POINTERS

LECTURE 04-1

JIM FIX, REED COLLEGE CS2-F20

RECALL

We've examined how the *call stack* operates.

We've allocated *arrays* and *structs* on the call stack.

We've passed arrays as *pointers*.

We've inspected pointers to stack variables using `&` notation.

We've obtained pointers to array data sitting on the heap with `new`.

We've released that array data by calling `delete []` on that pointer.

Let's now see how these constructs are generalized in C++...

Example: three.cc

```
#include <iostream>
int main(void) {
  int* A = new int[3];
  A[0]=10; A[1]=35; A[2]=17;

  int* front  = &(A[0]);
  int* middle = &(A[1]);
  int* end    = &(A[2]);

  std::cout<<A[0]<<" "<<A[1]<<" "<<A[2]<<"\n";
  front[1]  = 36;
  std::cout<<A[0]<<" "<<A[1]<<" "<<A[2]<<"\n";
  middle[0] = 37;
  std::cout<<A[0]<<" "<<A[1]<<" "<<A[2]<<"\n";
  end[-1]   = 38;
  std::cout<<A[0]<<" "<<A[1]<<" "<<A[2]<<"\n";

  delete [] A;
  // delete [] front; // would be ok, too.
}
```

Example: copyInto.cc

```
#include <iostream>

void outputArray(std::string lbl, int *A, int n) { ... }

void copyInto(int* src, int* dst, int num) {
  for (int i=0; i<num; i++) {
    dst[i] = src[i];
  }
}

int main(void) {
  int* A = new int[3];
  A[0]=10; A[1]=35; A[2]=17;
  int* B = new int[6];
  B[0]=16; B[1]=25; B[2]=36; B[3]=49; B[4]=64; B[5]=81;
  outputArray("A: ", A, 3);
  outputArray("B: ", B, 6);
  copyInto(A,&(B[2]),3);
  outputArray("A: ", A, 3);
  outputArray("B: ", B, 6);
  delete [] A; delete [] B;
}
```

Example: swap.cc

```
#include <iostream>

void swap(int* x, int* y) {
  int tmp = x[0];
  x[0] = y[0];
  y[0] = tmp;
}

int main(void) {
  int i = 42;
  int j = 37;
  std::cout << i << " " << j << std::endl;
  swap(&i,&j);
  std::cout << i << " " << j << std::endl;
}
```

A PROGRAM'S MEMORY

When your C++ program is run by the operating system, it runs as a ***process***.

- The system grants each process access to its own "fresh" array of ***memory;*** its own ***address space***
  - That memory area is essentially a huge array of bytes.
- Each byte holds a value that is 8 bits long.
  - The bit sequence 01011001, for example, represents the value 89. (Using base 2 notation, binary, versus base 10 notation, decimal)
- Your program stores variables, arrays, and structs in this memory as bytes.

A PROGRAM'S MEMORY (CONT'D)

Each memory byte has a location in memory. Each byte sits at an address.

- At a low level, your program executable requests bytes of data using their addresses.

Addresses are just numbers. Like indexes into an array.

- They run from 0 up to the size of the process address space (minus one).

Most system's C++ addresses are represented as 8 bytes, i.e. 64 bits long.

- Today's computer systems appear to use only 47 of those bits.
  - ➡ So 2^47 addressable memory locations. That's 128 terabytes.

VARIABLES IN MEMORY

The C++ compiler organizes your program so that each variable has its value stored in a sequence of bytes starting at some particular location in memory.

- Each program variable sits at some address in memory.
- You can use the address-of operator (**&*var-name***) to see that address.

```
double x = 42.0;
std::cout << "The storage for x is @" << (&x) << "\n";
```

- An int takes up 4 bytes, a double takes up 8 bytes, a char takes up one byte.
- Use sizeof(type), sizeof(var-name), or sizeof(expn) to get this number.

```
std::cout << "Ints use " << sizeof(int)  << " bytes.\n";
std::cout << "Doubles use " << sizeof(x) << " bytes.\n";
std::cout << "Chars use " << sizeof('a') << " bytes.\n";
```

VARIABLES IN MEMORY

Watching your program run, and when looking at the system level:

- When you access a variable's value, your program fetches the values of its bytes from the computer memory to calculate with them.

```
std::cout << i * 10 << std::endl;
```

- When you modify a variable's value, your program tells the memory system to update those bytes in its storage starting at that address.

```
i = i * 10;
```

VARIABLES IN MEMORY

Variables local to a function (including its parameters) are organized in a *frame*.

- Every running function has an active frame that resides somewhere in memory.
- Those active frames are "stacked up:"
    - Your code manages a *call stack*, made up of these active frames.

Suppose function `f` calls function `g`...

- The variables of `g` become "live," so they get space in a new frame for `g`
- The *callee* `g` gets an area in memory for its new frame.
    - Its stack frame sits just next to the stack frame of its *caller* `f`.
- When `g` returns, its stack frame's memory will be reused for other frames later.

INSPECTING STACK FRAMES

It's fun to inspect stack frames by using **&**, like so:

```
void g(int x) {
  int y=42;
  std::cout << "g: " << &x << " " << &y << "\n";
}
void f(int a) {
  int b=10;
  std::cout << "f: " << &a << " " << &b << "\n";
  g(37);
}
int main(void) {
  int i = 357;
  int j = 1000;
  std::cout << "main: " << &i << " " << &j << "\n";
  f(67);
  g(89);
}
```

STACK-ALLOCATED DATA

- We first placed arrays and structs as local data within a function.
- These are stack-allocated arrays and structs.

```
int a[10];
cmpx z;
```

- We use **&** to find the addresses of array and struct components:

```
std::cout << "a[2] lives at " << &(a[2]) << std::endl;
std::cout << "a[3] lives at " << &(a[3]) << std::endl;
std::cout << "z.re lives at " << &(z.re) << std::endl;
std::cout << "z.im lives at " << &(z.im) << std::endl;
```

- These array and struct components are laid out in their stack frame's memory.
- Their lifetime is the same as the lifetime of their function.

THE STACK, THE BINARY SEGMENT, GLOBALS, AND THE HEAP

There are four major areas of memory:

- The *call stack* lives at the highest addresses; it grows to use lower addresses.
- The program's code or "*binary*" lives at the lowest addresses.
- The program's *global* data and constants sit just above there.
- The *heap* starts above the global area and grows upward.

HEAP-ALLOCATED ARRAYS

We just learned how to allocate arrays on the heap:

- We use `new` to get a chunk of memory from the heap. Syntax:

  element-type `*` ***variable-name*** `= new` ***element-type*** `[`***size***`];`

- We are given ***size*** `*sizeof(`***element-type***`)` bytes from the heap.
- The value of is a *pointer value*, i.e. the address of the start of those bytes.

When you access an array item with ***variable-name*** `[`***index***`]` your program:

- It uses the pointer value as a *base address*
- It multiplies ***index*** by `sizeof(`***element-type***`)`, adds that to the base.
- This is an *offset* from the base. It fetches the data at that calculated address.

INSPECTING ARRAY DATA LOCATIONS

```
int main(void) {

  int* a = new int[10];
  int* b = new int[100];
  int* c = new int[10];

  std::cout << "a[0] is at " << &(a[0]) << std::endl;
  std::cout << "b[0] is at " << &(b[0]) << std::endl;
  std::cout << "c[0] is at " << &(c[0]) << std::endl;

  std::cout << "a starts at " << a << std::endl;
  std::cout << "b starts at " << b << std::endl;
  std::cout << "c starts at " << c << std::endl;

  std::cout << "a[0] is at " << &(a[0]) << std::endl;
  std::cout << "a[1] is at " << &(a[1]) << std::endl;
  std::cout << "a[2] is at " << &(a[2]) << std::endl;
  std::cout << "a[3] is at " << &(a[3]) << std::endl;
}
```

HEAP-ALLOCATED ARRAYS (CONT'D)

- When allocated on the heap, an array's lifetime is decoupled from its variables frame:
  - Can pass the pointer to an array's storage to other functions
  - Can `return` the pointer to an array's storage to the calling function.
- To "de-allocate" the array's heap storage, use the `delete` keyword:

  `delete []` *variable-name*;

- The heap can then re-use this storage for other allocation requests.

POINTERS

- The keyword **new** gives us back a pointer value:

```
int* a = new int[4];
```

- It gives us back a "pointer to an array of four integers"
  - ➡16 bytes that live within the heap.

The address-of operator also gives us pointers! Consider the code below

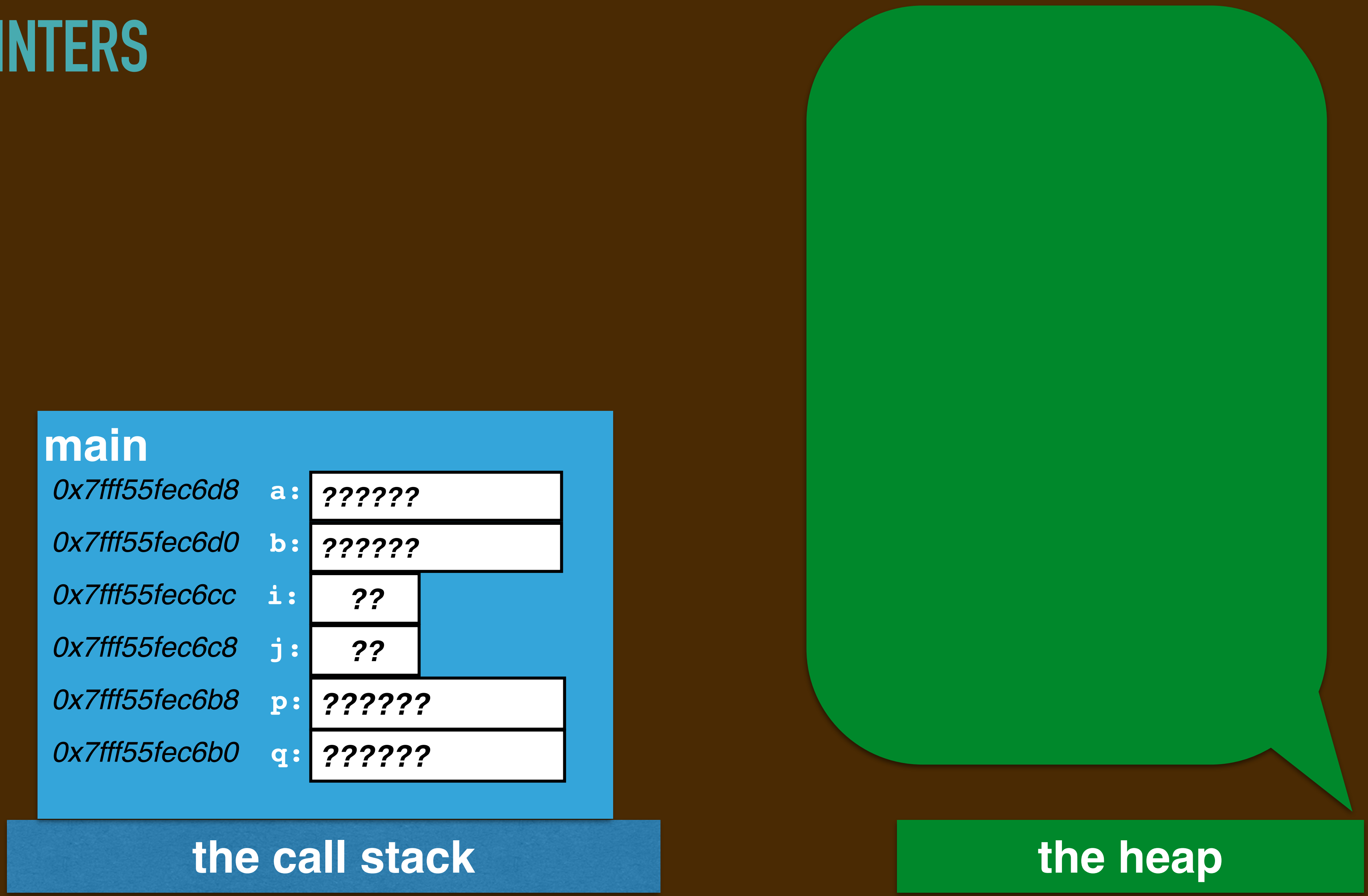
```
int main(void) {
  int i = 42;
  int j = 37;
  int* p = &i;
  int* q = &j;
  std::cout << "i lives at" << p << std::endl;
  std::cout << "j lives at" << q << std::endl;
}
```

POINTERS

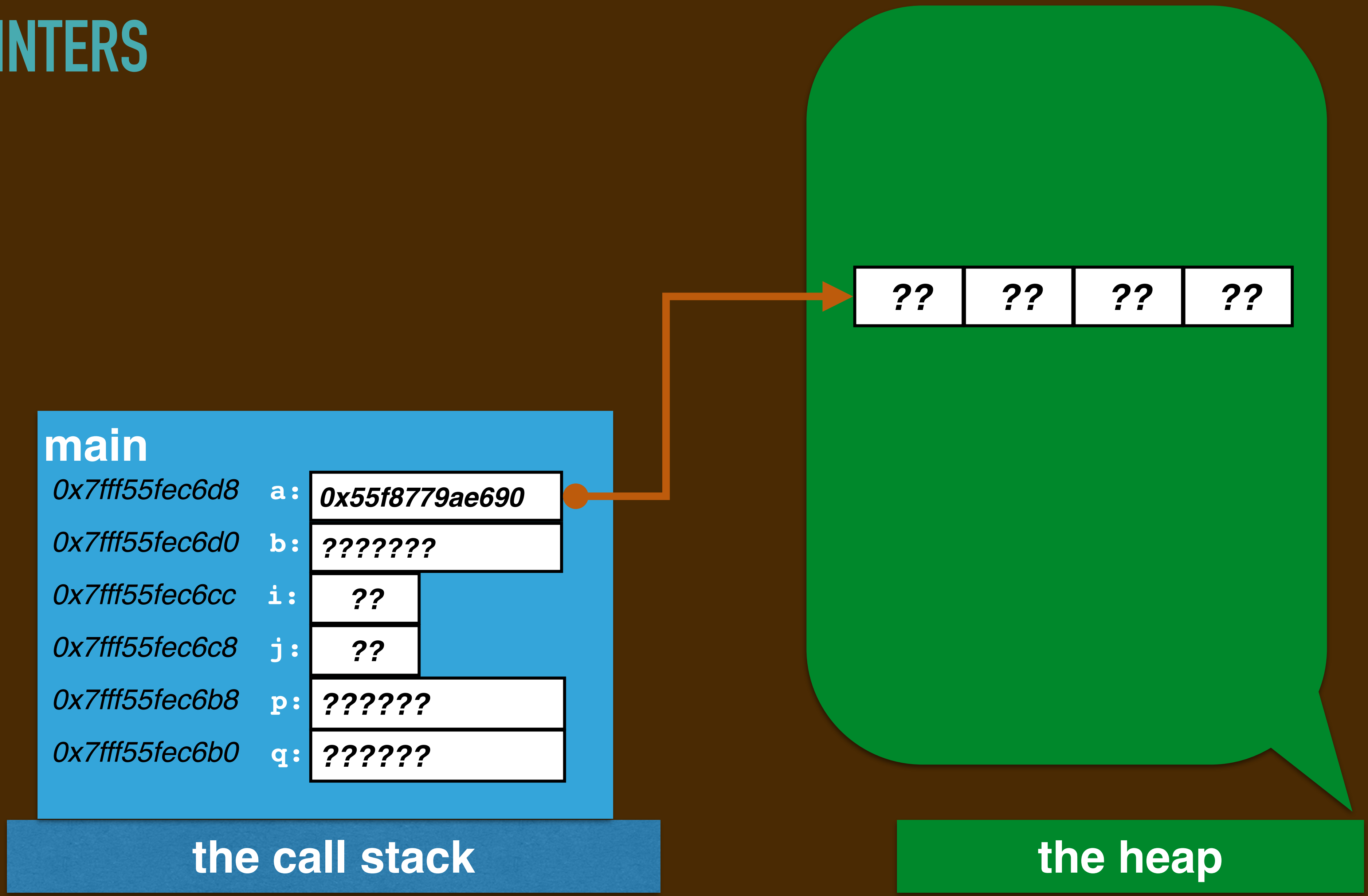
The address-of operator also gives us pointers! Consider the code below

```
int main(void) {
  int a = new int[4];
  int b = new int[3];
  int i = 42;
  int j = 37;
  int* p = &i;
  int* q = &j;
  std::cout << "i lives at" << p << std::endl;
  std::cout << "j lives at" << q << std::endl;
}
```

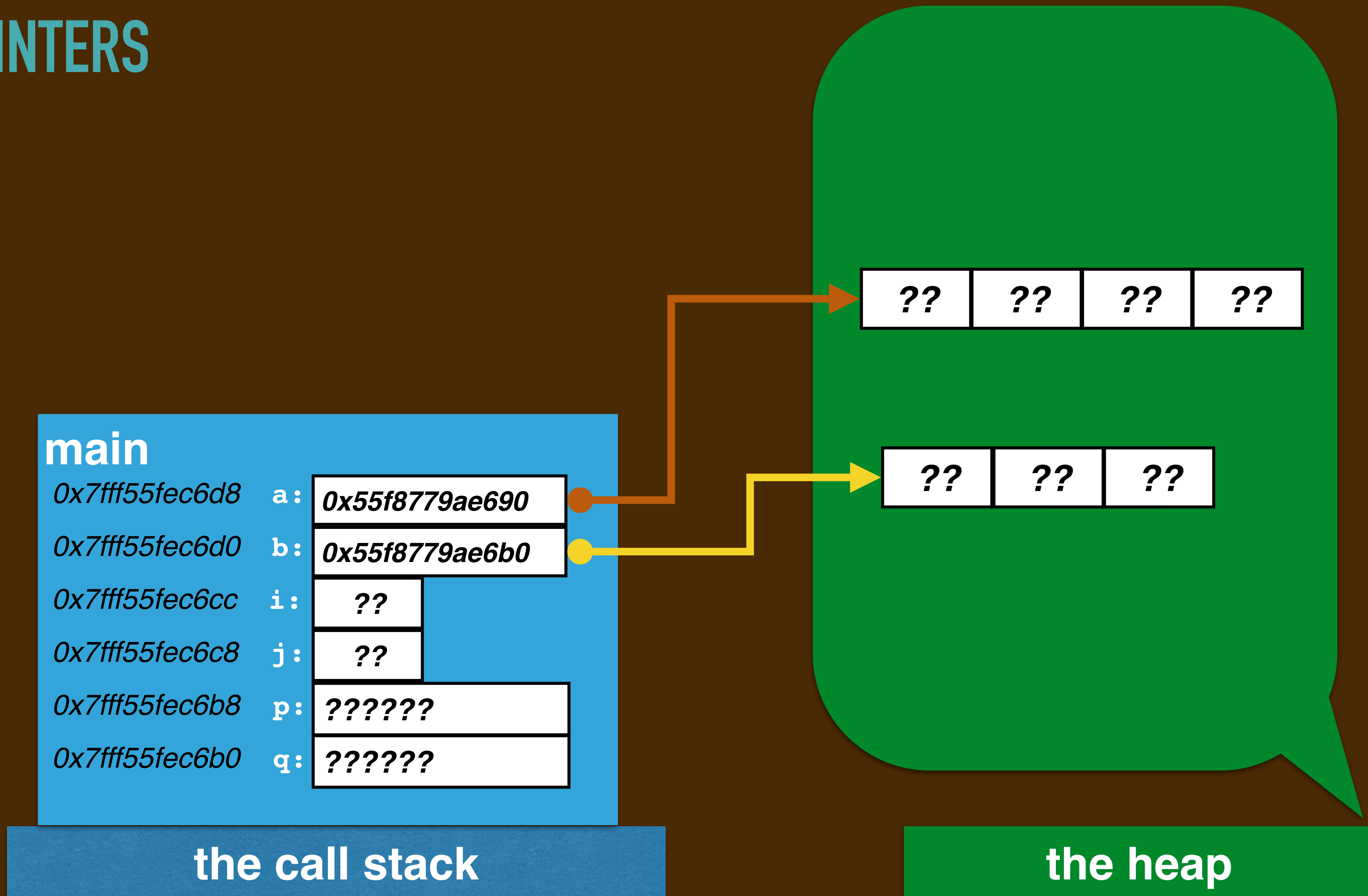
POINTERS


main
0x7fff55fec6d8 a: ??????
0x7fff55fec6d0 b: ??????
0x7fff55fec6cc i: ??
0x7fff55fec6c8 j: ??
0x7fff55fec6b8 p: ??????
0x7fff55fec6b0 q: ??????
the call stack
the heap

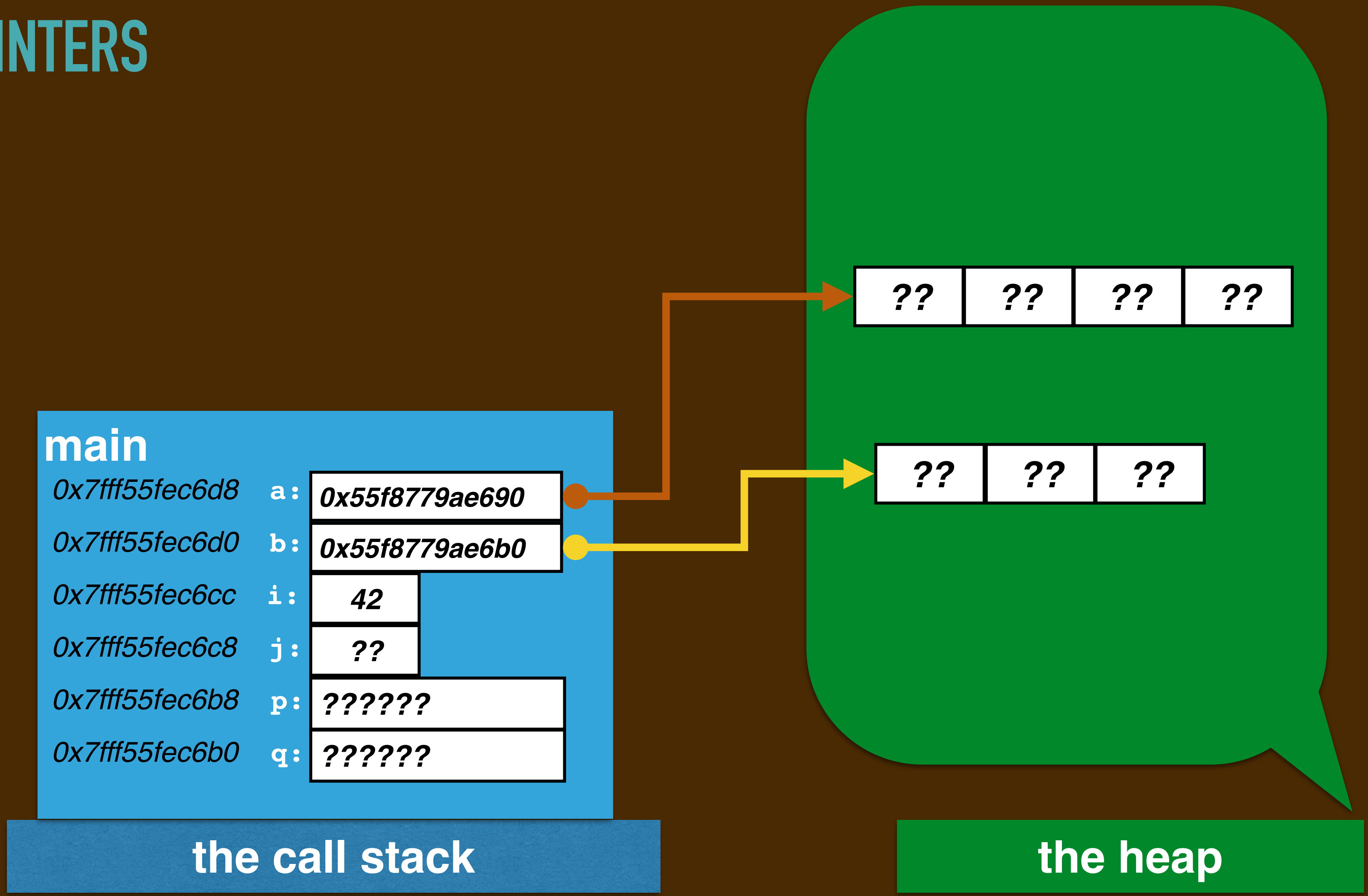
POINTERS


main
0x7fff55fec6d8
a:
0x55f8779ae690
0x7fff55fec6d0
b:
???????
0x7fff55fec6cc
i:
??
0x7fff55fec6c8
j:
??
0x7fff55fec6b8
p:
??????
0x7fff55fec6b0
q:
??????
??
??
??
??
the call stack
the heap

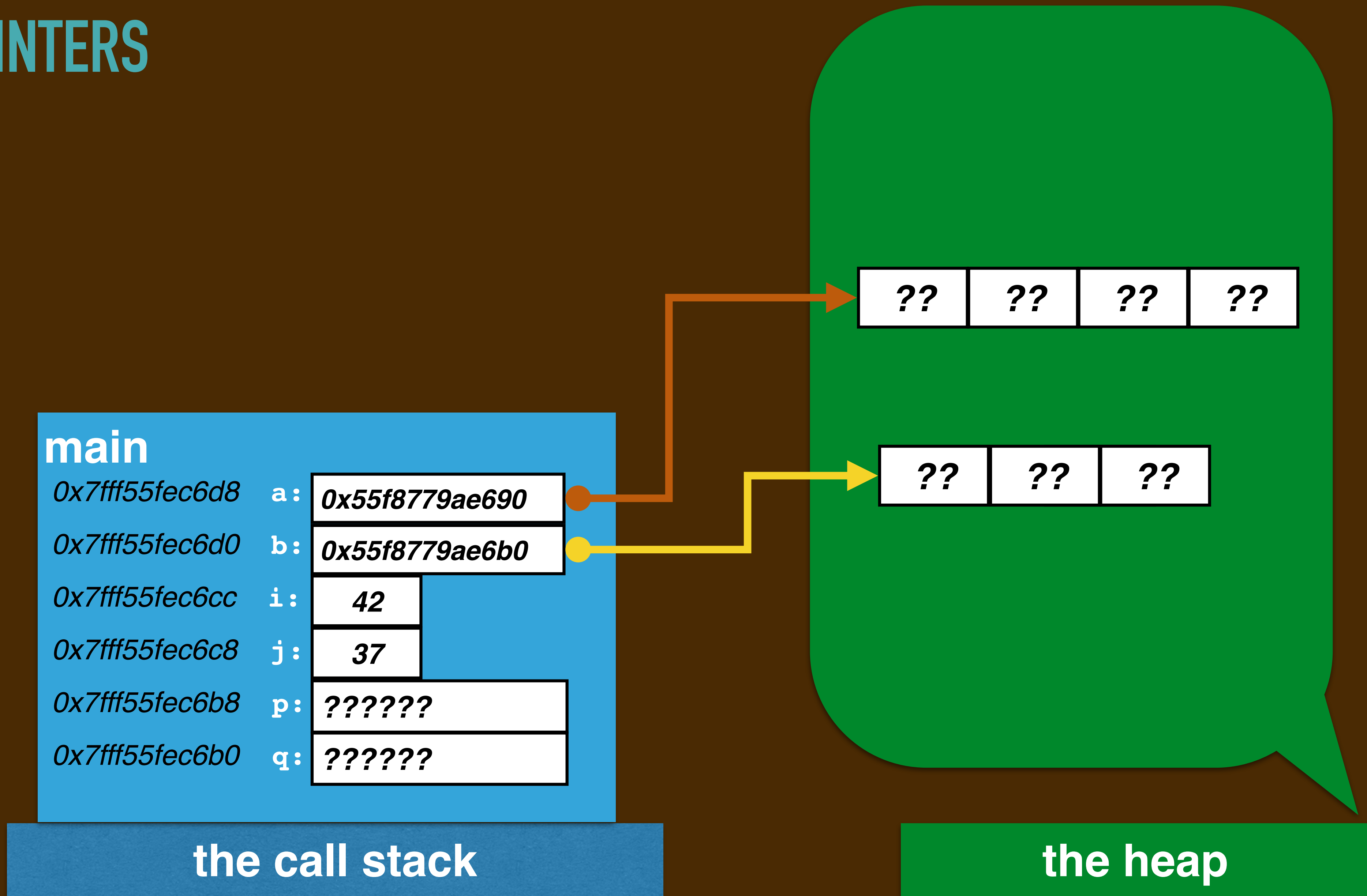
POINTERS


main
0x7fff55fec6d8
a:
0x55f8779ae690
0x7fff55fec6d0
b:
0x55f8779ae6b0
0x7fff55fec6cc
i:
??
0x7fff55fec6c8
j:
??
0x7fff55fec6b8
p:
??????
0x7fff55fec6b0
q:
??????
??
??
??
??
??
??
??
the call stack
the heap

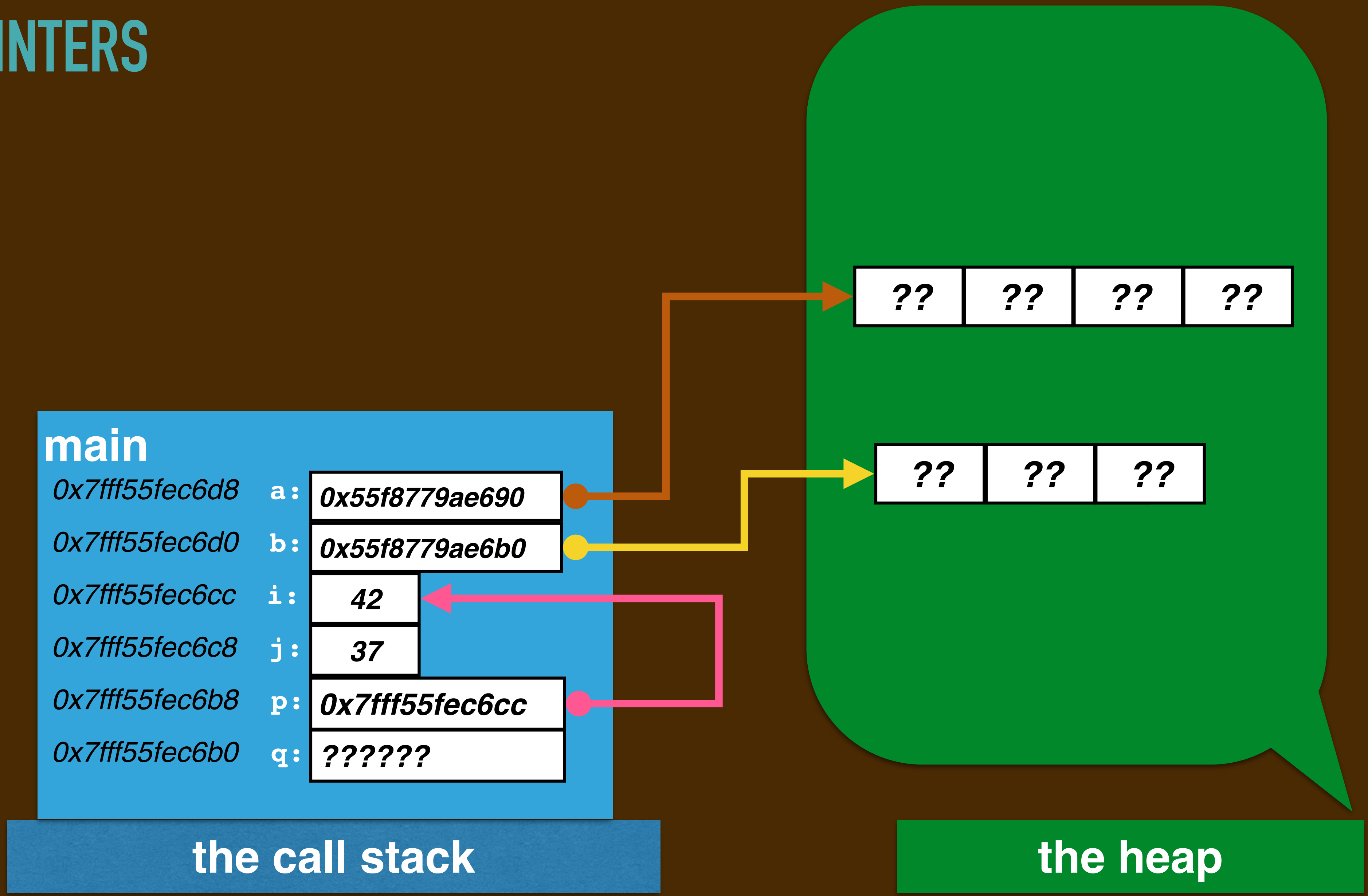
POINTERS


main
0x7fff55fec6d8
a:
0x55f8779ae690
0x7fff55fec6d0
b:
0x55f8779ae6b0
0x7fff55fec6cc
i:
42
0x7fff55fec6c8
j:
??
0x7fff55fec6b8
p:
??????
0x7fff55fec6b0
q:
??????
?? ?? ?? ??
?? ?? ??
the call stack
the heap

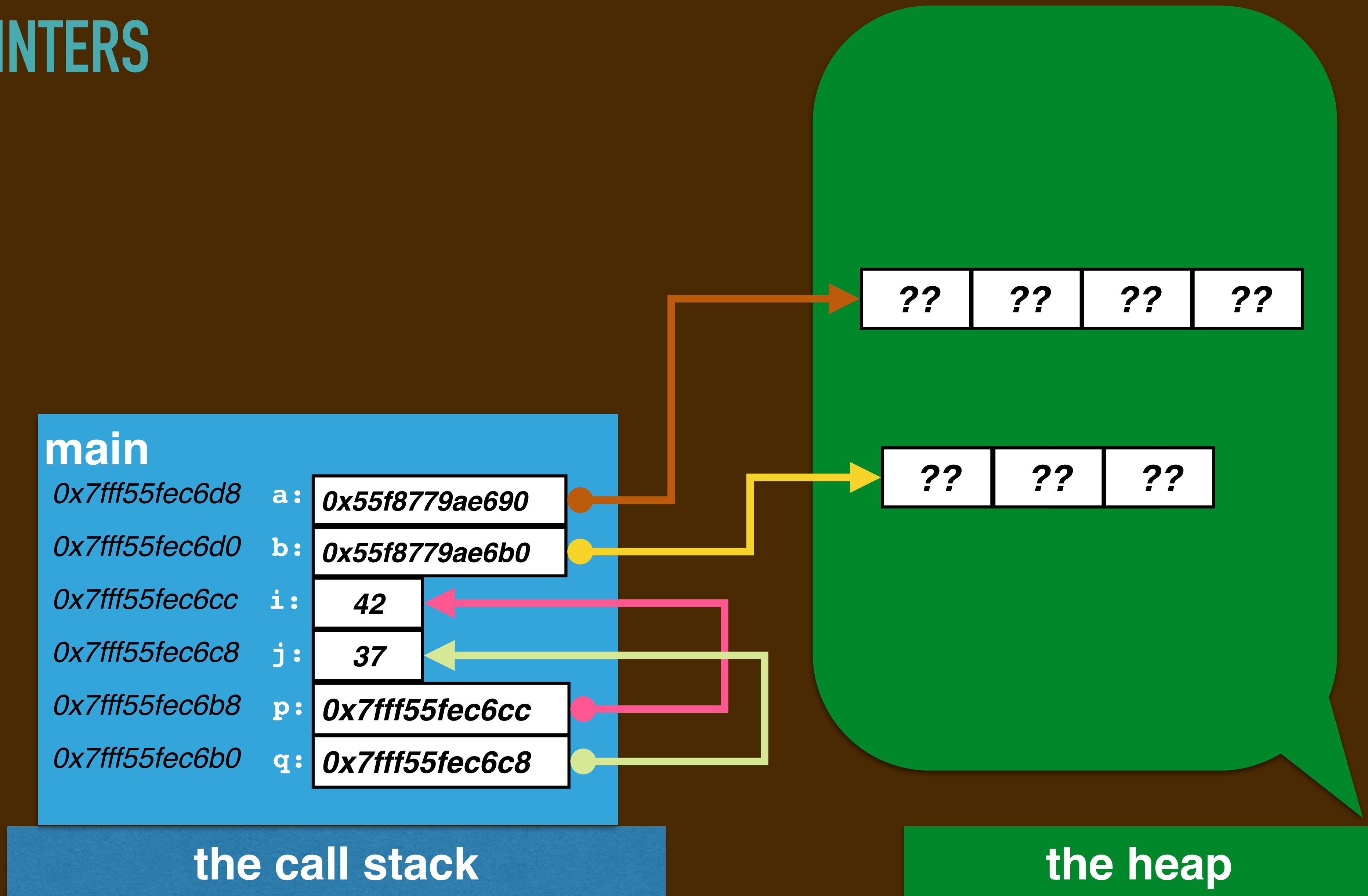
POINTERS


main
0x7fff55fec6d8
a:
0x55f8779ae690
0x7fff55fec6d0
b:
0x55f8779ae6b0
0x7fff55fec6cc
i:
42
0x7fff55fec6c8
j:
37
0x7fff55fec6b8
p:
??????
0x7fff55fec6b0
q:
??????
?? ?? ?? ??
?? ?? ??
the call stack
the heap

POINTERS


main
0x7fff55fec6d8
a:
0x55f8779ae690
0x7fff55fec6d0
b:
0x55f8779ae6b0
0x7fff55fec6cc
i:
42
0x7fff55fec6c8
j:
37
0x7fff55fec6b8
p:
0x7fff55fec6cc
0x7fff55fec6b0
q:
??????
?? ?? ?? ??
?? ?? ??
the call stack
the heap

POINTERS



main
0x7fff55fec6d8
a:
0x55f8779ae690
0x7fff55fec6d0
b:
0x55f8779ae6b0
0x7fff55fec6cc
i:
42
0x7fff55fec6c8
j:
37
0x7fff55fec6b8
p:
0x7fff55fec6cc
0x7fff55fec6b0
q:
0x7fff55fec6c8
?? ?? ?? ??
?? ?? ??
the call stack
the heap

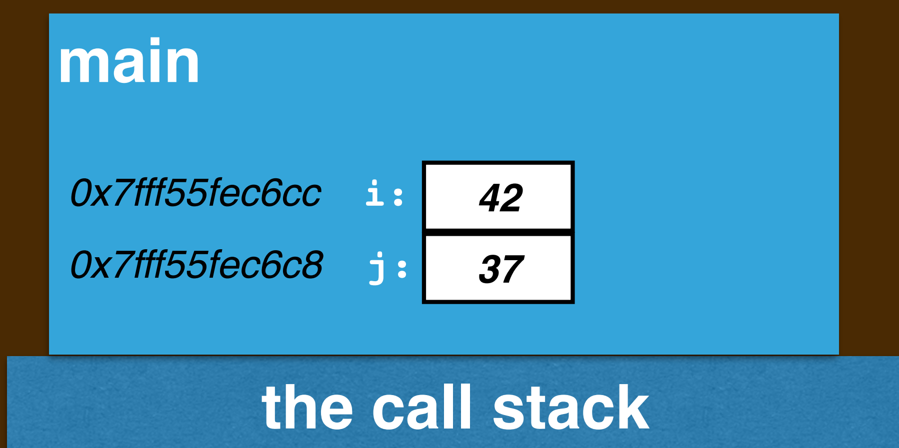
POINTERS AS ARRAYS!

We can treat **p** and **q** as arrays:

```
int main(void) {
  int i = 42;
  int j = 37;
  int* p = &i;
  int* q = &j;
  std::cout << "i lives at" << p << std::endl;
  std::cout << p[0] << "is stored there and ";
  std::cout << p[1] << "is just above" << std::endl;
  std::cout << "j lives at" << q << std::endl;
  std::cout << q[0] << "is stored there and ";
  std::cout << q[1] << "is just above" << std::endl;
}
```

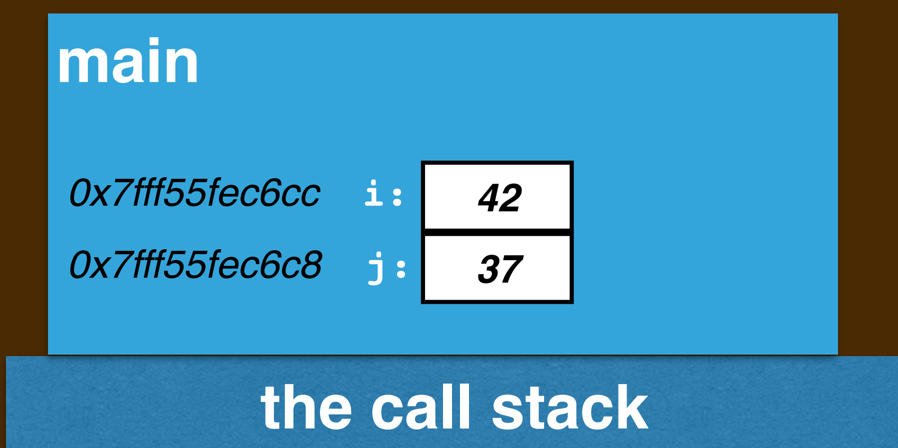
SWAP-AT ILLUSTRATED

```
void swapAt(int* a, int* b) {
  int temporary = a[0];
  a[0] = b[0];
  b[0] = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```



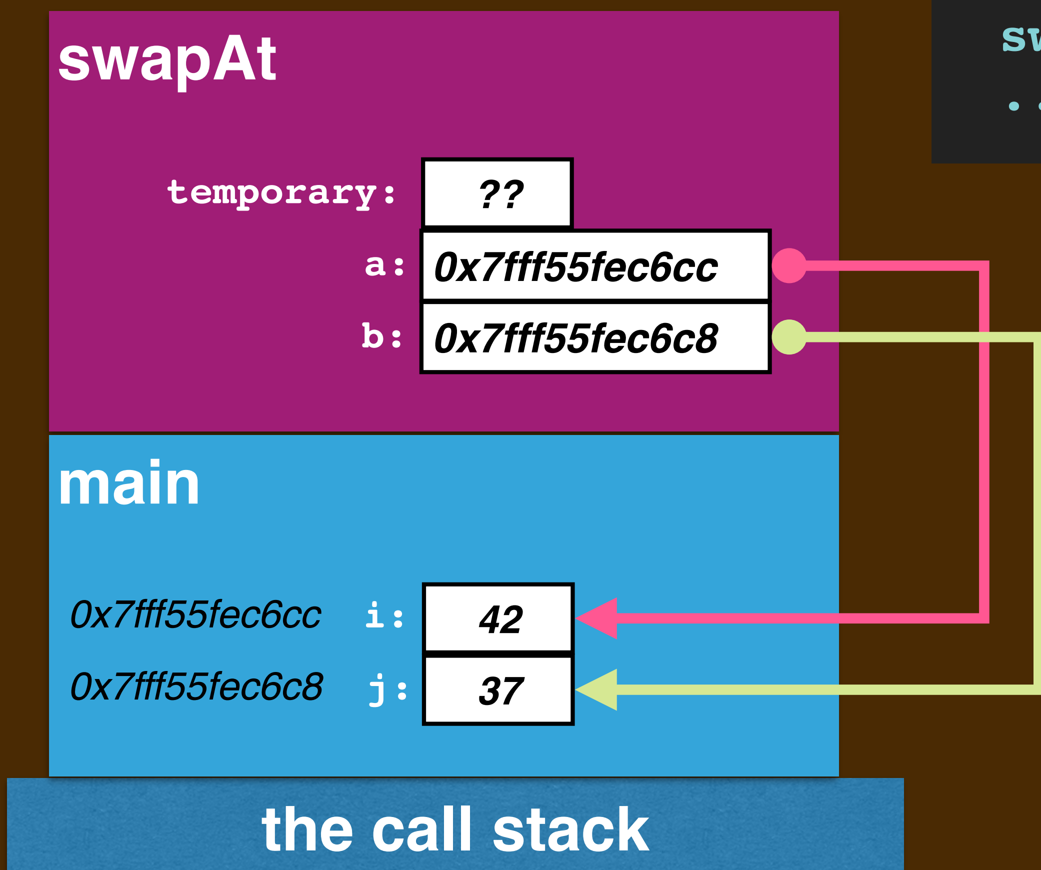
SWAP-AT ILLUSTRATED

```
void swapAt(int* a, int* b) {
  int temporary = a[0];
  a[0] = b[0];
  b[0] = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```



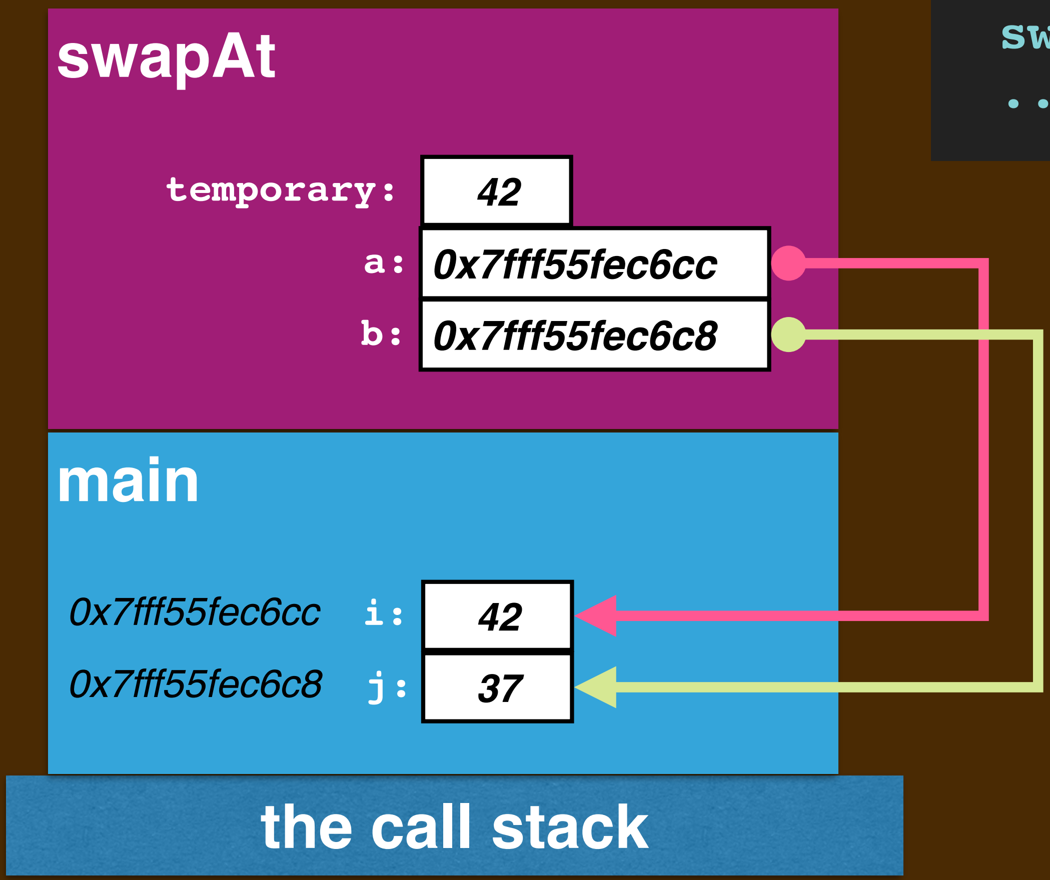
SWAP-AT ILLUSTRATED

```
void swapAt(int* a, int* b) {
   int temporary = a[0];
   a[0] = b[0];
   b[0] = temporary;
}
   ...
   swapAt(&i,&j);
   ...
```



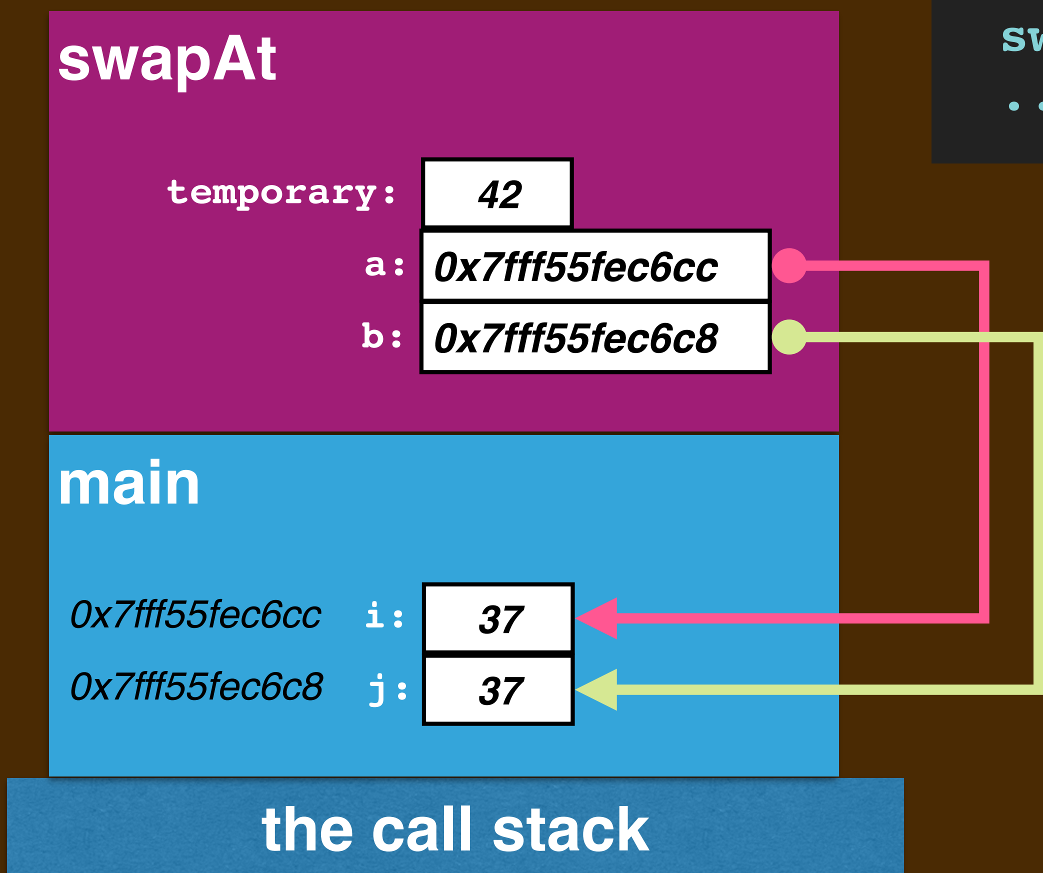
SWAP-AT ILLUSTRATED

```
void swapAt(int* a, int* b) {
  int temporary = a[0];
  a[0] = b[0];
  b[0] = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```



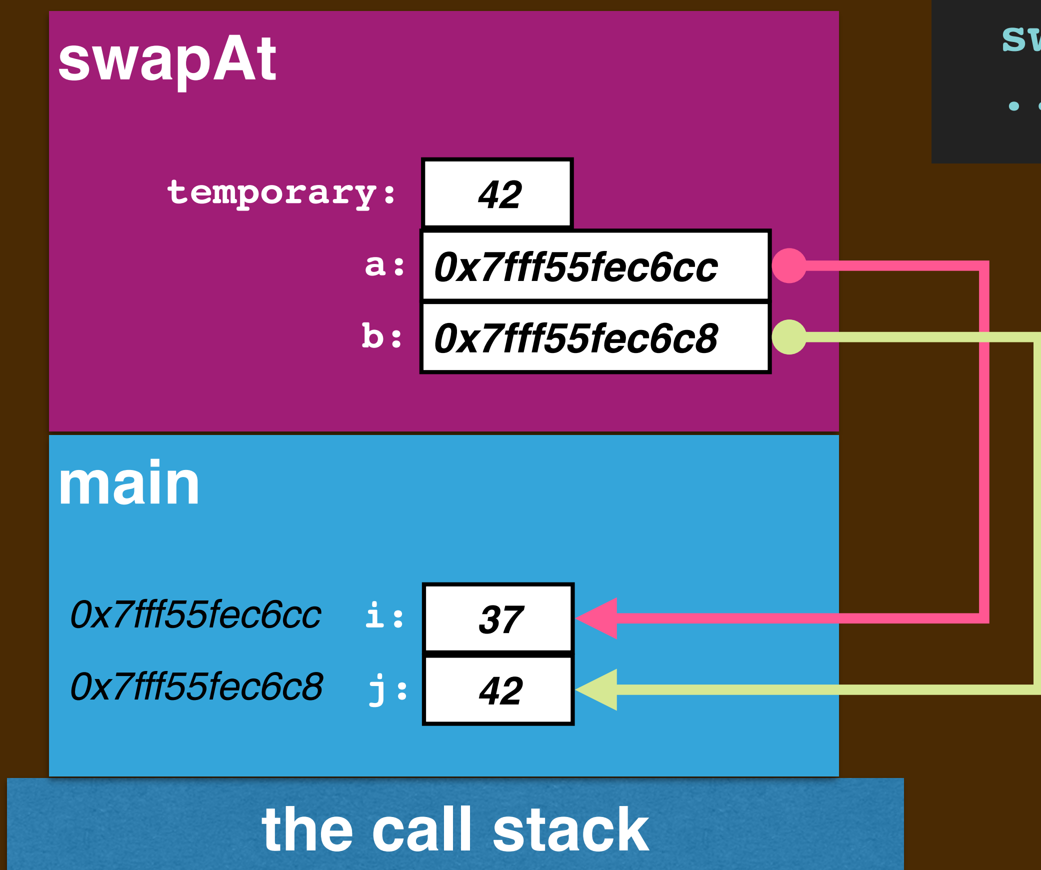
SWAP-AT ILLUSTRATED

```
void swapAt(int* a, int* b) {
  int temporary = a[0];
  a[0] = b[0];
  b[0] = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```



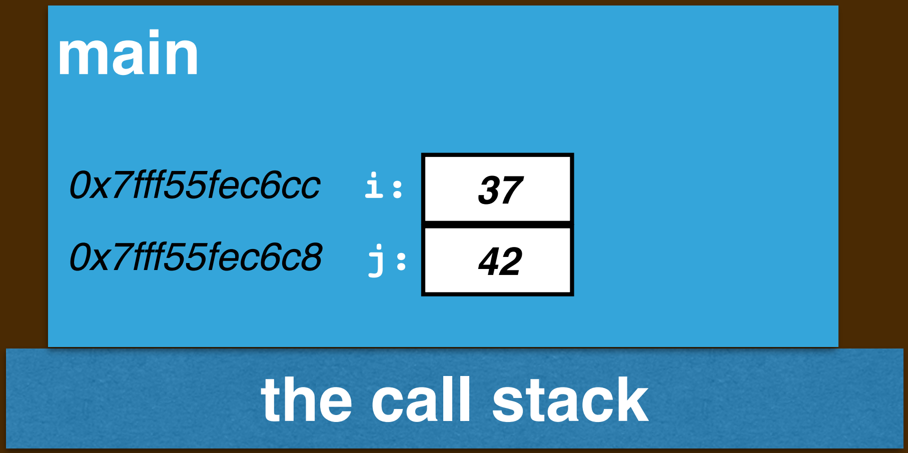
SWAP-AT ILLUSTRATED

```
void swapAt(int* a, int* b) {
  int temporary = a[0];
  a[0] = b[0];
  b[0] = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```



SWAP-AT ILLUSTRATED

```
void swapAt(int* a, int* b) {
  int temporary = a[0];
  a[0] = b[0];
  b[0] = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```



ALTERNATE ARRAY ACCESS NOTATION: DEREFERENCE *

The array index notation `array[index]` is actually shorthand for the "dereference at" notation:

`*(array+index)`

This means

> *"consider the pointer nudged* **`index`** *values further… access the memory there."*

- The nudge depends on the array element's data type:
  - ➡ `4*index` for **`int`**, `1*index` for **`char`**, `8*index` for **`double`**, etc.
- The calculation in parenthesis is called "pointer arithmetic."
- The `*` means "access the value at" and is called *"dereferencing the pointer."*

DEREFERENCE OPERATOR

This means that `array[0]` can instead be written `*(array)`.

DEREFERENCE OPERATOR

This means that `array[0]` can instead be written `*array`.

DEREFERENCE OPERATOR

This means that `array[0]` can instead be written `(*array)`.

```
void swapAt(int* a, int* b) {
  int temporary = a[0];
  a[0] = b[0];
  b[0] = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```

DEREFERENCE OPERATOR

This means that `array[0]` can instead be written `(*array)`.

```
void swapAt(int* a, int* b) {
  int temporary = (*a);
  (*a) = (*b);
  (*b) = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```

DEREFERENCE OPERATOR

This means that `array[0]` can instead be written `(*array)`.

Example. The code for `swapAt` is normally written like so:

```
void swapAt(int* a, int* b) {
  int temporary = (*a);
  (*a) = (*b);
  (*b) = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```

DEREFERENCE OPERATOR

This means that `array[0]` can instead be written `(*array)`.

Example. The code for `swapAt` is normally written like so:

```
void swapAt(int* a, int* b) {
  int temporary = (*a);
  (*a) = (*b);
  (*b) = temporary;
}
  ...
  swapAt(&i,&j);
  ...
```

Do not confuse the `&` and `*` operators!!!!! (They are *inverses*, actually.)

- The `&` eans "get the address of" and the `*` means "access the value at."

POINTER PARAMETERS REVISITED

```
void swapAt(int* a, int* b) {
  int temporary = (*a);
  (*a) = (*b);
  (*b) = temporary;
}
void incrementAt(int *p) {
  (*p) = (*p) + 1;
}
int main(void) {
  int i = 42;
  int j = 37;
  std::cout << "i lives at" << &i << " with value" << i << "\n";
  std::cout << "j lives at" << &j << " with value" << j << "\n";
  swapAt(&i,&j);
  incrementAt(&i);
  std::cout << "i lives at" << &i << " with value" << i << "\n";
  std::cout << "j lives at" << &j << " with value" << j << "\n";
}
```

ALLOCATING "SINGLETONS" ON THE HEAP

We can also request single data locations, not just arrays, from the heap:

```
int main(void) {
  int *p = new int;
  (*p) = 42;
  int *q = new int;
  (*q) = 37;
  std::cout << "The value at "<< p << " is " << (*p) << ".\n";
  std::cout << "The value at "<< q << " is " << (*q) << ".\n";
  swapAt(p,q);
  incrementAt(p);
  std::cout << "The value at "<< p << " is " << (*p) << ".\n";
  std::cout << "The value at "<< q << " is " << (*q) << ".\n";
  delete p;
  delete q;
}
```

ALLOCATING "SINGLETONS" ON THE HEAP

We can also request single data locations, not just arrays, from the heap:

```
int main(void) {
  int *p = new int;
  (*p) = 42;
  int *q = new int;
  (*q) = 37;
  std::cout << "The value at "<< p << " is " << (*p) << ".\n";
  std::cout << "The value at "<< q << " is " << (*q) << ".\n";
  swapAt(p,q);
  incrementAt(p);
  std::cout << "The value at "<< p << " is " << (*p) << ".\n";
  std::cout << "The value at "<< q << " is " << (*q) << ".\n";
  delete p;
  delete q;
}
```

SINCE THESE ARE HEAP-ALLOCATED, MUST RELEASE THEIR STORAGE!

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0, vwbus);
  outputCar(*vwbus);
}
```

ALLOCATING STRUCTS ON THE HEAP

We can ***allocate structs*** within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0, vwbus);
  outputCar(*vwbus);
}
```

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0,vwbus);
  outputCar(*vwbus);
}
```

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0,vwbus);
  outputCar(*vwbus);
}
```

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0,vwbus);
  outputCar(*vwbus);
}
```

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0,vwbus);
  outputCar(*vwbus);
}
```

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0,vwbus);
  outputCar(*vwbus);
}
```

NOTICE HOW ALL THE TYPES MATCH UP!

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0,vwbus);
  outputCar(*vwbus);
  delete vwbus;
}
```

WHOOPS! DON'T FORGET TO GIVE RELEASE THE POINTER.

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of `car.cc` from **Lab 03**:

```
struct car { ... };

void outputCar(car c) { ... }

void drive (double distance, car* p) { ... }

int main(void) {
  car *vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  outputCar(*vwbus);
  drive(100.0,vwbus);
  outputCar(*vwbus);
  delete vwbus;
}
```

WHOOPS! DON'T FORGET TO GIVE RELEASE THE POINTER.

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of drive from **Lab 03**:

```
car drive(double d, car c) {
  double fuelNeeded = d / c.mpg;
  if (c.fuel > fuelNeeded) {
    c.fuel -= fuelNeeded;
    c.odometer += d;
  } else {
    double fraction = c.fuel / fuelNeeded;
    c.fuel = 0.0;
    c.odometer += fraction * d;
  }
  return c;
}
int main(void) {
  car vwbus {"VW", "Bus", 12300, 10.8, 19};
  ...
  vwbus = drive(100.0,vwbus)
  ...
```

ALLOCATING STRUCTS ON THE HEAP

We can allocate structs within the heap.

- **Example.** rewrite of drive from **Lab 03**:

```
void drive(double d, car* p) {
  double fuelNeeded = d / (*p).mpg;
  if ((*p).fuel > fuelNeeded) {
    (*p).fuel -= fuelNeeded;
    (*p).odometer += d;
  } else {
    double fraction = (*p).fuel / fuelNeeded;
    (*p).fuel = 0.0;
    (*p).odometer += fraction * d;
  }
  return;
}
int main(void) {
  car* vwbus = new car {"VW", "Bus", 12300, 10.8, 19};
  ...
  drive(100.0,vwbus)
  ...
```

ON WEDNESDAY

We'll look at ***linked data structures***.

Our goal is to eventually...

- ...build our own sequence data structures using "***linked lists***."
- ...build our own search data structures using "***binary trees***."
- ...build "resizeable" arrays and dictionaries E.g. a "***bucket hashtable***."