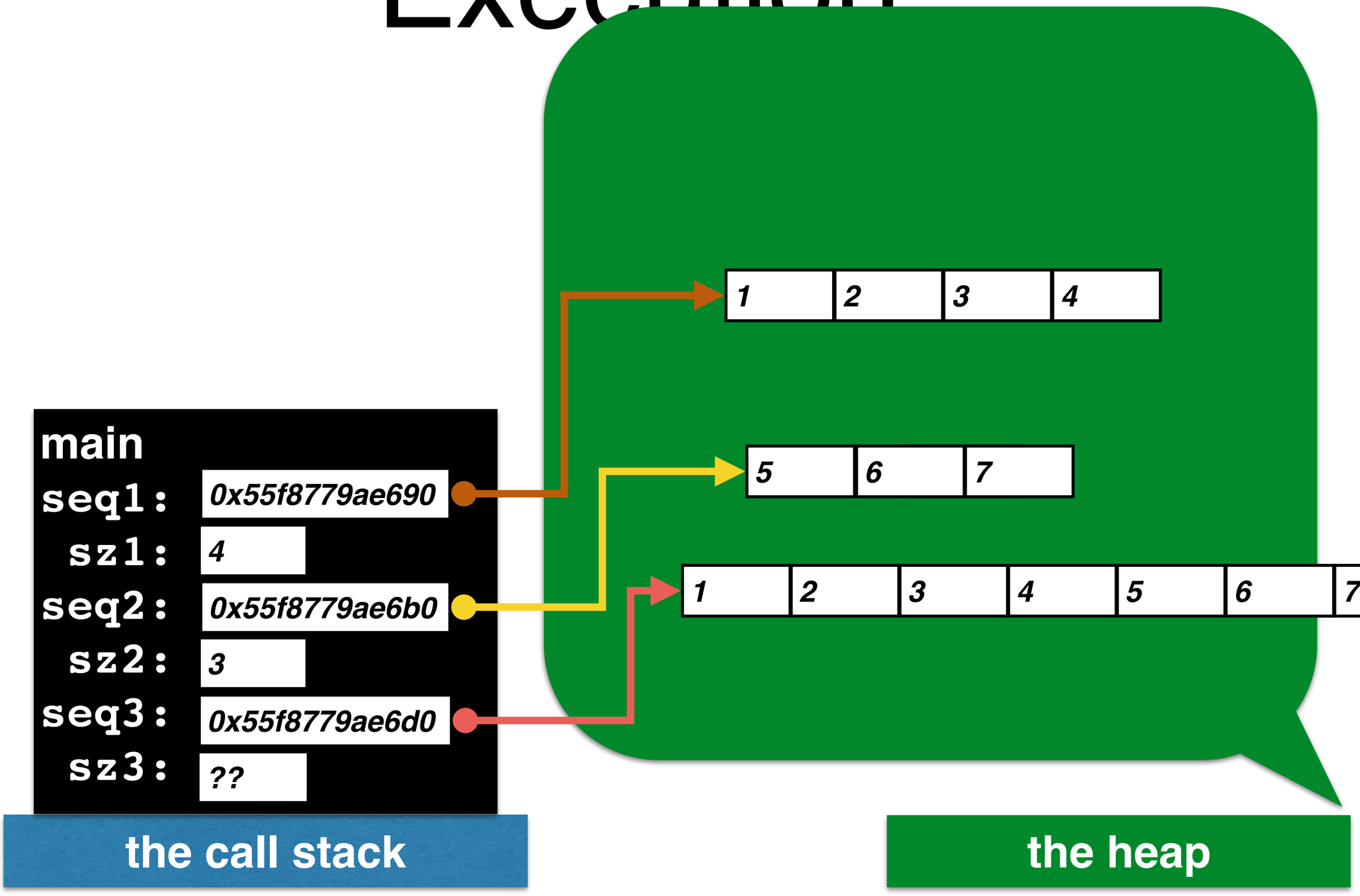
1 | 2 | 3 | 4 | ?? | ?? | ??

the heap

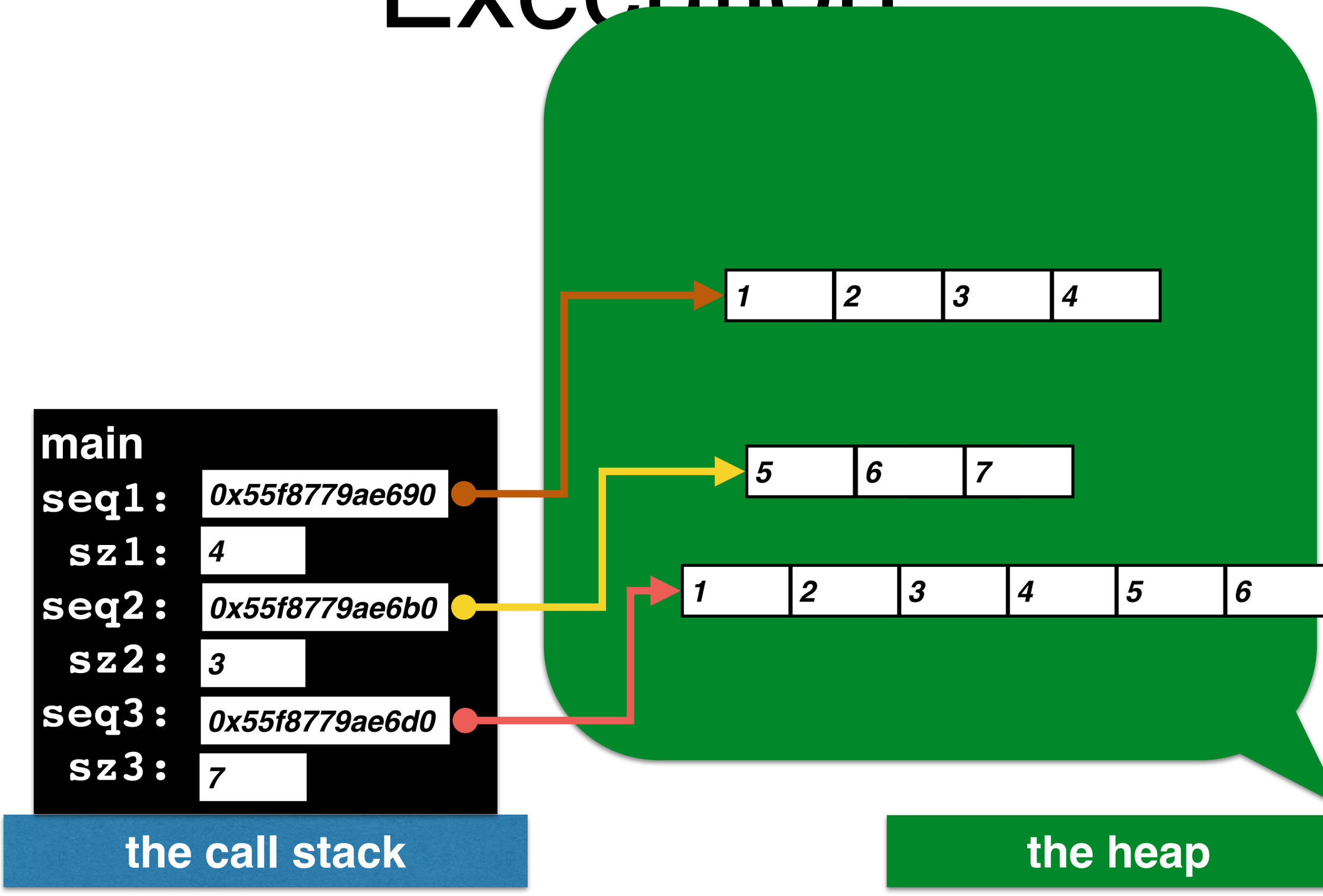
Picture of Append's Execution



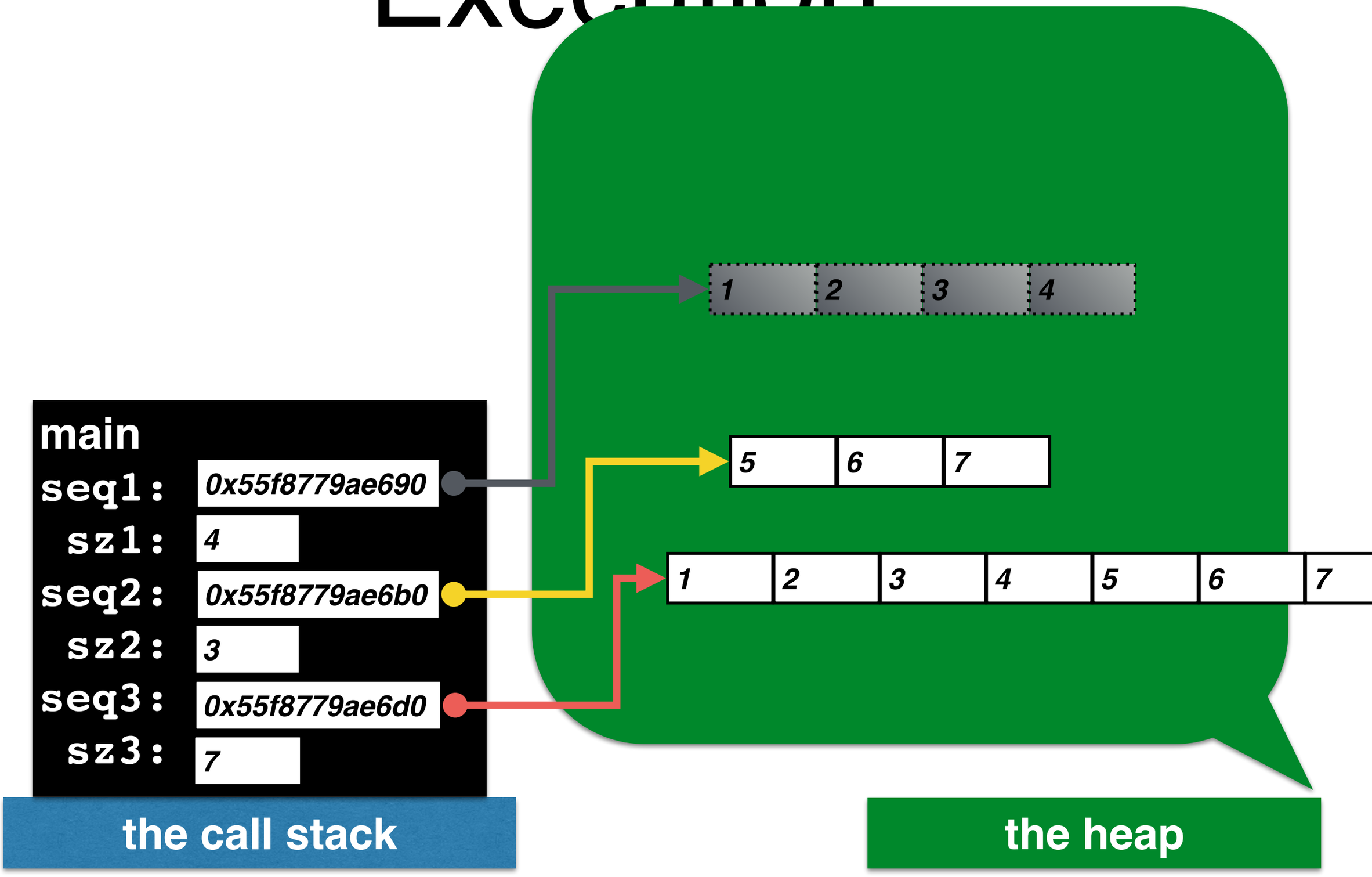
Picture of Append's Execution



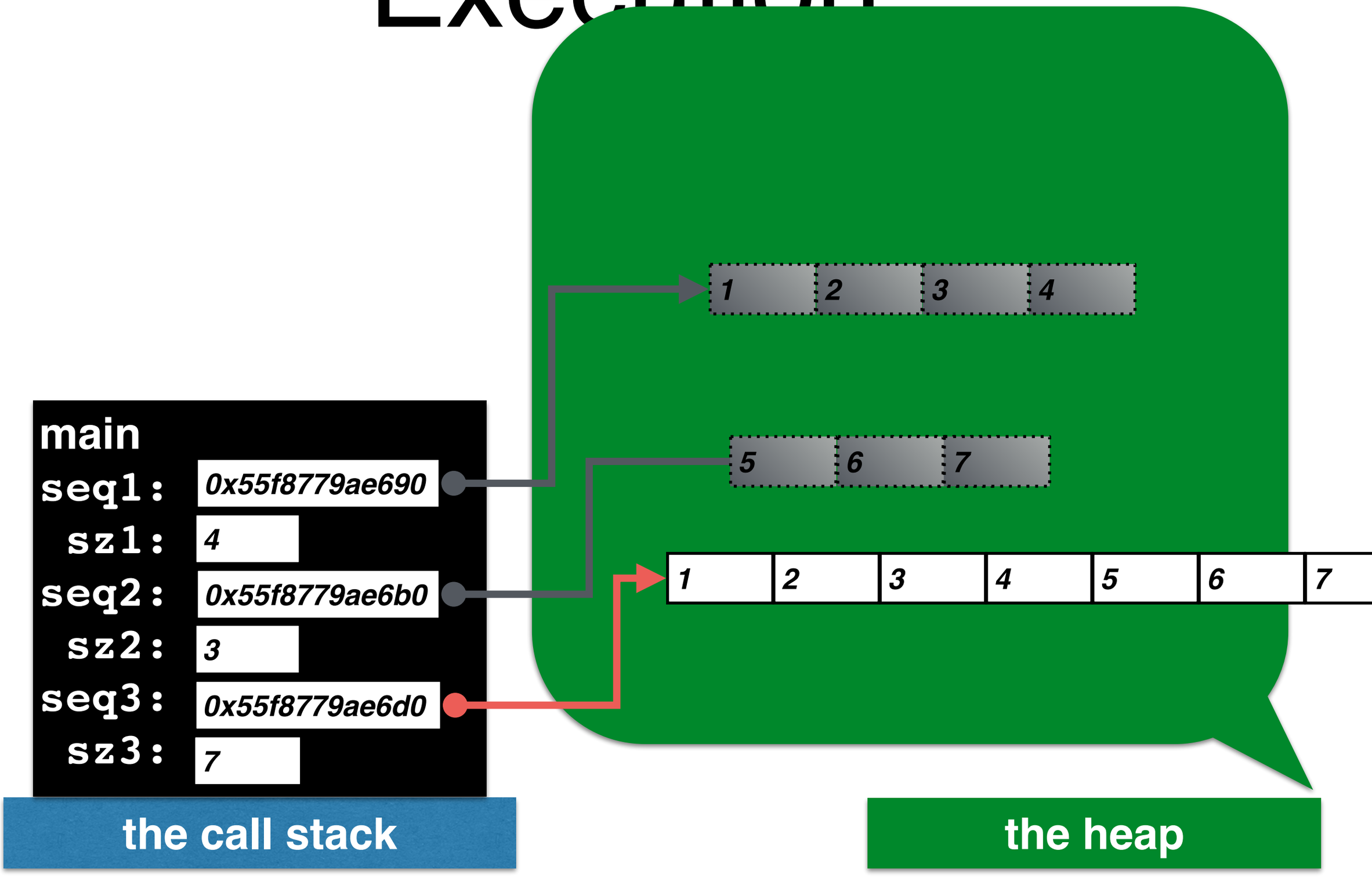
Picture of Append's Execution



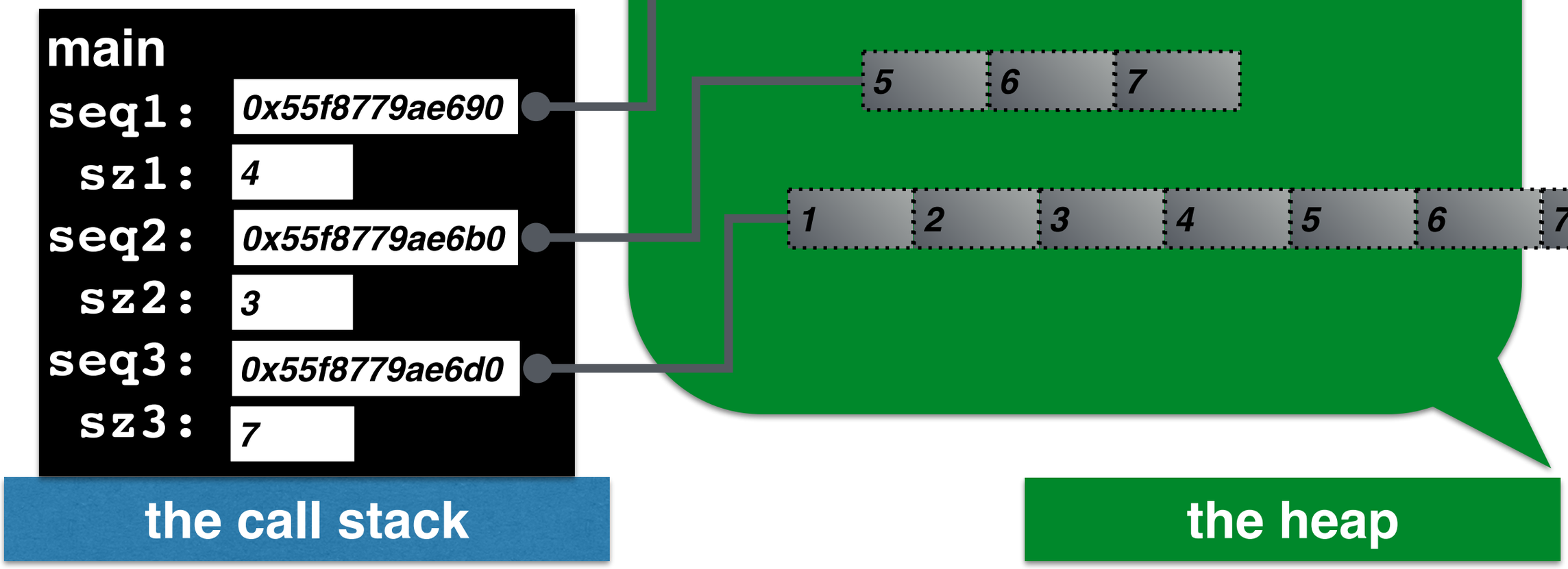
Picture of Append's Execution



Picture of Append's Execution



Picture of Append's Execution



Revised Array Syntax

➡ Declaration of an array variable.

statement ::= ***type**** ***var-name;***

➡ Type signature of a function/procedure:

func-proc-defn ::= ***type-or-void name*(...*, type var-name*, ...) {*block*}**

type ::= **`int | char | bool | double | std::string |`** ***type****

➡ Request for allocation of an array's storage within the heap:

statement ::= ***var-name*** **`=`** **`new`** ***type*`[`*int-expression*`];`**

➡ Accessing or modifying an array item:

store-loc ::= ***store-loc*** **`[`** ***int-expression*** **`]`**

expression ::= ***store-loc***

statement ::= ***store-loc*** **`=`** ***expression;***

➡ Release the allocation of an array's heap storage:

statement ::= **`delete []`** ***var-name;***

Next week

- ➡ Heap-allocated structs.
- ➡ Pointers and the "de-reference operator" *.
- ➡ Linked data structures.