[368] Vectors and Movement

Tyler Caraza-Harter

Outline

Worksheet and TopHat

Vectors

Moving vs. Copying

What will you learn today?

Learning objectives

- make informed decisions about how to store values with vectors
- take advantage of move semantics to avoid unnecessary copies

Outline

Worksheet and TopHat

Vectors

Moving vs. Copying

C Arrays, C++ Arrays, Vectors

```
// C array
int size = 10;
auto nums1 = new int[size];
```

nums1 is a pointer, need to keep/pass size variable around with nums1

```
// C++ array
auto nums2 = new array<int, 10>();
cout << nums2->size() << "\n";
f(nums2);
```

```
// vector
auto nums3 = new vector<int>(10);
cout << nums3->size() << "\n";
nums3->push_back(368);
g(nums3);
```

C Arrays, C++ Arrays, Vectors

```
// C array
int size = 10;
auto nums1 = new int[size];
```

the size is part of the type! Must be a literal or a constexpr.

```
// C++ array
auto nums2 = new array<int, 10>();
cout << nums2->size() << "\n";
f(nums2);
```

don't need to track size separately

f must accept arrays of size 10 specifically!

```
// vector
auto nums3 = new vector<int>(10);
cout << nums3->size() << "\n";
nums3->push_back(368);
g(nums3);
```

C Arrays, C++ Arrays, Vectors

```
// C array
int size = 10;
auto nums1 = new int[size];


// C++ array
auto nums2 = new array<int, 10>();
cout << nums2->size() << "\n";
f(nums2);
```

int is part of the type, but size is just an argument and is decided at runtime

```
// vector
auto nums3 = new vector<int>(10);
cout << nums3->size() << "\n";
nums3->push_back(368);
g(nums3);
```

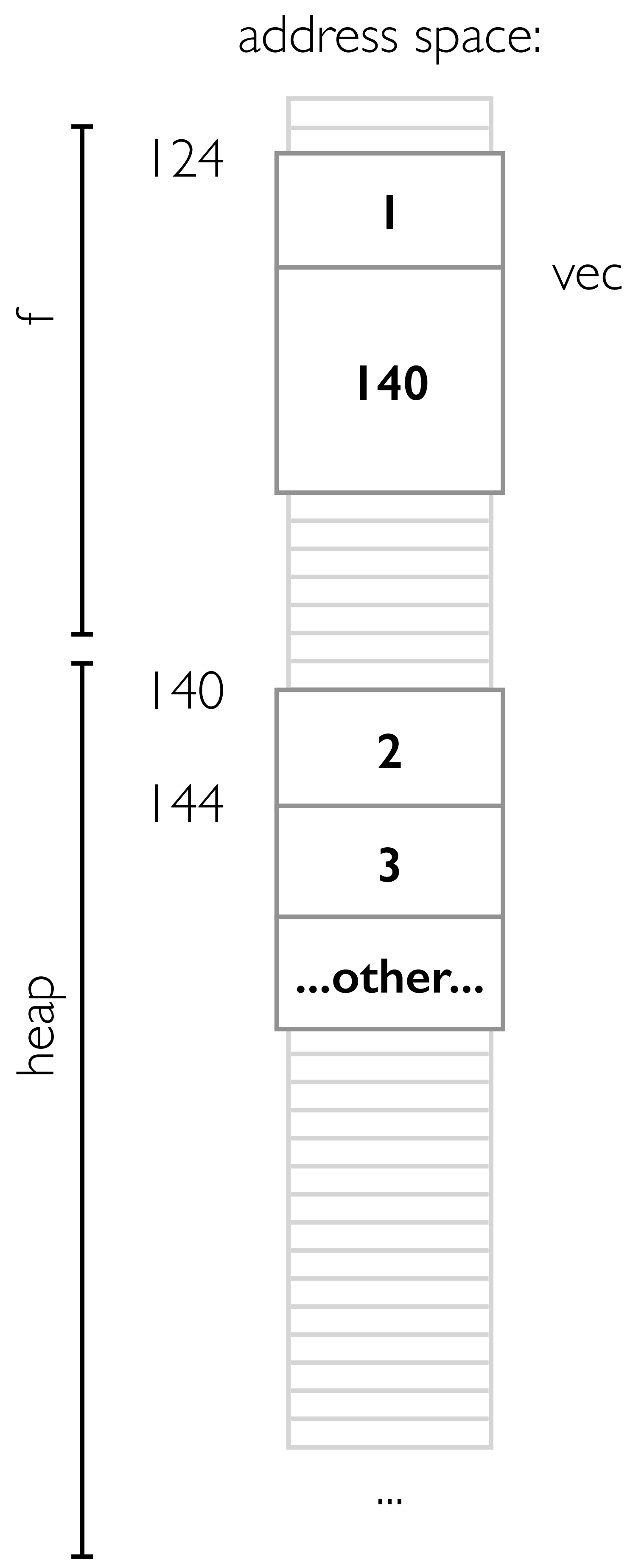
don't need to track size separately

size can change

g may take an int vector of any size

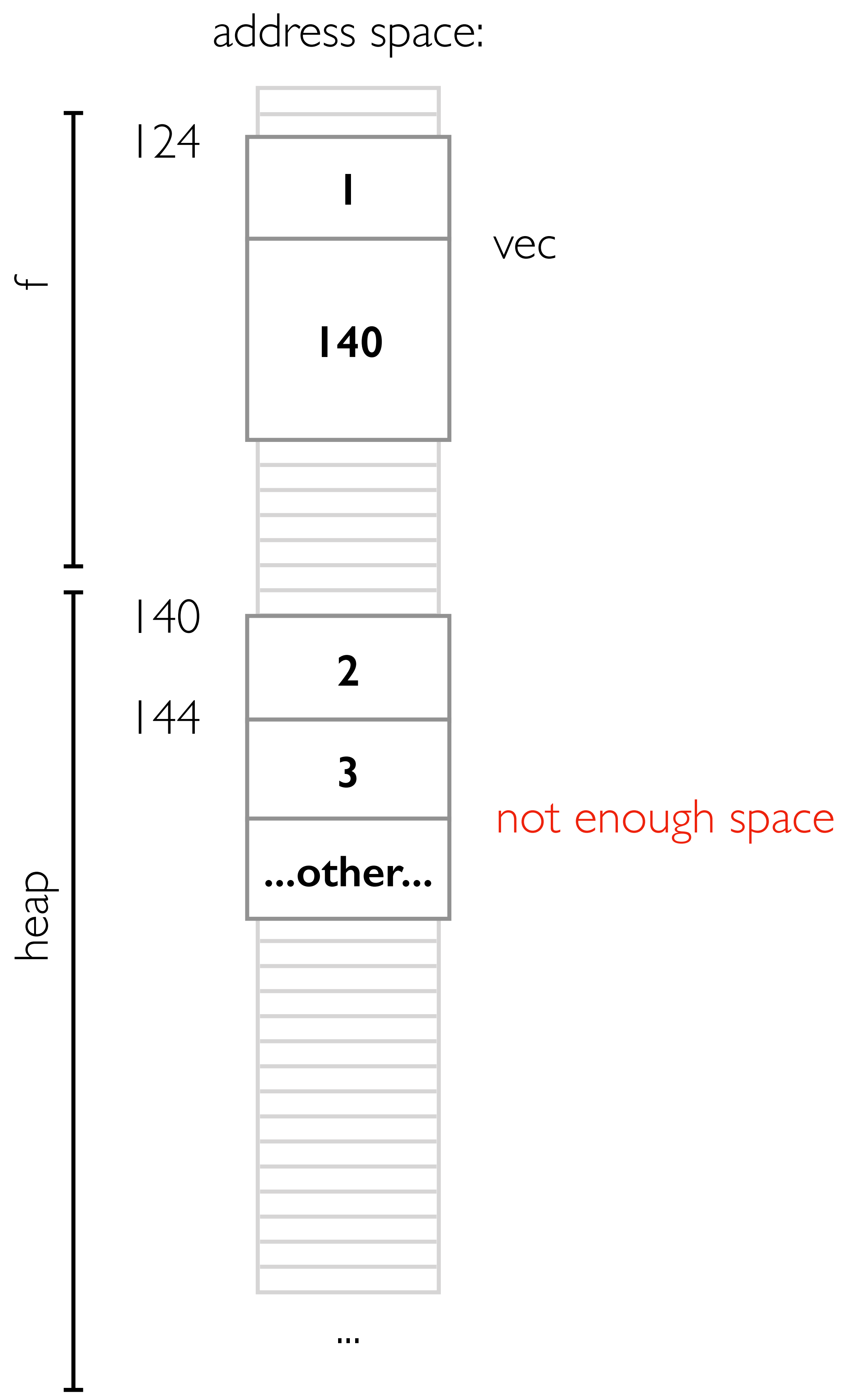
Growing a Vector

```
struct Loc {
  int x = 0;
  int y = 0;
};

int f() {
  vector<Loc> vec{{2,3}};

}
```



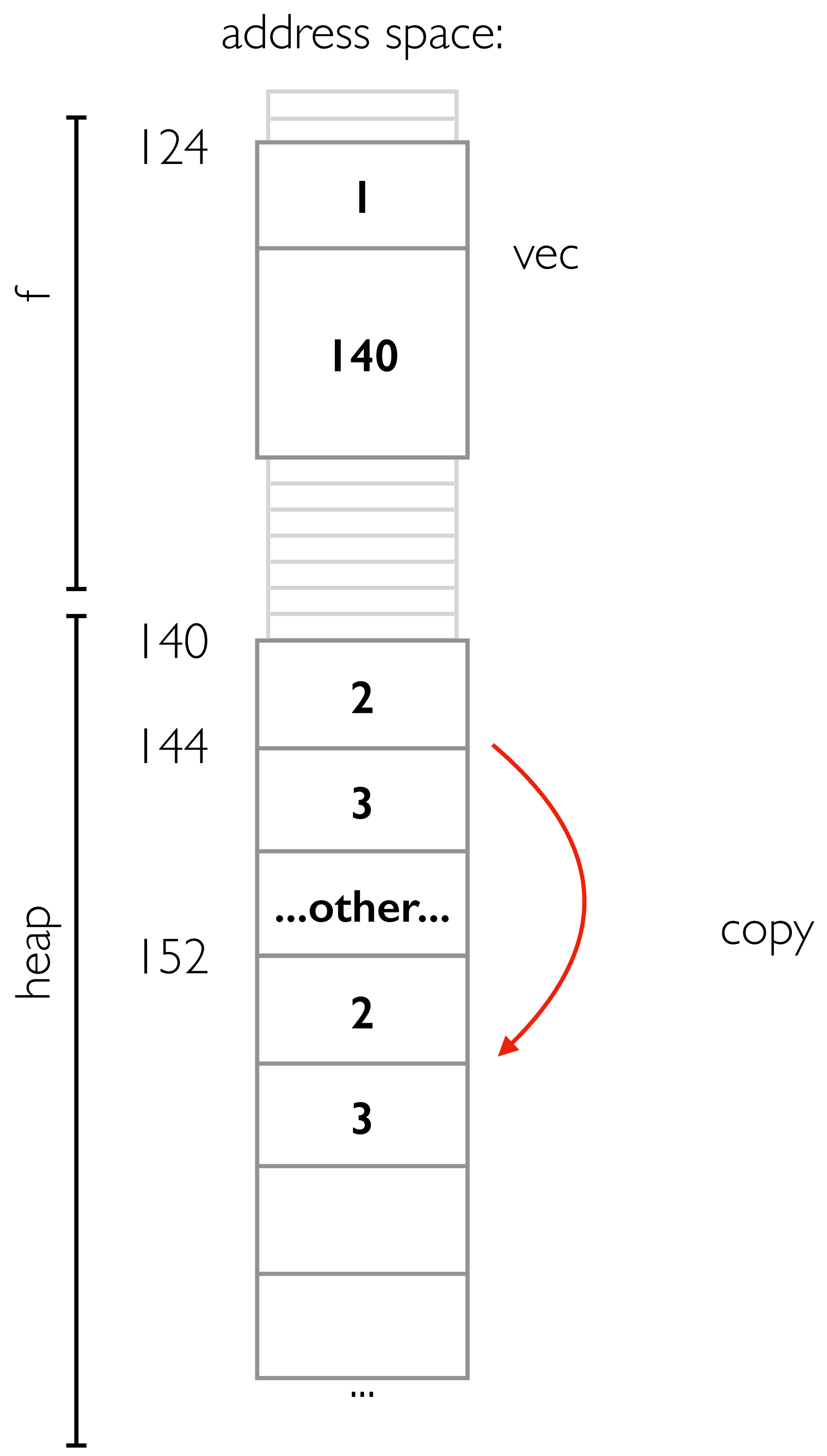
Growing a Vector

```
struct Loc {
  int x = 0;
  int y = 0;
};

int f() {
  vector<Loc> vec{{2,3}};
  vec.push_back({4,5})
}
```



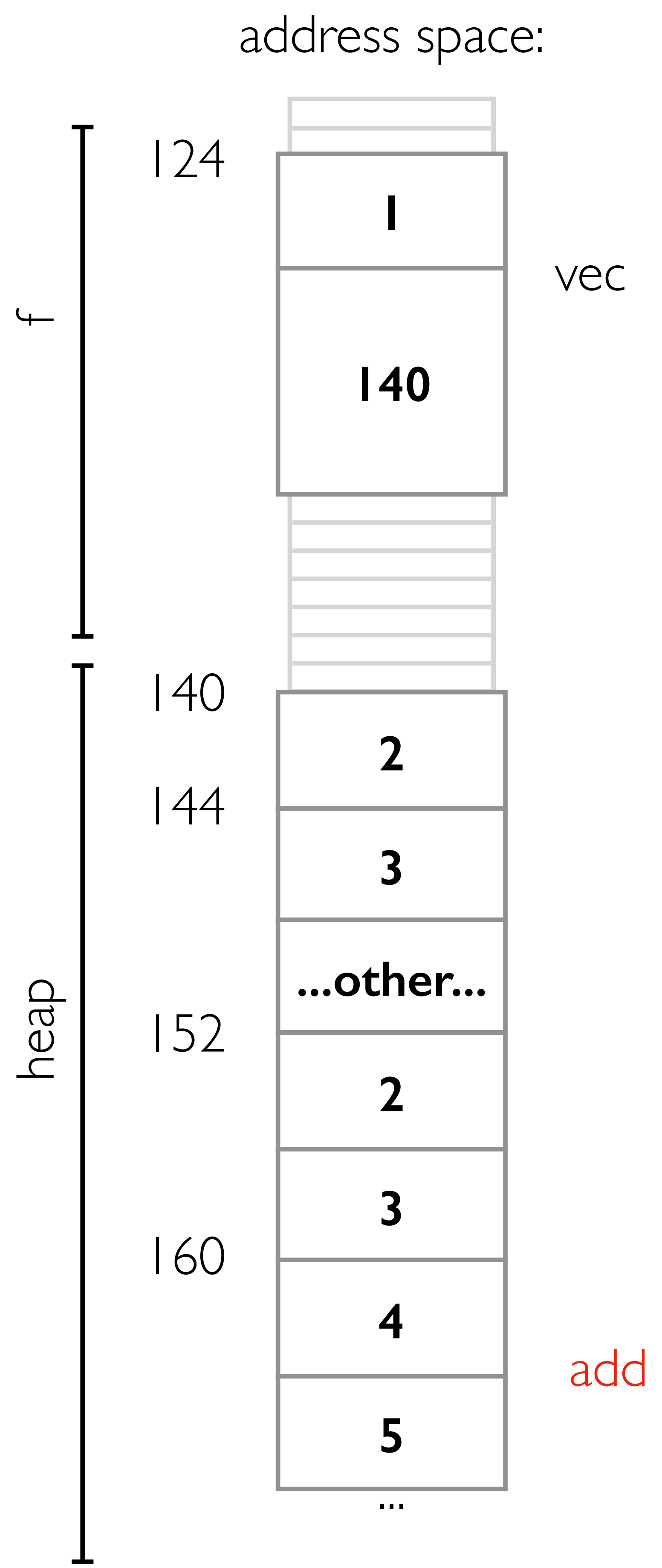
Growing a Vector

```
struct Loc {
  int x = 0;
  int y = 0;
};

int f() {
  vector<Loc> vec{{2,3}};
  vec.push_back({4,5})
}
```



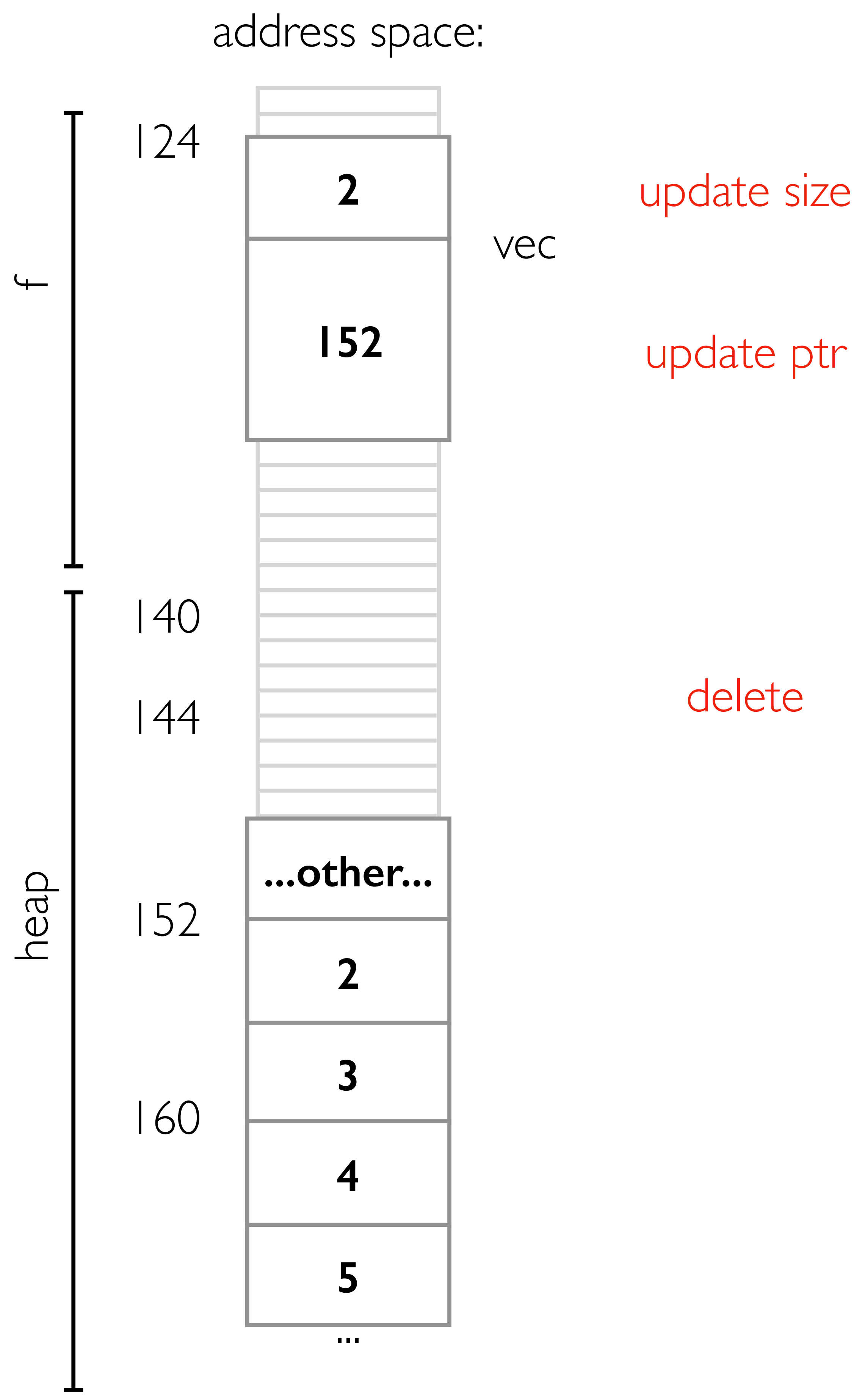
Growing a Vector

```
struct Loc {
  int x = 0;
  int y = 0;
};

int f() {
  vector<Loc> vec{{2,3}};
  vec.push_back({4,5})
}
```



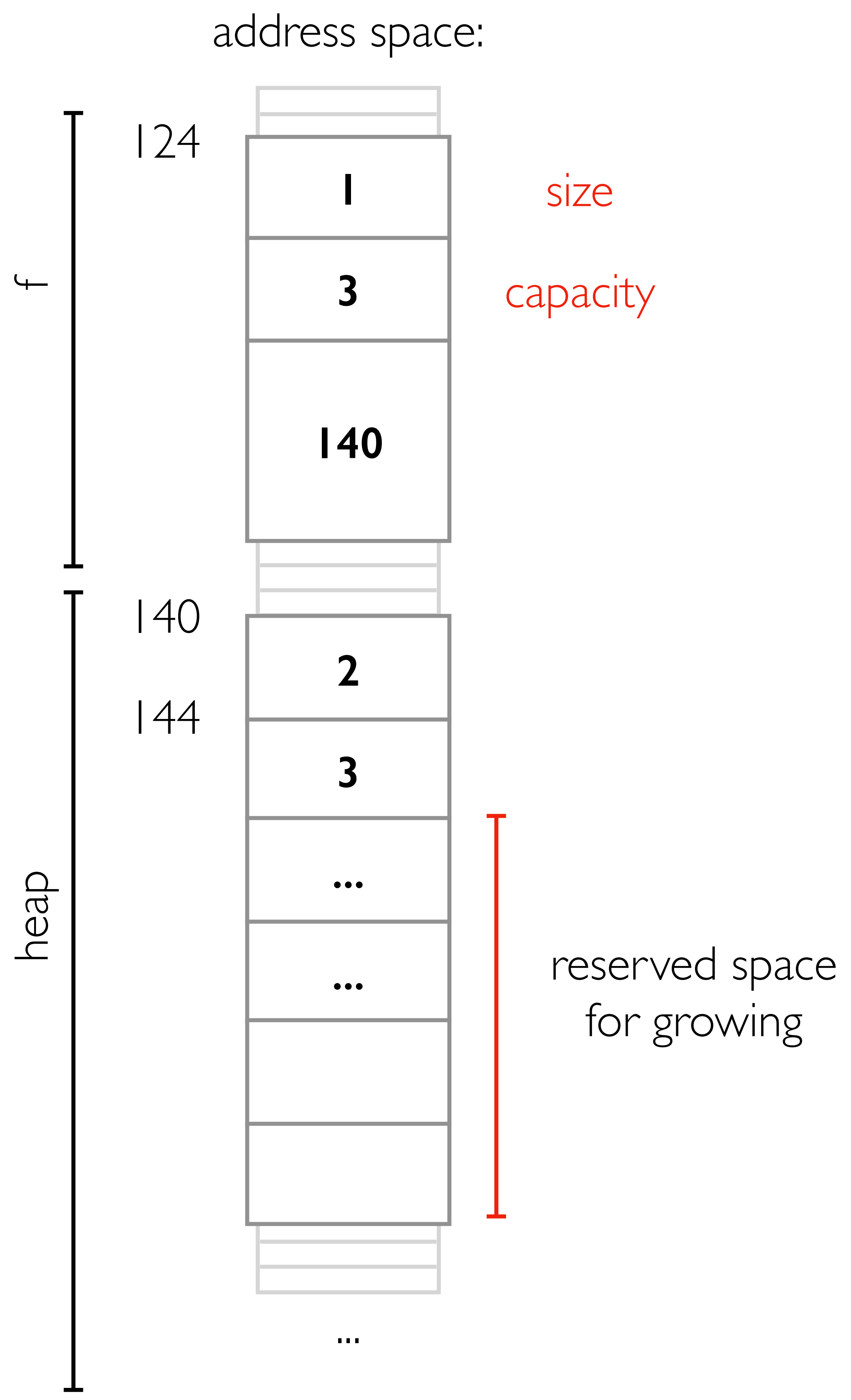
Growing a Vector

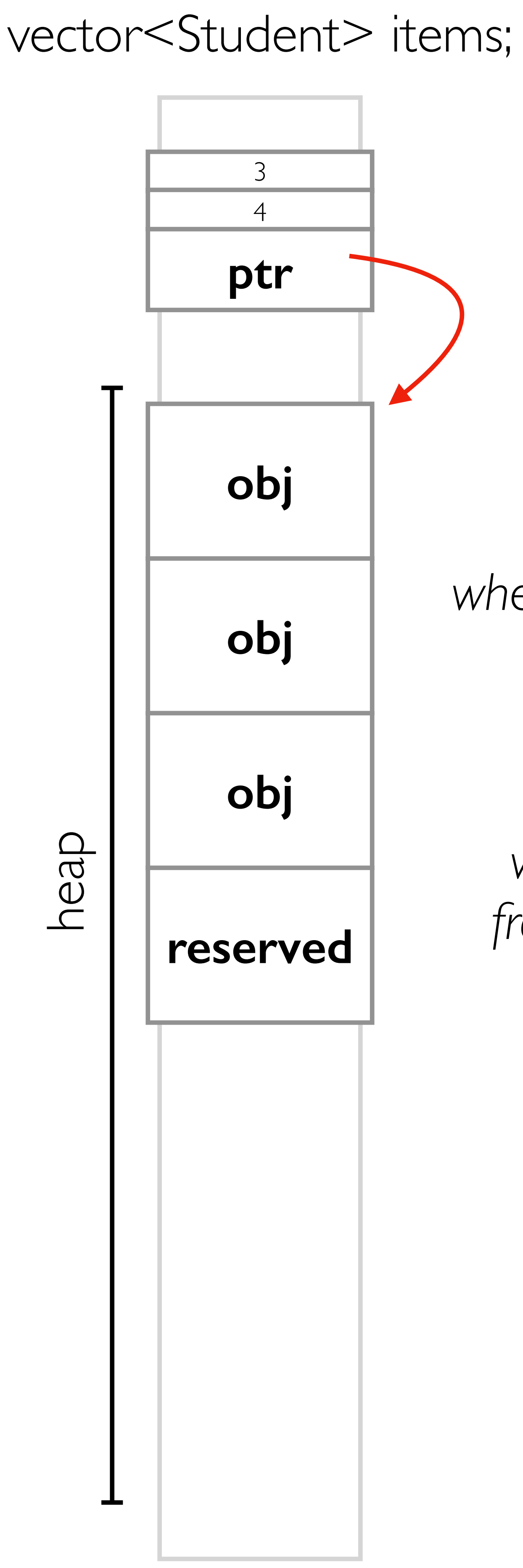
```
struct Loc {
  int x = 0;
  int y = 0;
};

int f() {
  vector<Loc> vec{{2,3}};
  vec.push_back({4,5})
}
```



Size vs. Capacity

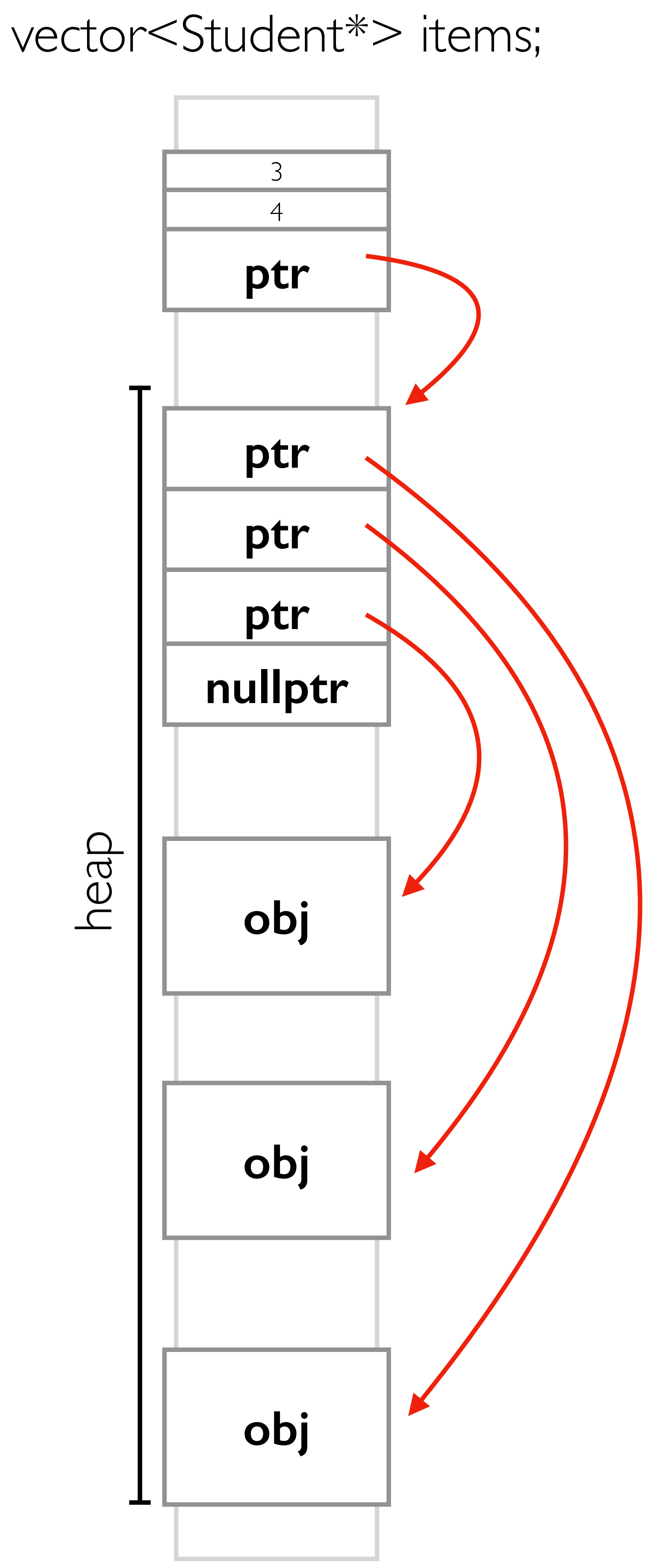
```
struct Loc {
  int x = 0;
  int y = 0;
};

int f() {
  vector<Loc> vec{{2,3}};
  vec.push_back({4,5})
}
```





discussion question 1:
when would each layout be more efficient?

discussion question 2:
when would each layout be preferable from the perspective of application logic or programmer convenience?



Vector Demos

Things to note (in the demos)

- vectors sometimes call constructors for us
- vectors sometimes call destructors for us
- we don't get automatic help if we have a vector of pointers
- there is an underlying array
- items are tightly packed
- bounds checking is optional
- size is not capacity
- if you use a vector to store actual values, be careful about secondary references to those values if the size can change!
- resizing involves copying, which your objects must support!
- use reserve and emplace to minimize copying
- initialization style affects which overloaded constructor is chosen
- assignment+init do copy by default
- iterators give more flexibility than for-each loops
- ranges are replacing iterators in many cases, starting in C++20

Outline

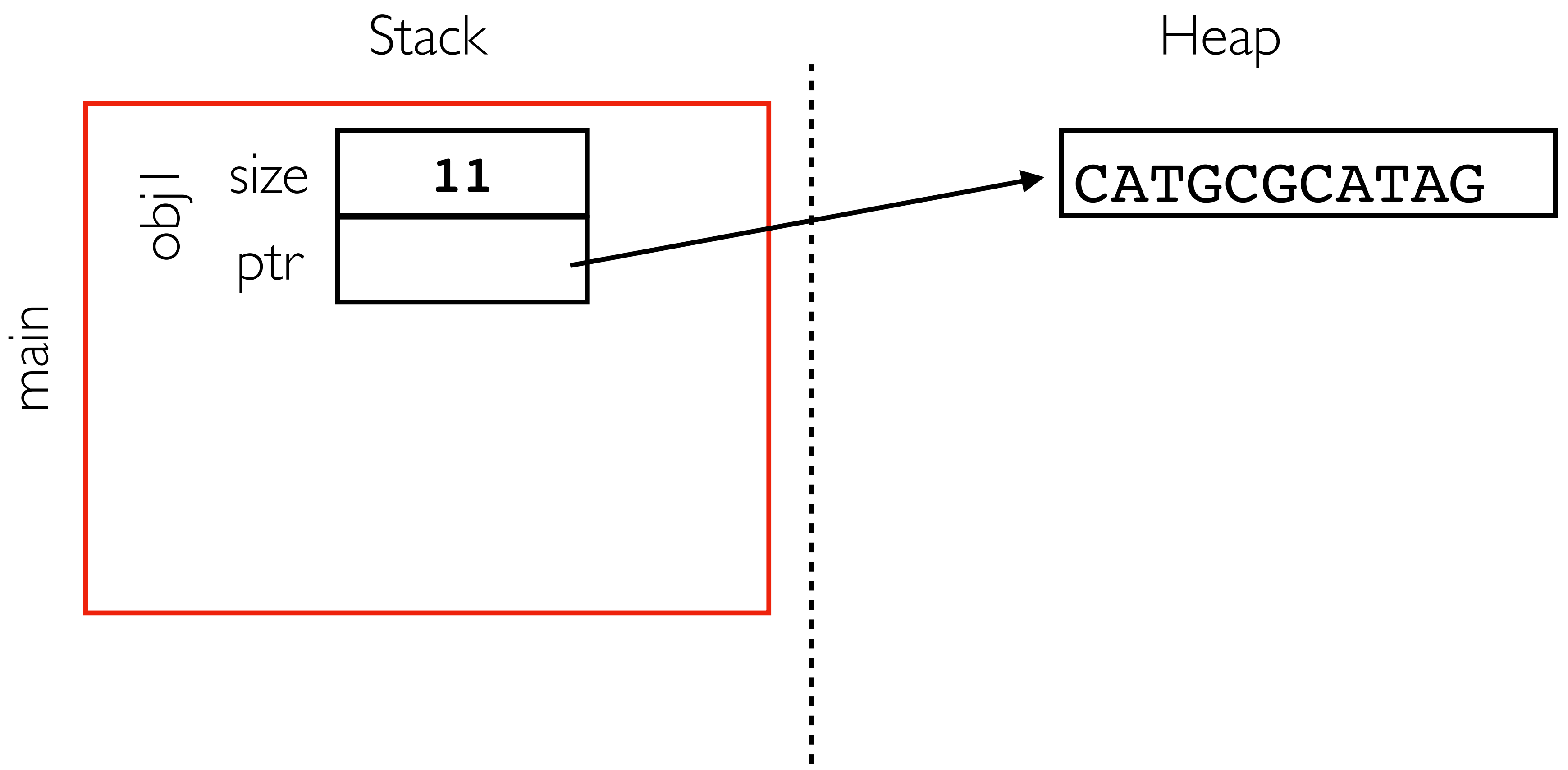
Worksheet and TopHat

Vectors

Moving vs. Copying

Copy Review

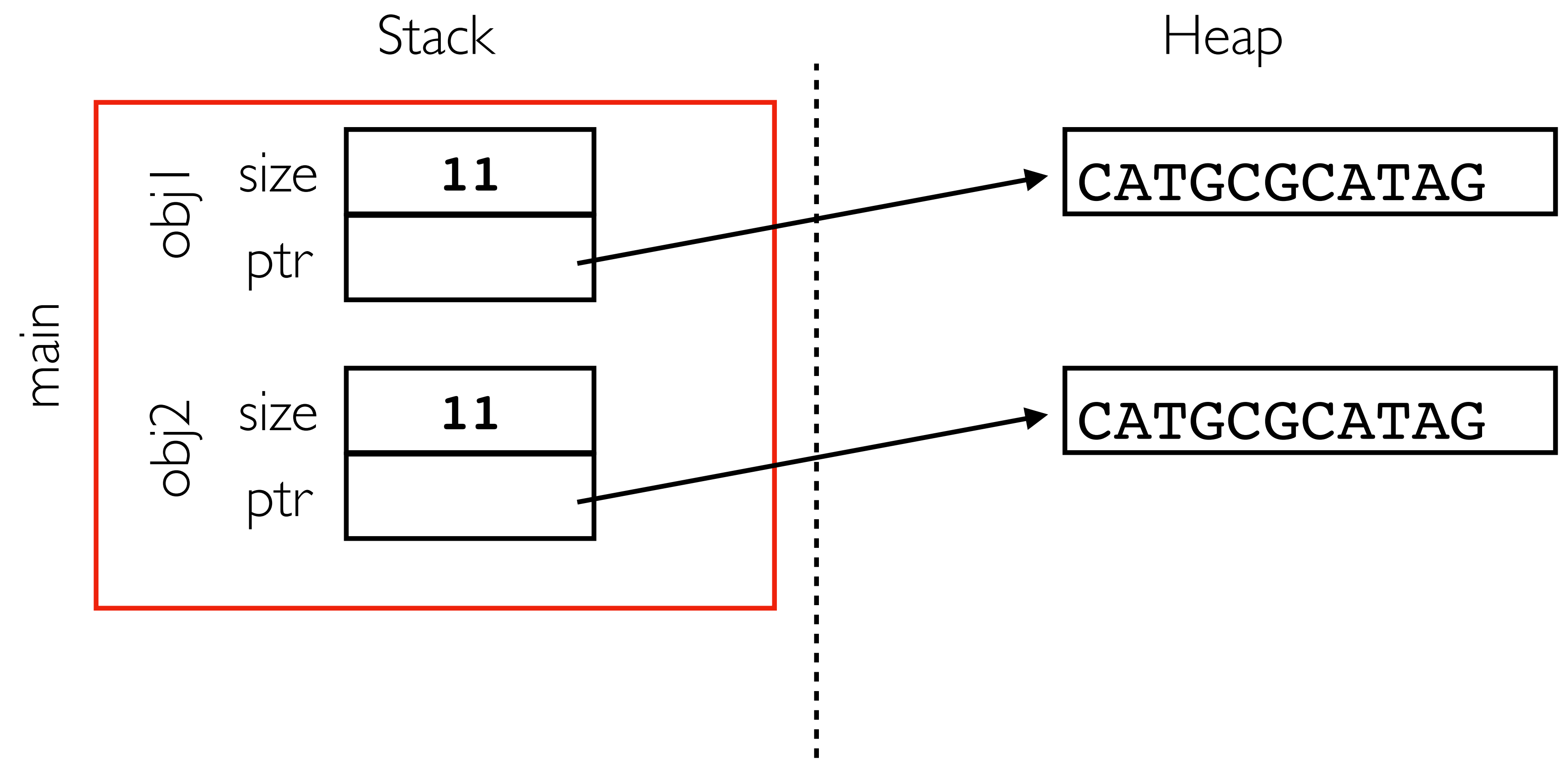
```
// CODE:
DNA obj1{"CATGCGCATAG"};
```



Copy Review

```
// CODE:
DNA obj1{"CATGCGCATAG"};
DNA obj2 = obj1;
```

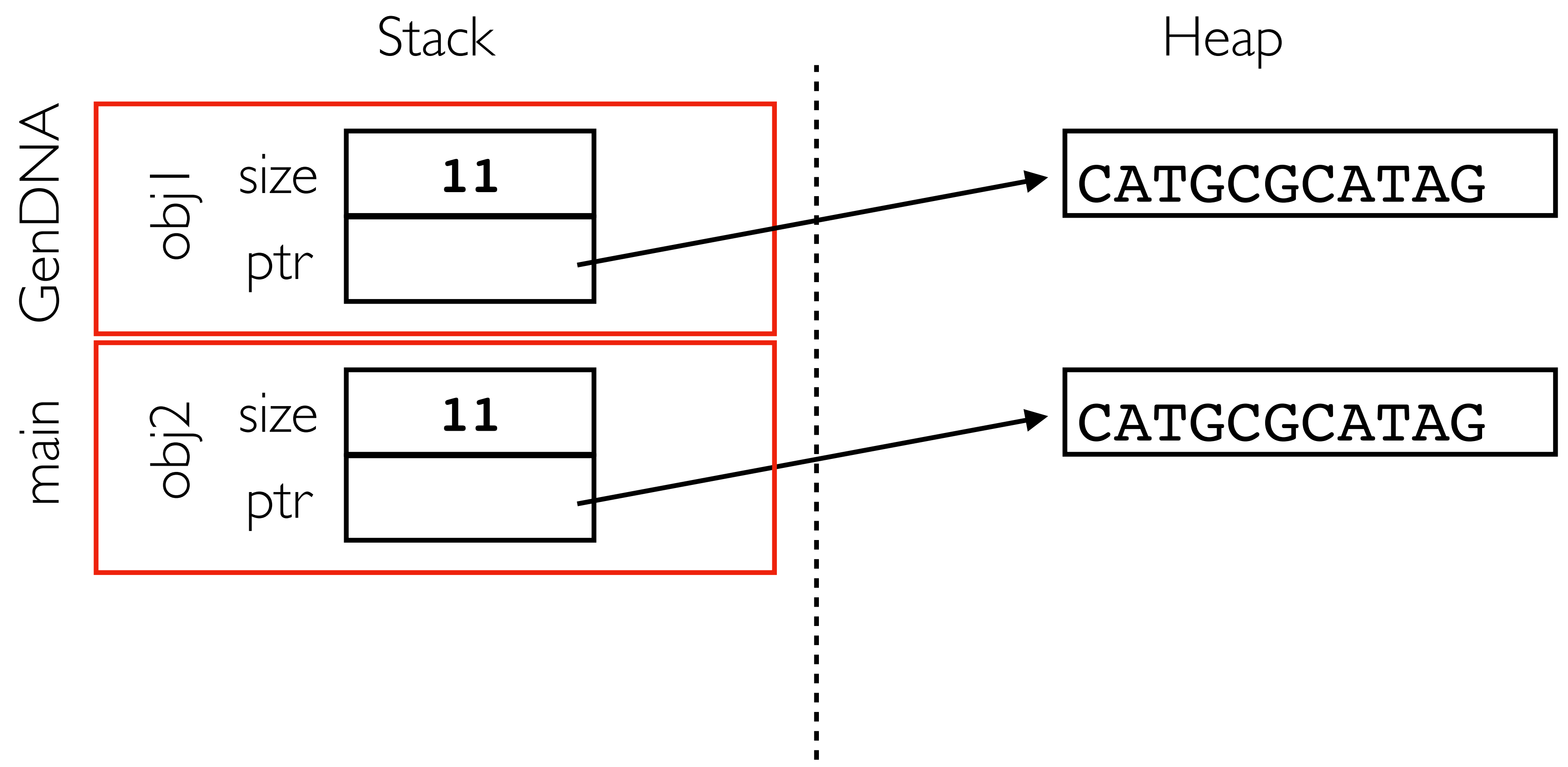
we could write a copy constructor to get this behavior



Copying Upon Return

```
DNA GenDNA() {
  DNA obj1{"CATGCGCATAG"};
  return obj1;
}

void main() {
  DNA obj2 = GenDNA();
}
```

copying is wasteful here!

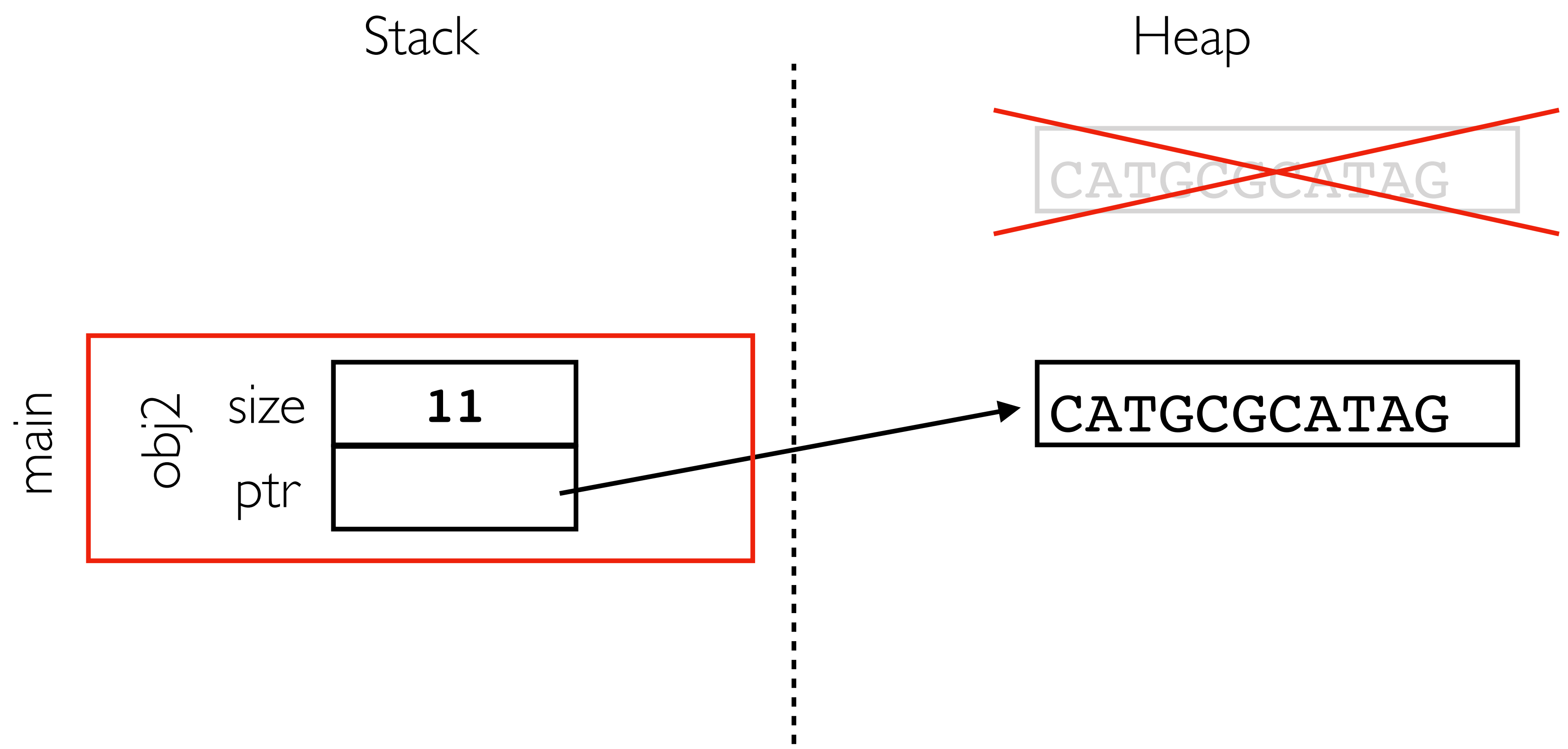


Copying Upon Return

```
DNA GenDNA() {
  DNA obj1{"CATGCGCATAG"};
  return obj1;
}

void main() {
  DNA obj2 = GenDNA();
}
```

copying is wasteful here!

we immediately discard one of the copies

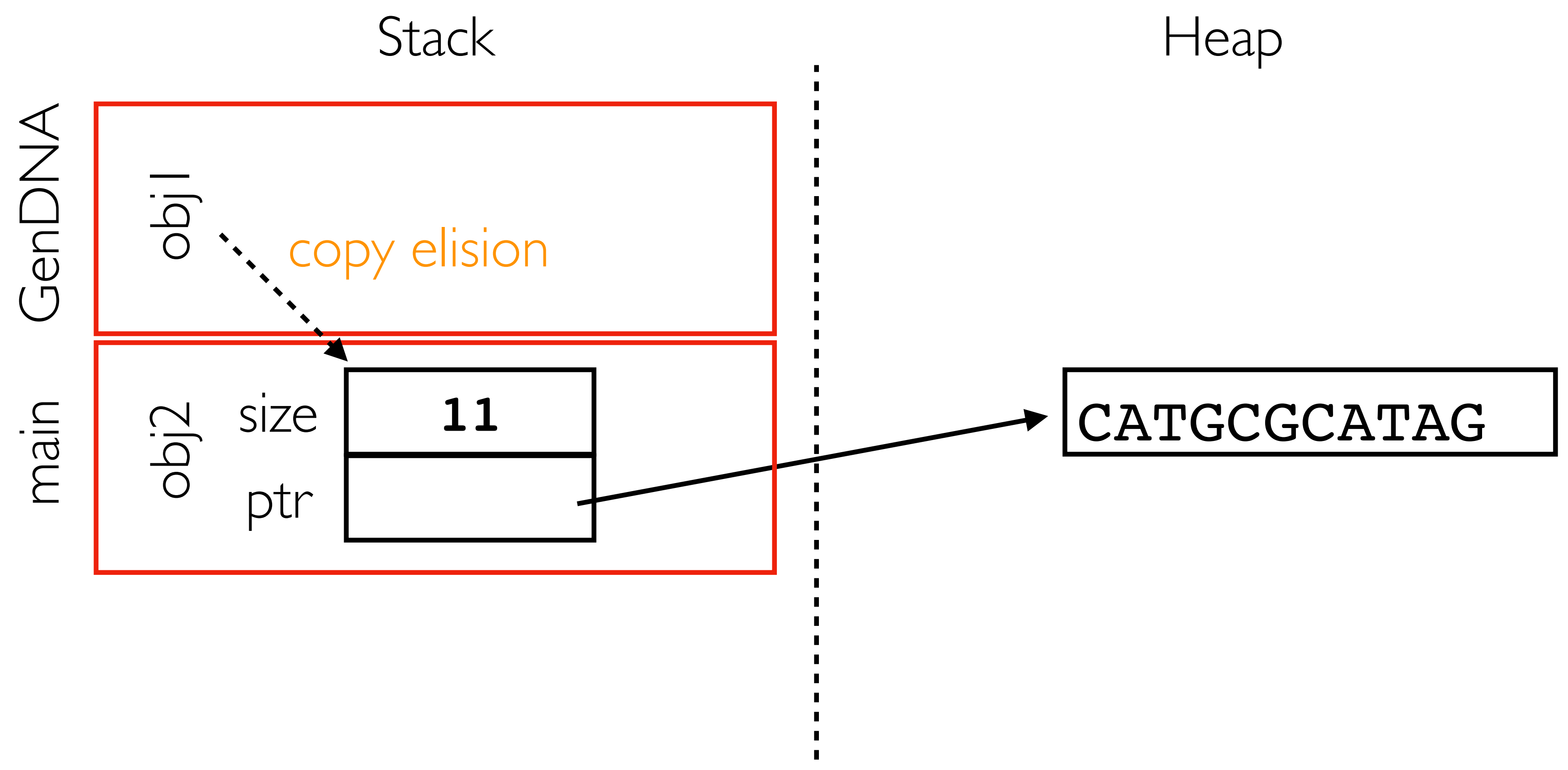


Copy Elision

```
DNA GenDNA() {
  DNA obj1{"CATGCGCATAG"};
  return obj1;
}

void main() {
  DNA obj2 = GenDNA();
}
```

https://en.cppreference.com/w/cpp/language/copy_elision

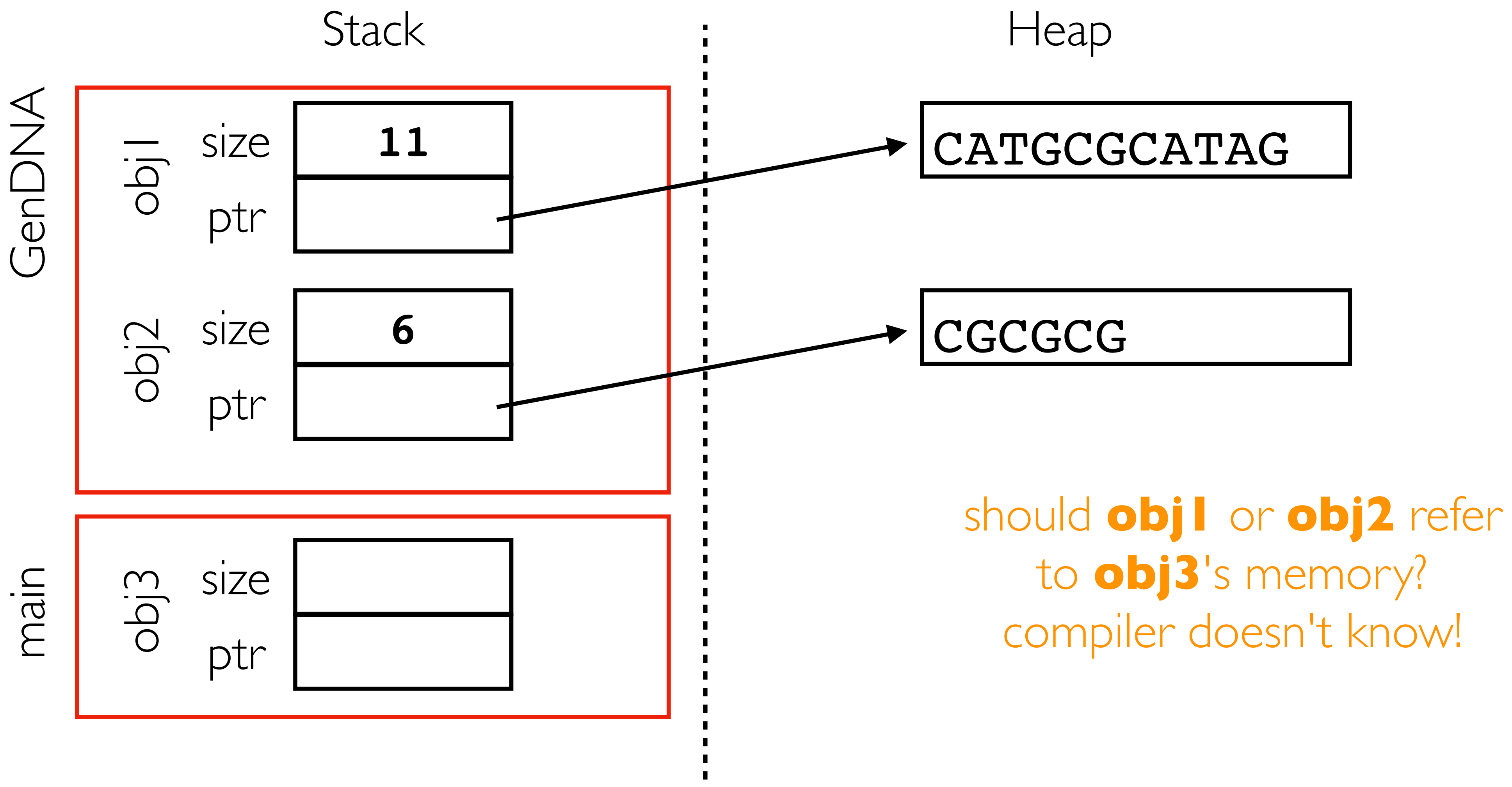
- this is an old optimization many compilers chose to do
- since C++17, it is mandatory is some scenarios



Trickier Returns

```
DNA GenDNA() {
  DNA obj1{"CATGCGCATAG"};
  DNA obj2{"CGCGCG"};
  if (???)
    return obj1;
  else
    return obj2;
}
```

```
void main() {
  DNA obj3 = GenDNA();
}
```

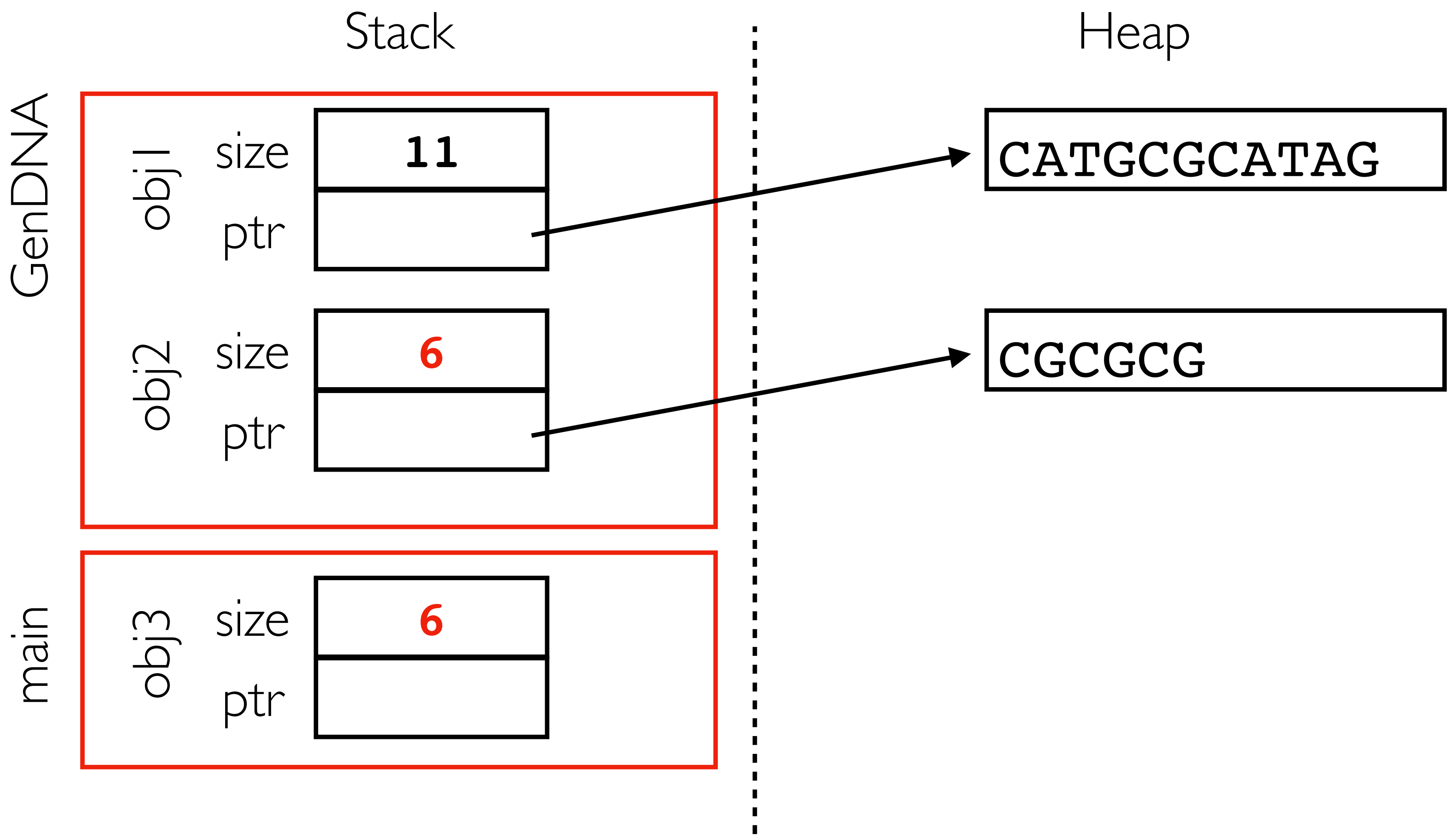


should **obj1** or **obj2** refer to **obj3**'s memory? compiler doesn't know!

Move Semantics

```
DNA GenDNA() {
  DNA obj1{"CATGCGCATAG"};
  DNA obj2{"CGCGCG"};
  if (???)
    return obj1;
  else
    return obj2;
}
```

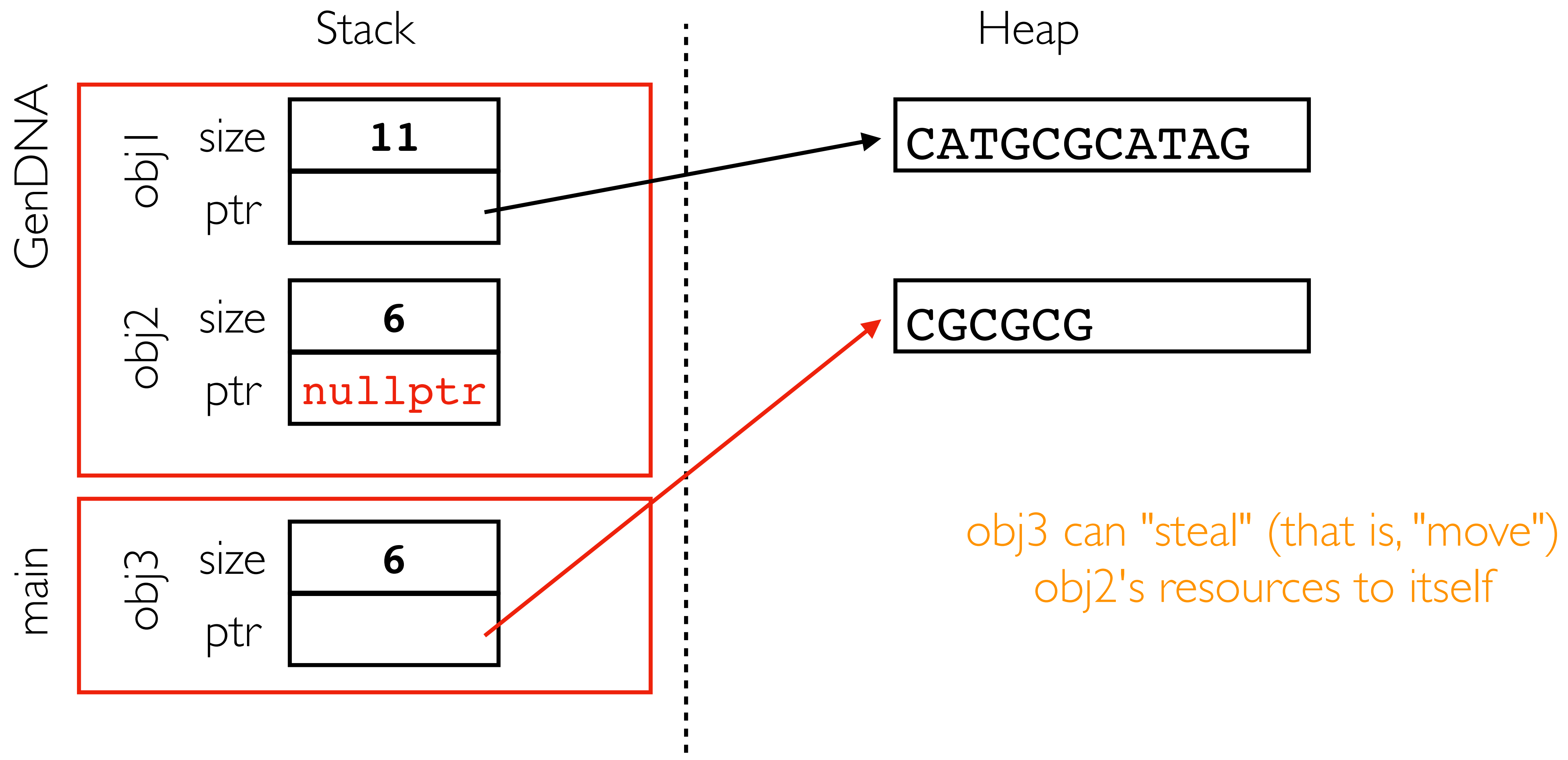
```
void main() {
  DNA obj3 = GenDNA();
}
```



Move Semantics

```
DNA GenDNA() {
  DNA obj1{"CATGCGCATAG"};
  DNA obj2{"CGCGCG"};
  if (???)
    return obj1;
  else
    return obj2;
}
```

```
void main() {
  DNA obj3 = GenDNA();
}
```

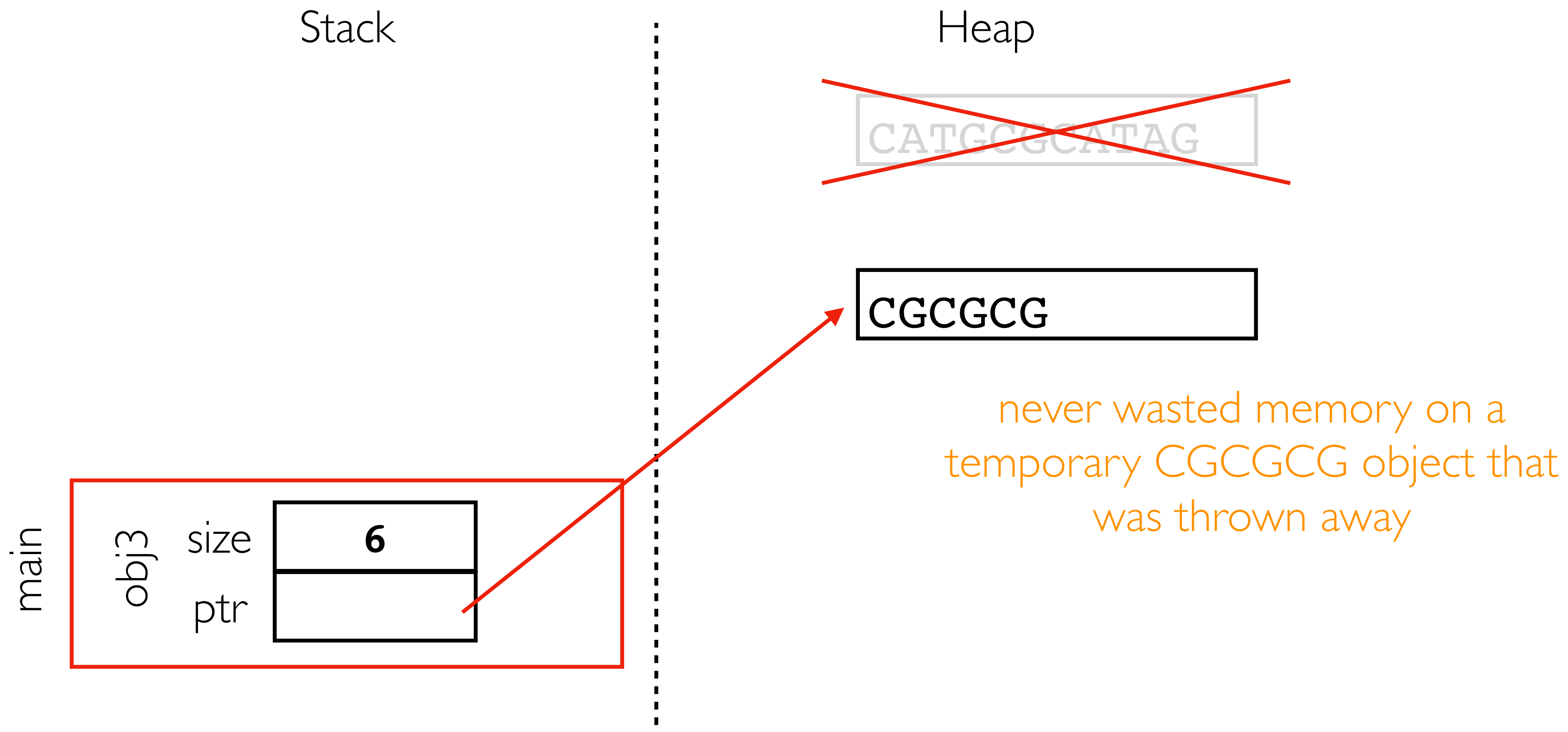


obj3 can "steal" (that is, "move") obj2's resources to itself

Move Semantics

```
DNA GenDNA() {
  DNA obj1{"CATGCGCATAG"};
  DNA obj2{"CGCGCG"};
  if (???)
    return obj1;
  else
    return obj2;
}
```

```
void main() {
  DNA obj3 = GenDNA();
}
```


Stack
Heap
~~CATGCGCATAG~~
CGCGCG
main
obj3
size
6
ptr
never wasted memory on a temporary CGCGCG object that was thrown away

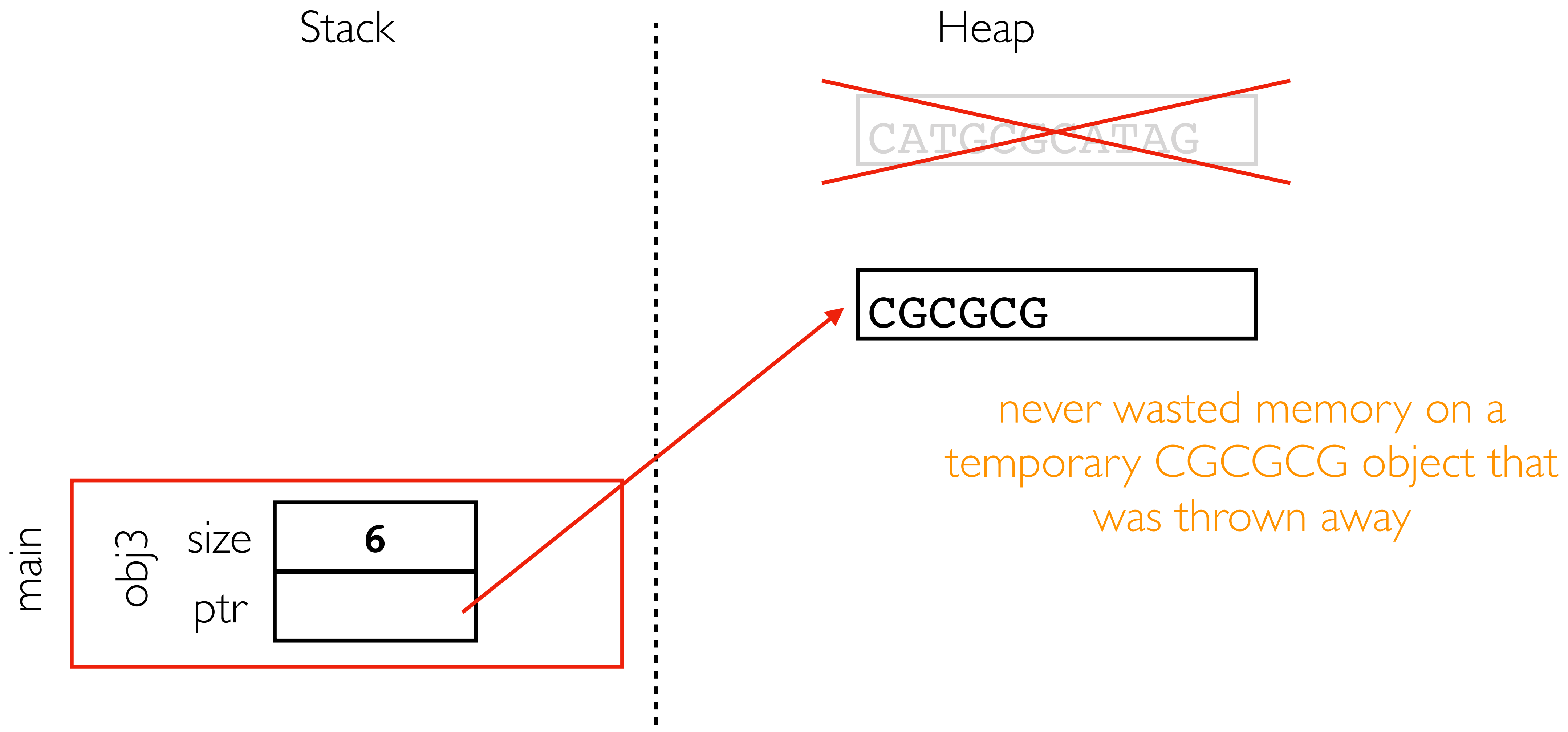
Move Semantics

```
DNA GenDNA() {
  DNA obj1{"CATGCGCATAG"};
  DNA obj2{"CGCGCG"};
  if (???)
    return obj1;
  else
    return obj2;
}
```

```
void main() {
  DNA obj3 = GenDNA();
}
```

compilers often move automatically for us upon return!

there are other cases where we might want move semantics, but we'll need to be explicit about it



never wasted memory on a temporary CGCGCG object that was thrown away

When is Move OK?

Big question: can we "steal" the contents of an object without breaking anything?

Meaning of reference types

- lvalue reference: stealing is NOT ok
- rvalue reference: take what you want!

Syntax

- lvalue reference: MyClass& obj
- rvalue reference: MyClass&& obj

std::move(obj)

- cast obj to an rvalue reference
- don't actually move anything!
- overloaded functions can have different behaviors depending on reference type

Demos...