[368] More Memory

Tyler Caraza-Harter

What will you learn today?

Learning objectives

- describe memory layout
- decide when to use stack or heap for a particular piece of data
- use new/delete correctly (avoiding memory bugs such as segfaults and leaks)
- writer safer code with const and references

Outline

TopHat and Worksheet

Memory Layout: Code/Stack/Heap

new/delete

arrays

Worksheet

Safety

Outline

TopHat and Worksheet

Memory Layout: Code/Stack/Heap

new/delete

arrays

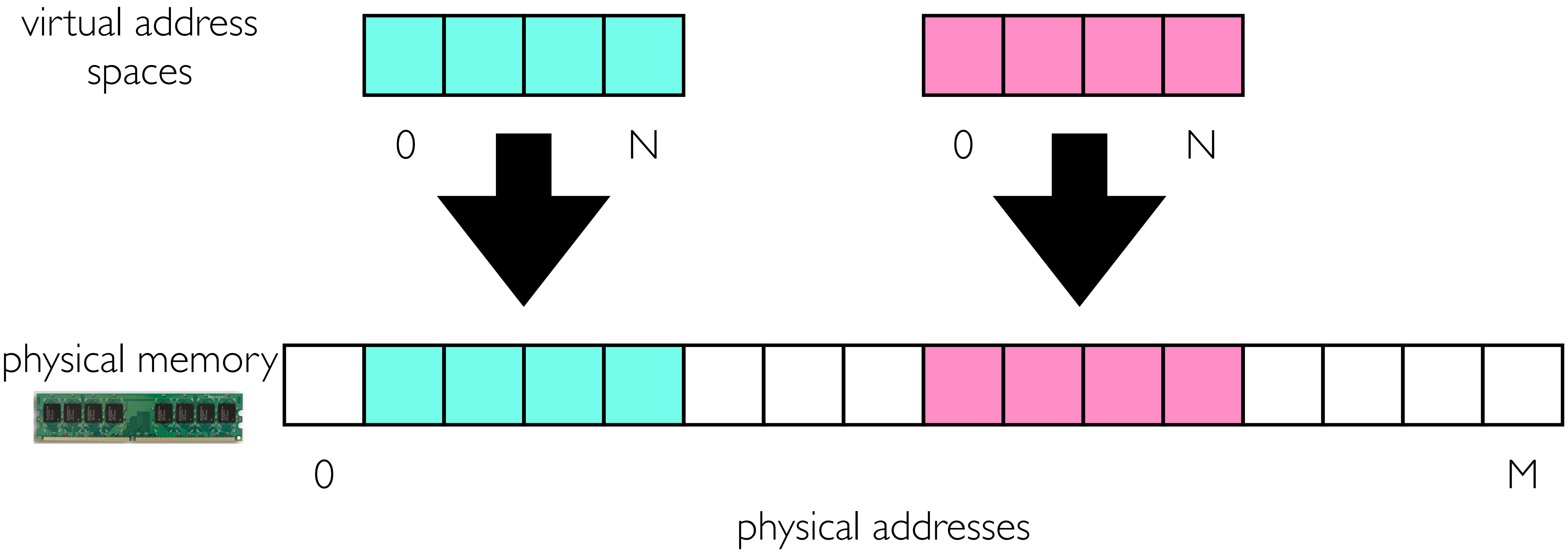
Worksheet

Safety

Processes and Address Spaces

Address spaces

- A process is a running program
- Each process has it's own virtual address space
- The same virtual address generally refers to different memory in different processes
- Regular processes cannot directly access physical memory or other addr spaces



virtual address spaces

0 N

0 N

physical memory

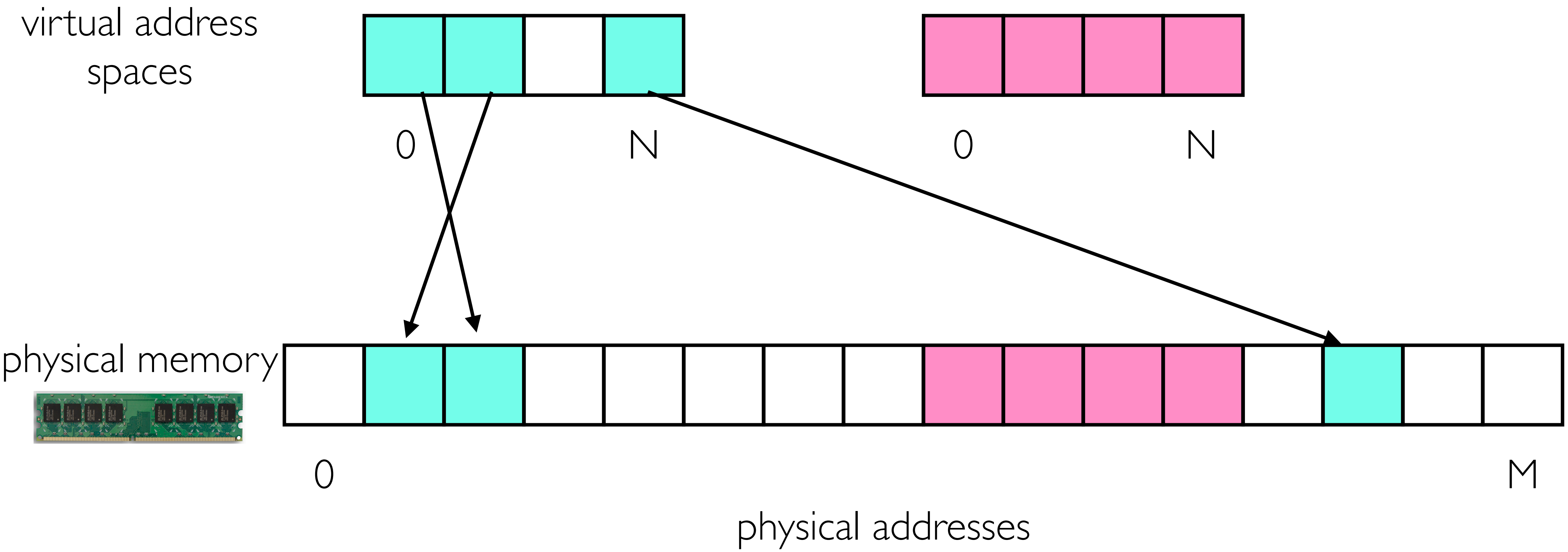
0 M

physical addresses

Processes and Address Spaces

Address spaces

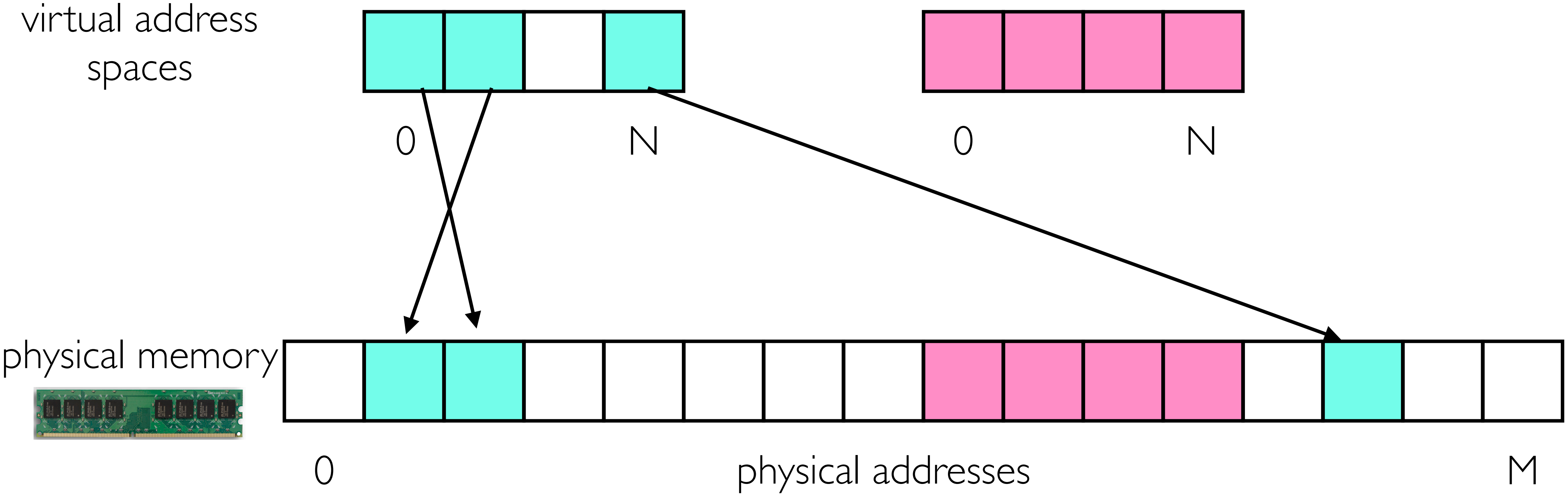
- A process is a running program
- Each process has it's own virtual address space
- The same virtual address generally refers to different memory in different processes
- Regular processes cannot directly access physical memory or other addr spaces
- Address spaces can have holes (N is usually MUCH bigger than M)
- Physical memory for a process need not be contiguous



Pages

Address spaces

- pages (usually 4 KB) of memory are mapped to physical memory
- UNIX operating systems provide mmap (memory map) call to fill addr space



virtual address spaces

0 N

0 N

physical memory

0

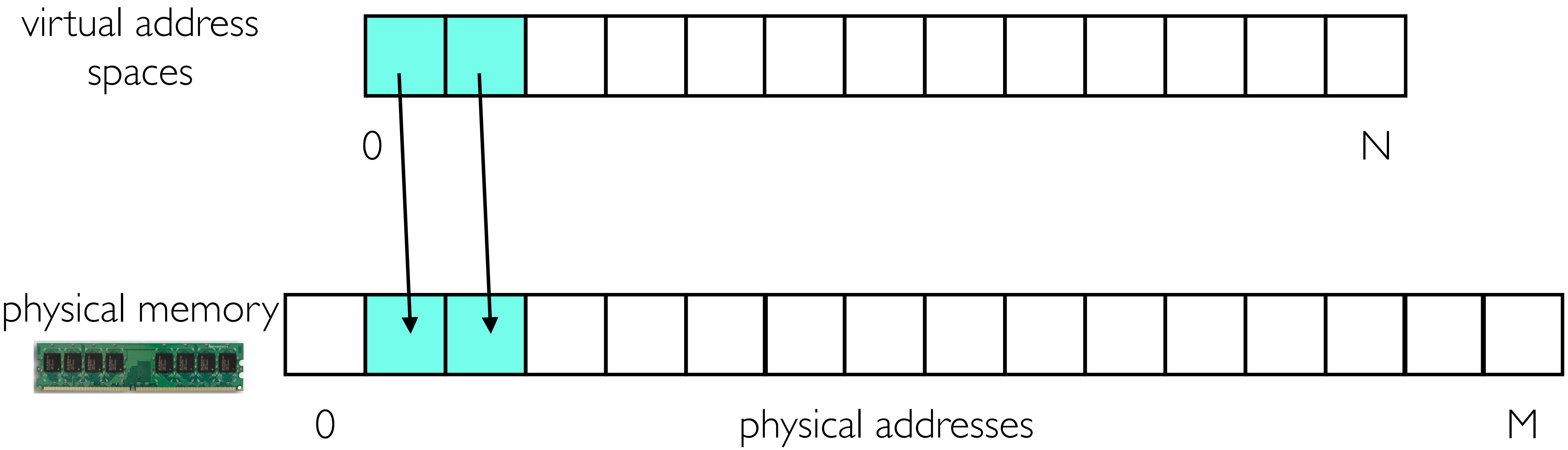
physical addresses

M

mmap (Memory Map)

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file



virtual address spaces

0

N

physical memory

0

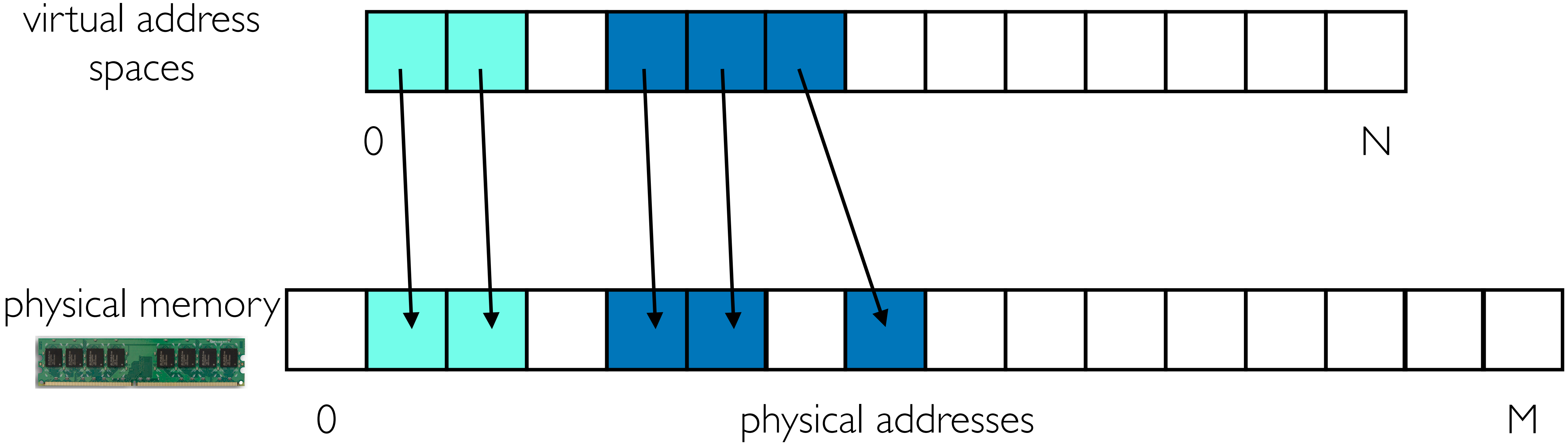
physical addresses

M

Anonymous mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file



virtual address spaces

0

N

physical memory

0

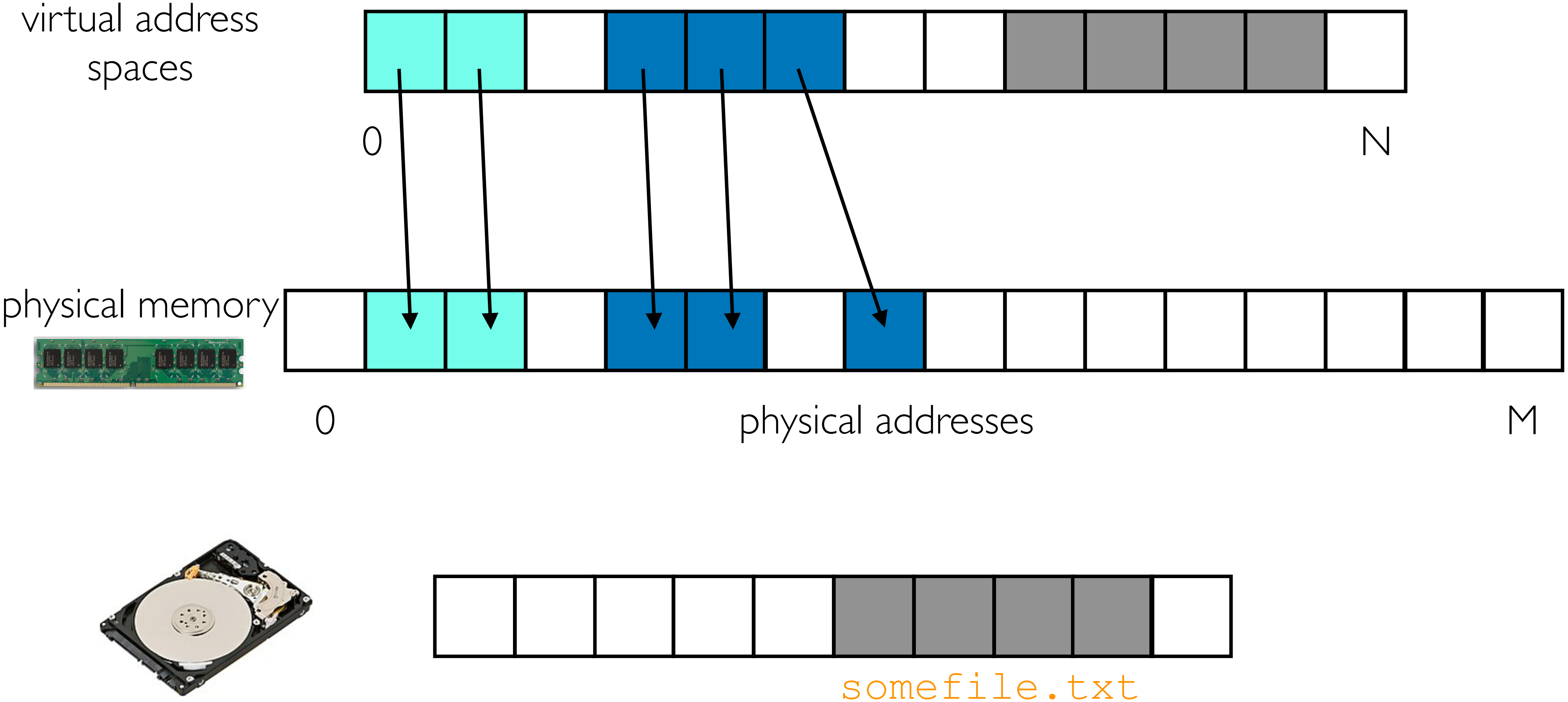
physical addresses

M

File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:

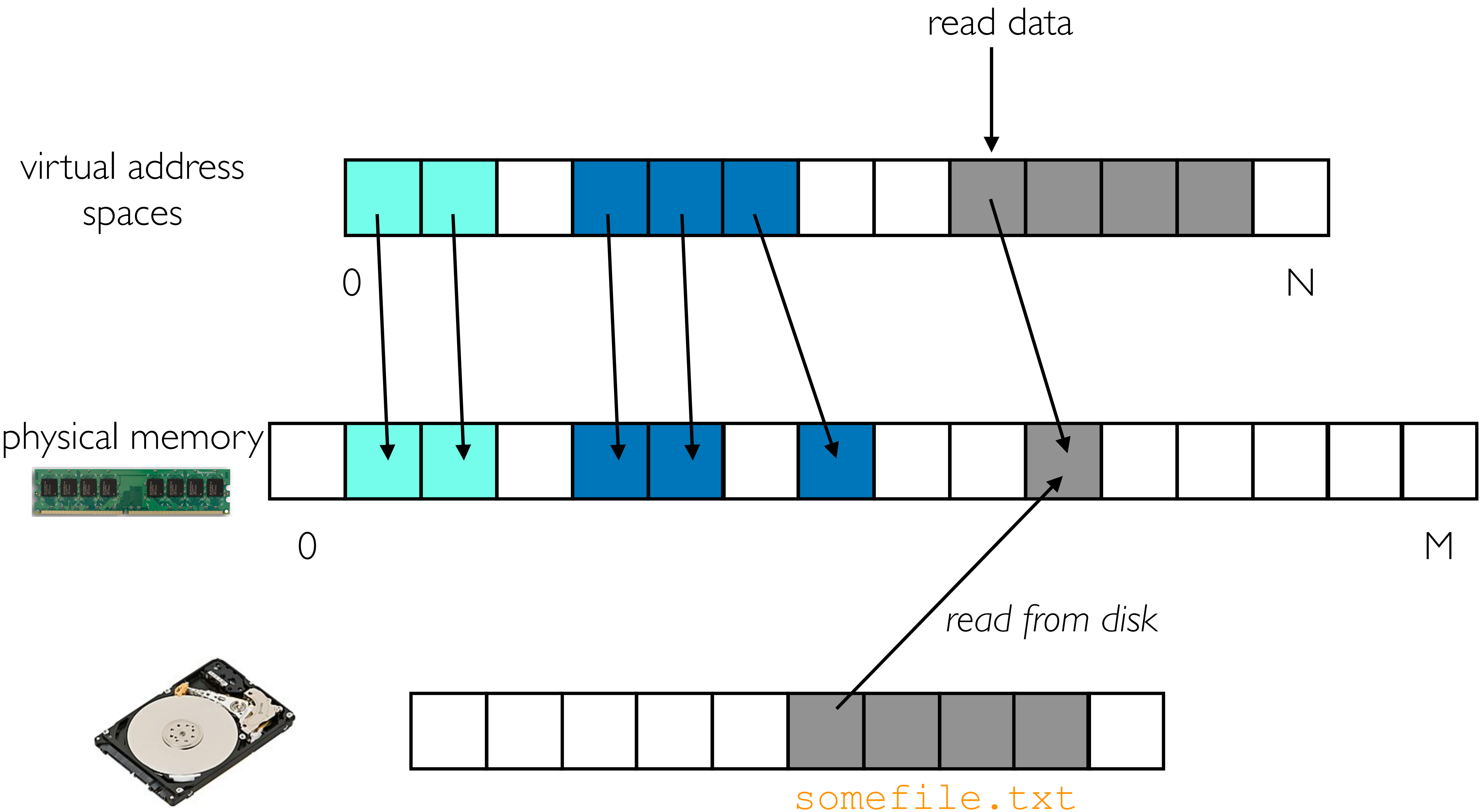
- anonymous
- backed by a file



File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file



read data

virtual address spaces

0

N

physical memory

0

M

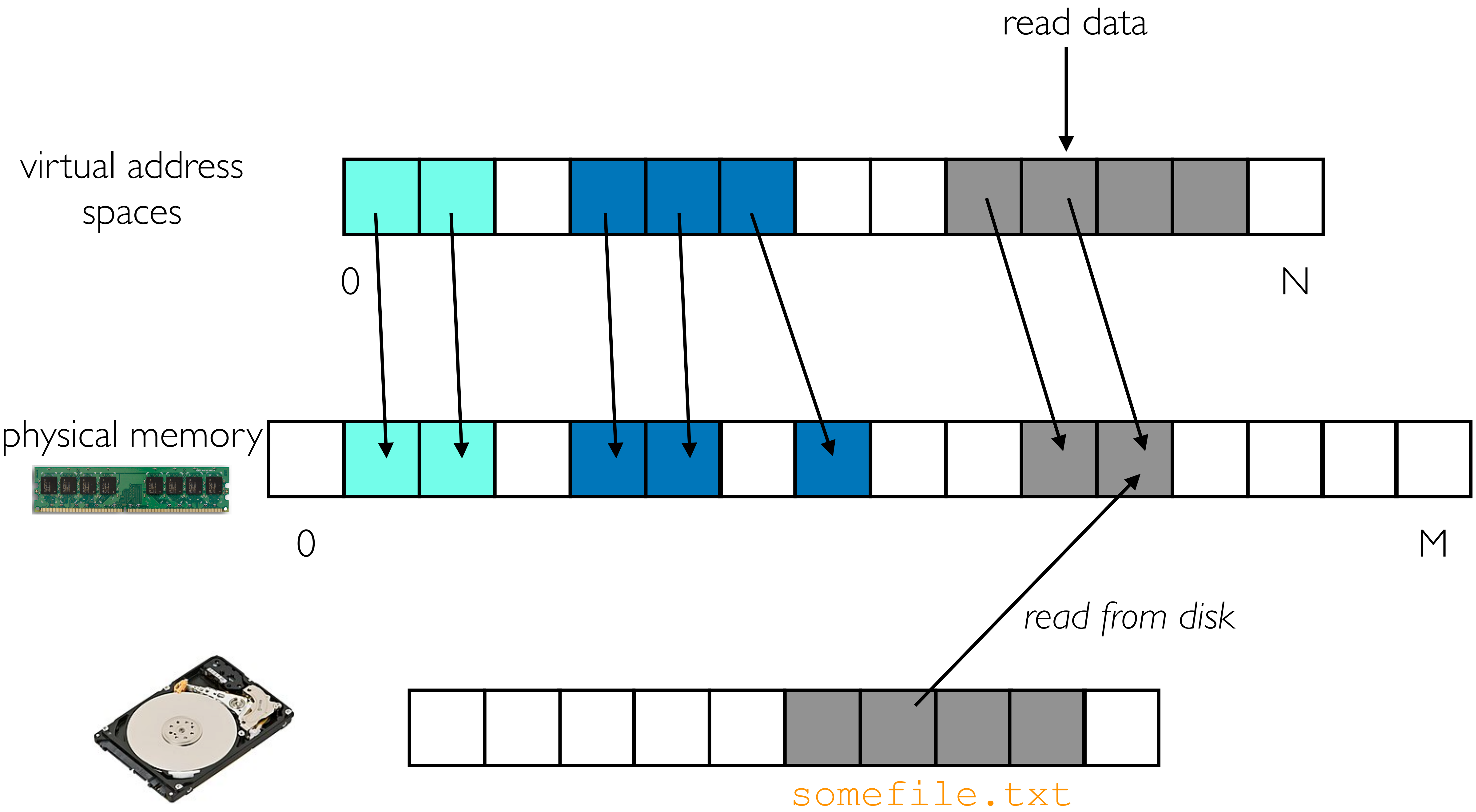
read from disk

somefile.txt

File-Backed mmap

An mmap call can add new regions to a virtual address space. Two varities:

- anonymous
- backed by a file



read data

virtual address spaces

0

N

physical memory

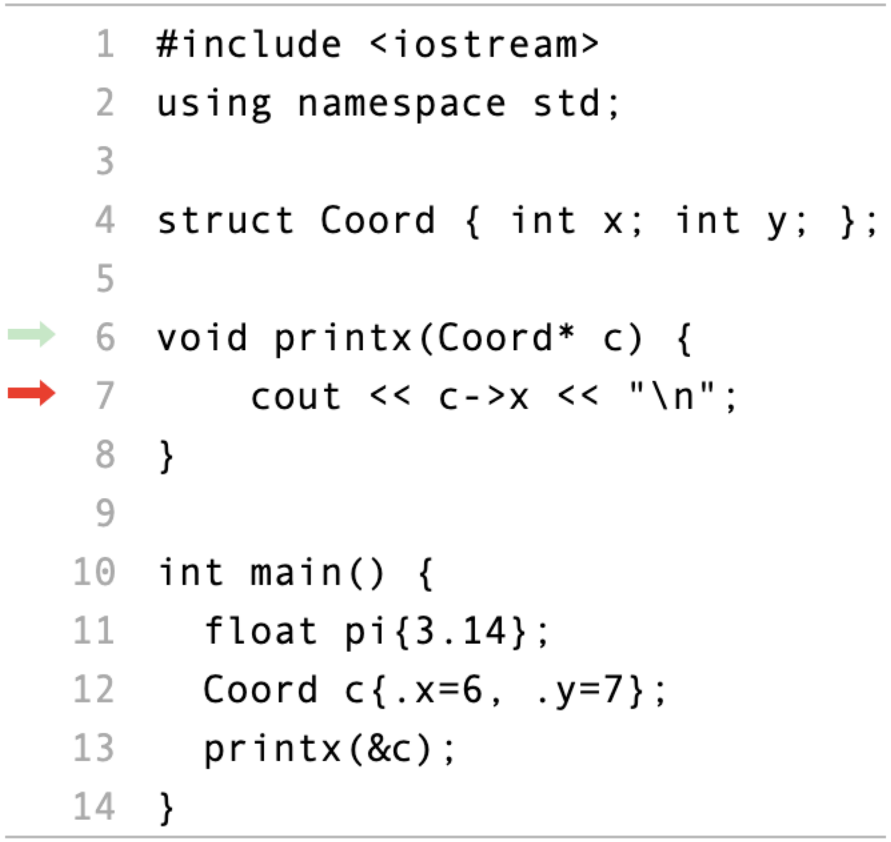
0

M

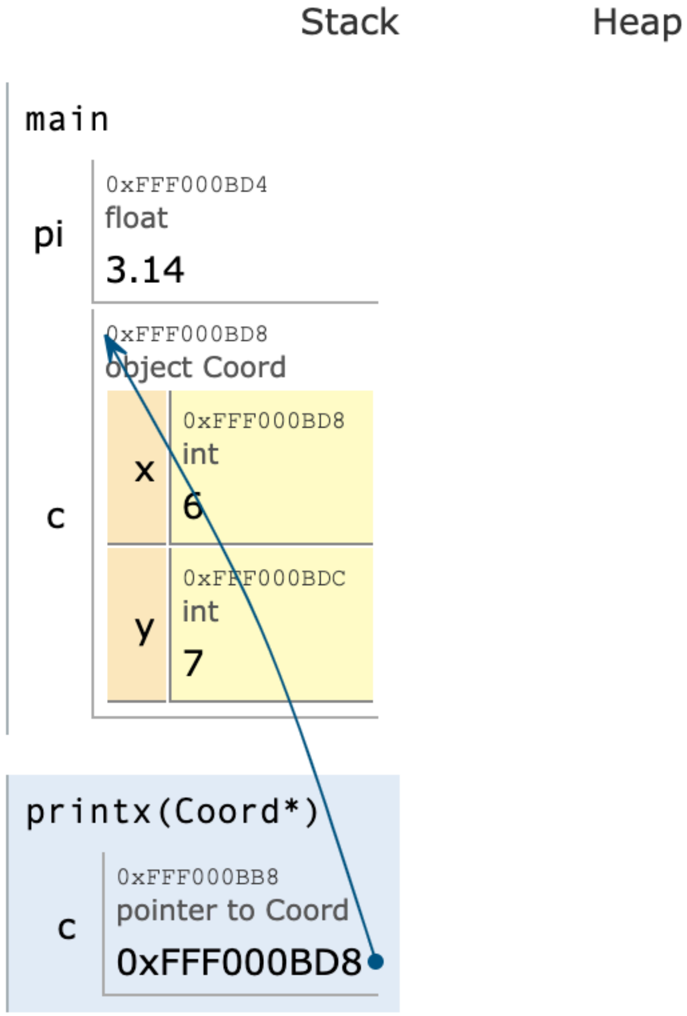
read from disk

`somefile.txt`

What goes in an address space?

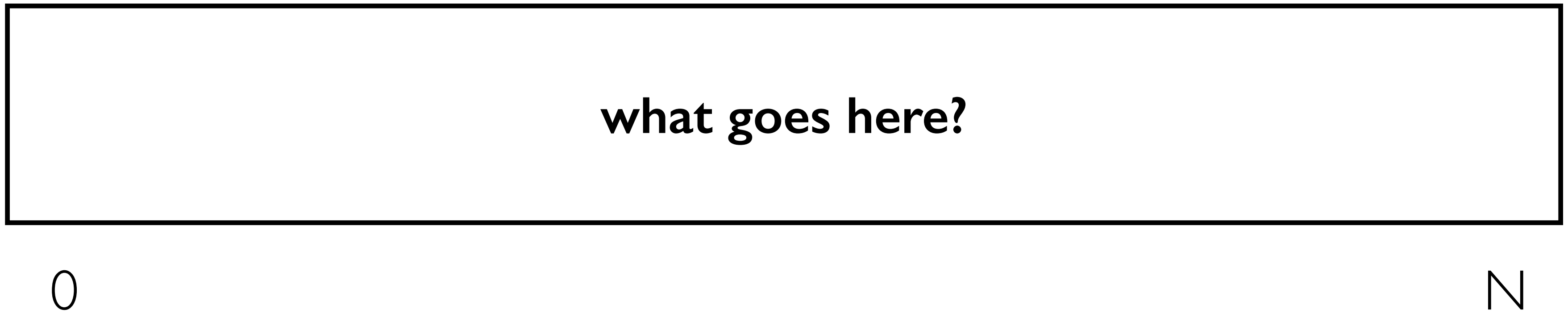


```
#include <iostream>
using namespace std;

struct Coord { int x; int y; };

void printx(Coord* c) {
    cout << c->x << "\n";
}

int main() {
  float pi{3.14};
  Coord c{.x=6, .y=7};
  printx(&c);
}
```

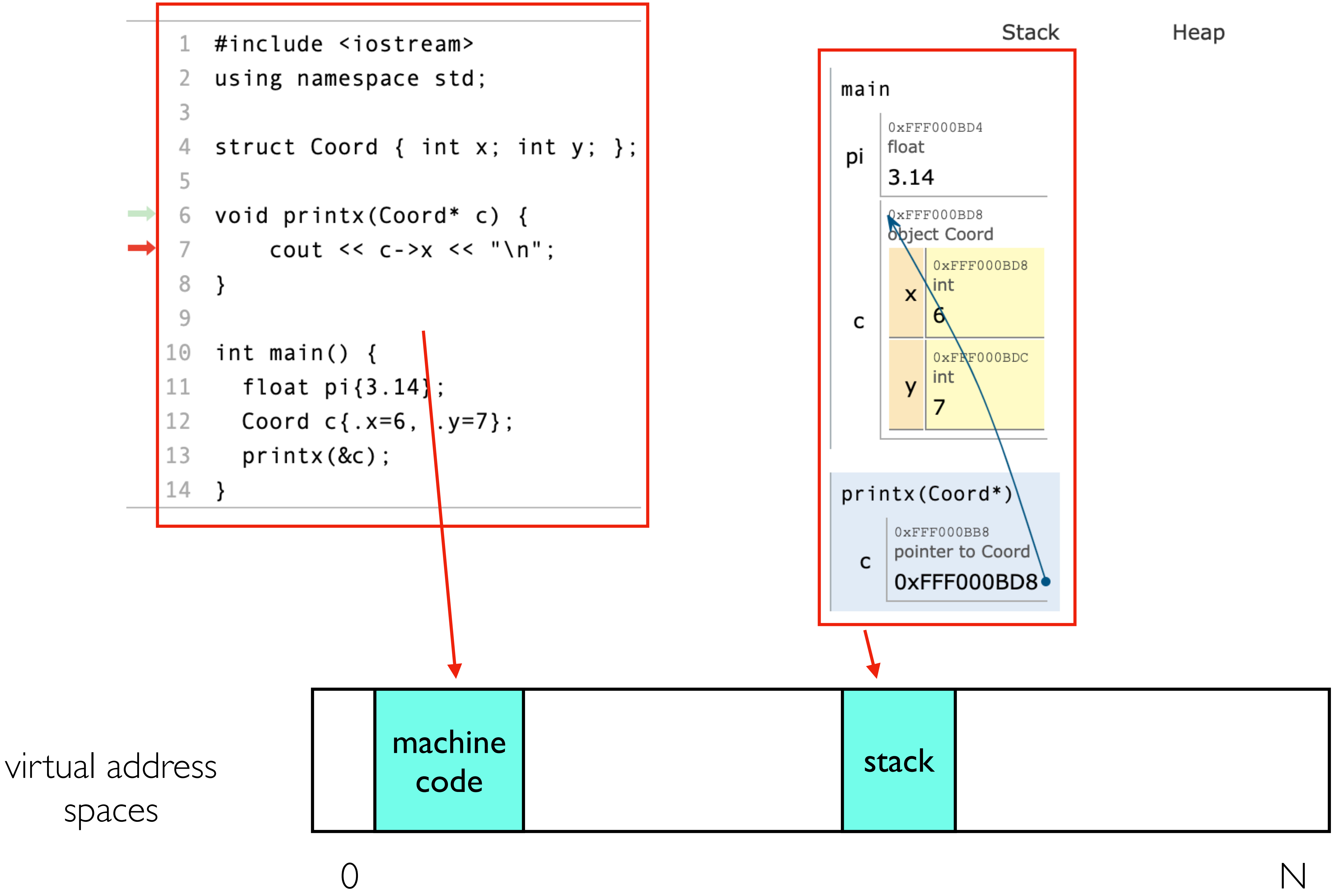


https://pythontutor.com/cpp.html#mode=edit

virtual address spaces



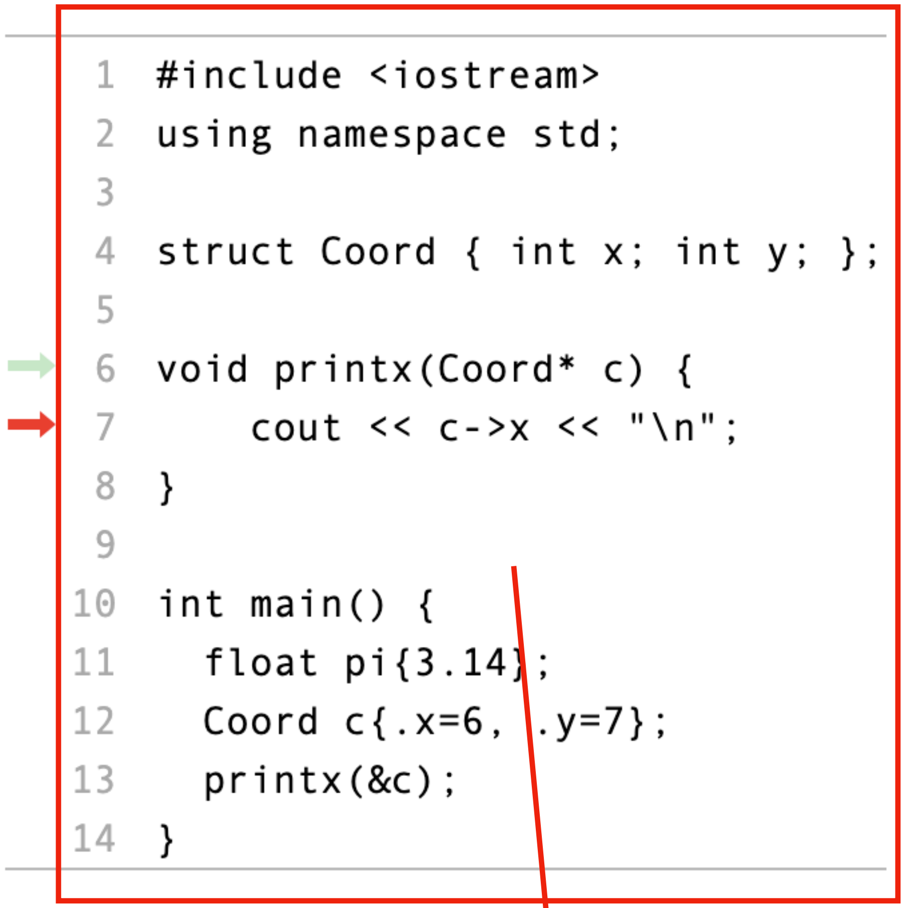
What goes in an address space?



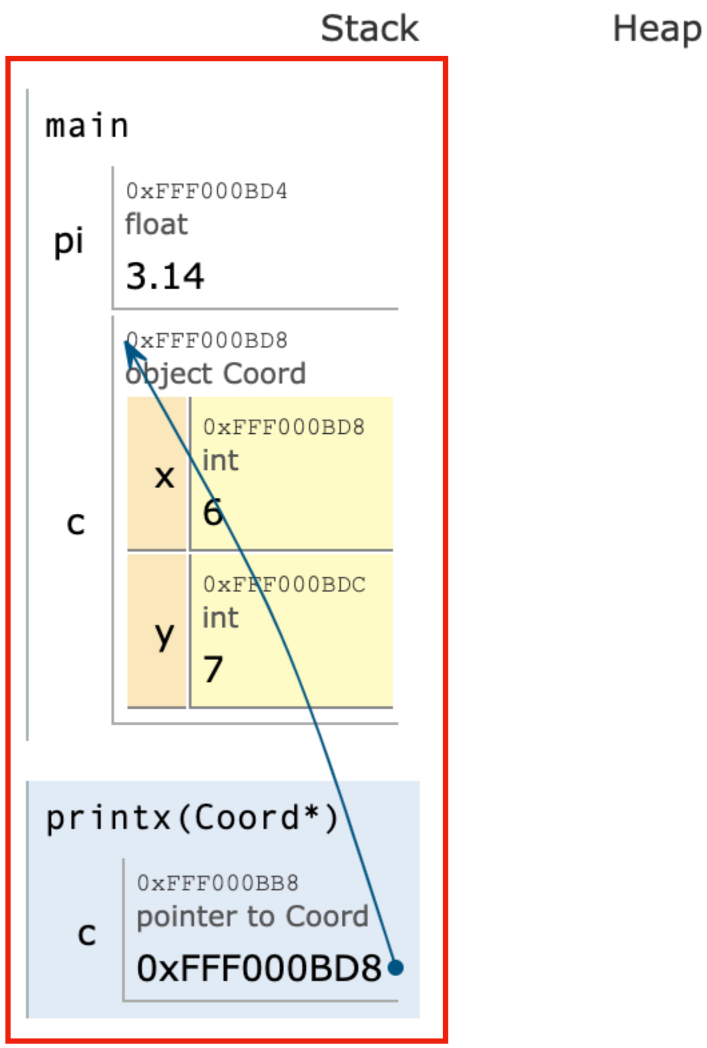
```
#include <iostream>
using namespace std;

struct Coord { int x; int y; };

void printx(Coord* c) {
    cout << c->x << "\n";
}

int main() {
  float pi{3.14};
  Coord c{.x=6, .y=7};
  printx(&c);
}
```

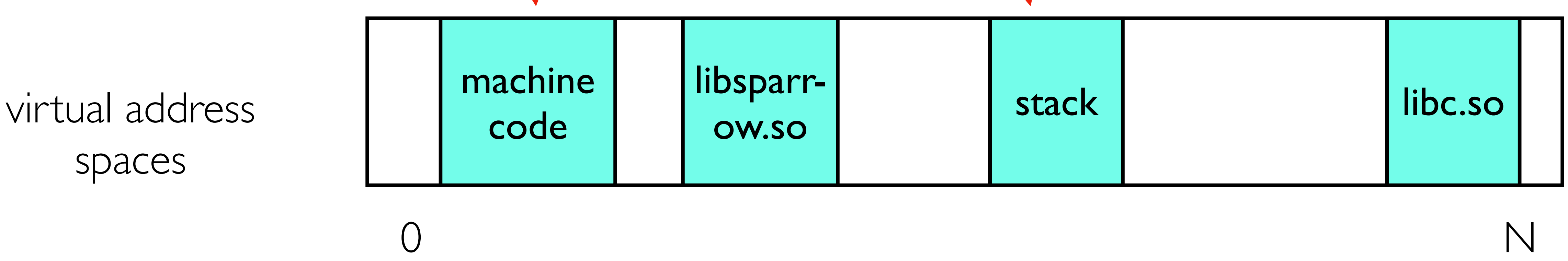
Note: a stack has parms, local vars, etc -- it's contiguous in mem

What goes in an address space?



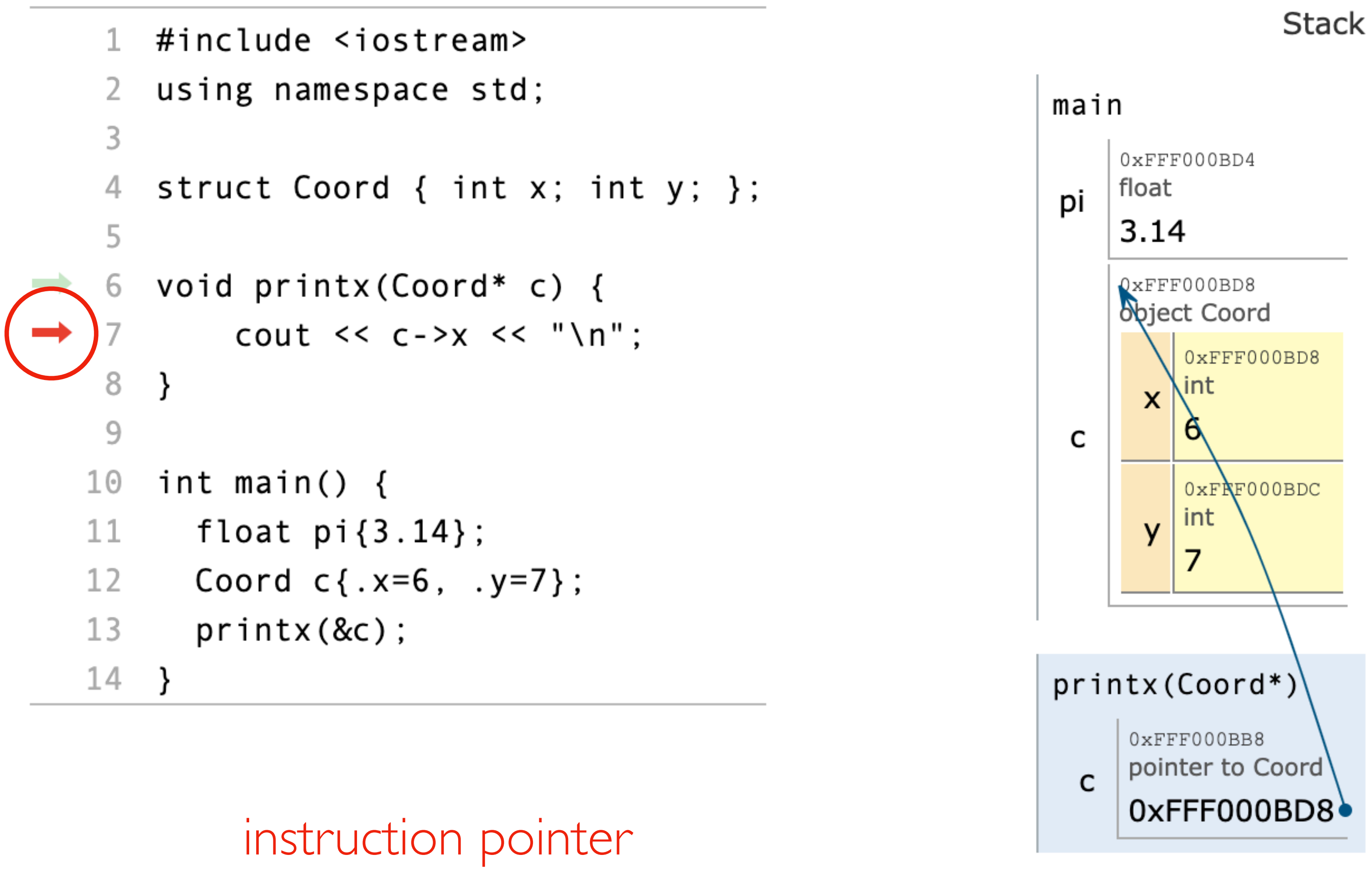
```
#include <iostream>
using namespace std;

struct Coord { int x; int y; };

void printx(Coord* c) {
    cout << c->x << "\n";
}

int main() {
  float pi{3.14};
  Coord c{.x=6, .y=7};
  printx(&c);
}
```





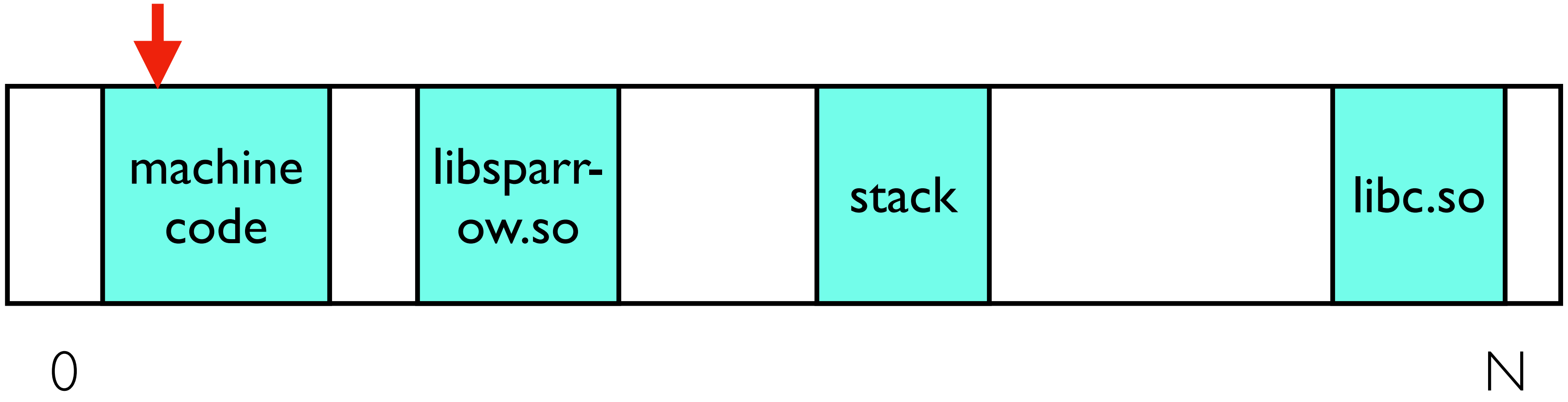
Note: file-backed mmaps load in other code (e.g., .so files)

How does code execute?



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

struct Coord { int x; int y; };

void printx(Coord* c) {
    cout << c->x << "\n";
}

int main() {
  float pi{3.14};
  Coord c{.x=6, .y=7};
  printx(&c);
}
```

instruction pointer

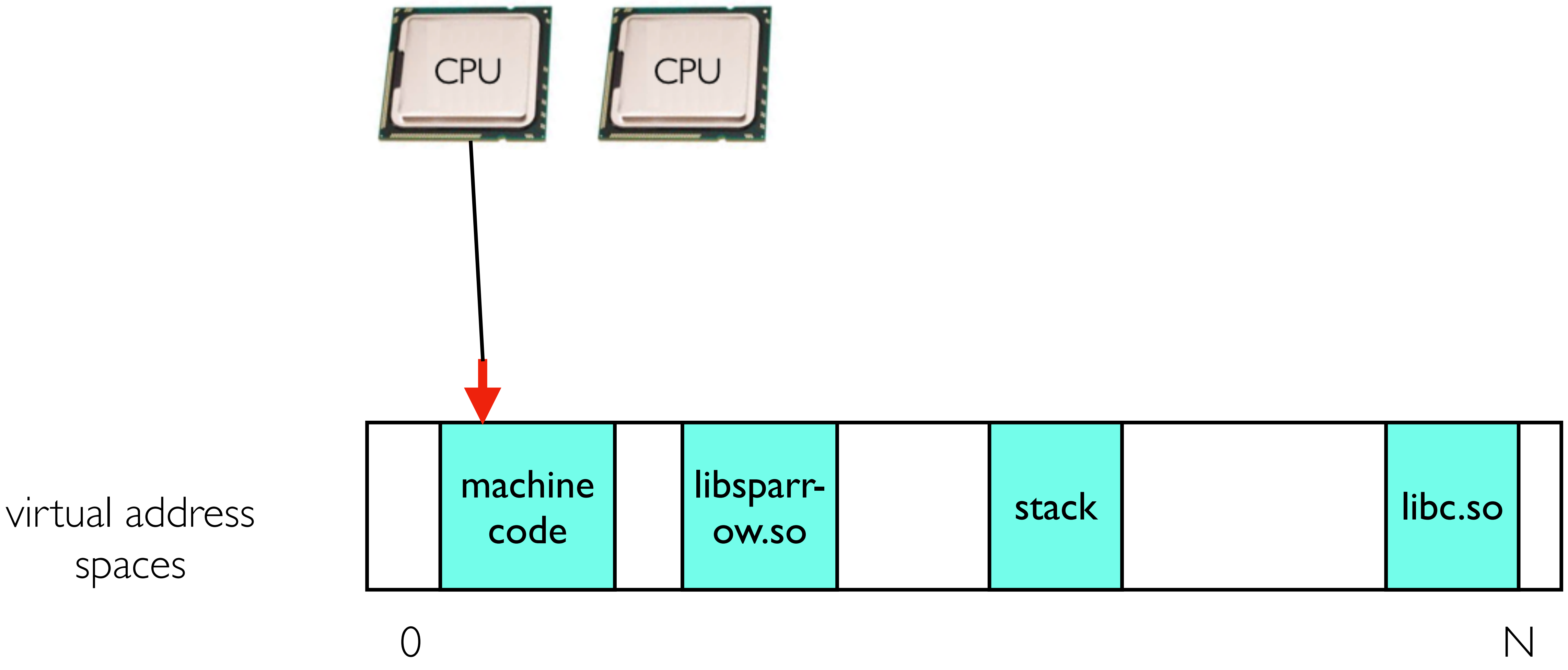


virtual address spaces

How does code execute?

CPUs

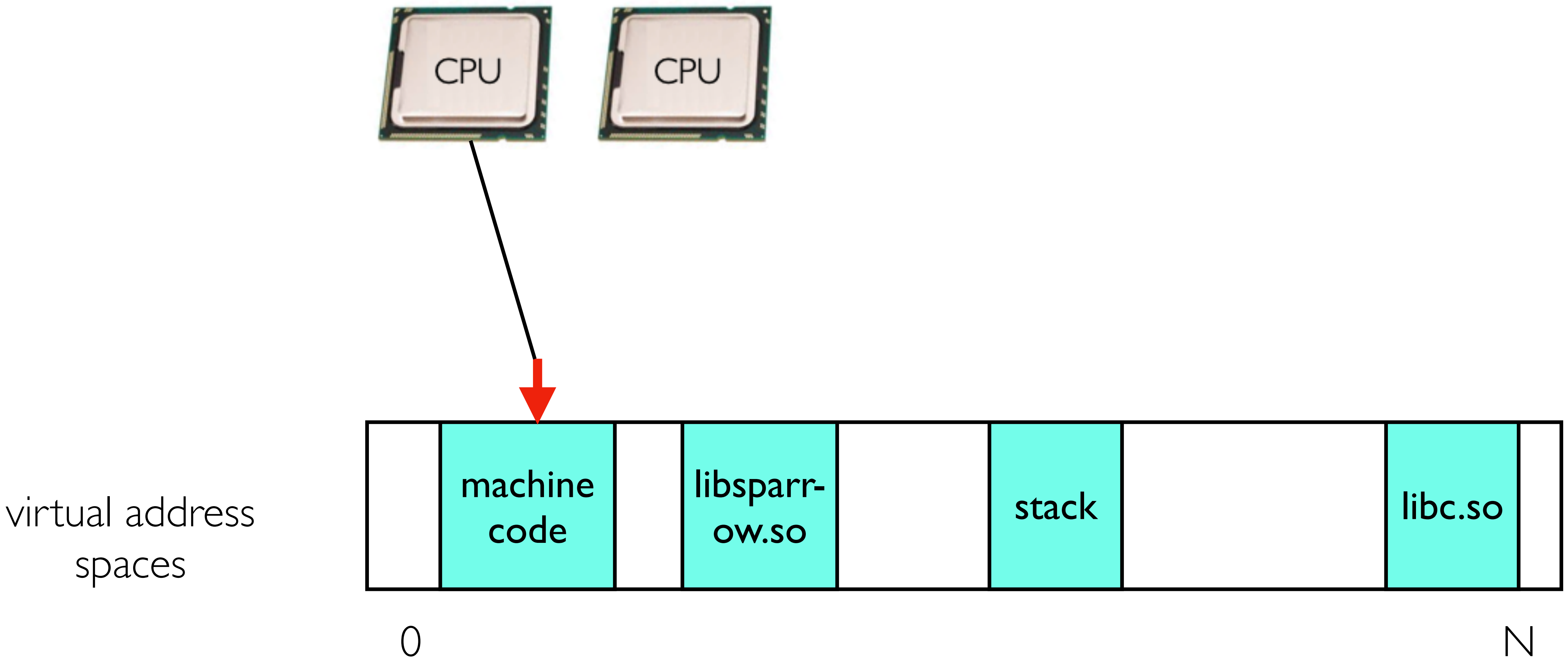
- CPUs are attached to at most one instruction pointer at any given time
- they run code by executing instructions and advancing the instruction pointer



How does code execute?

CPUs

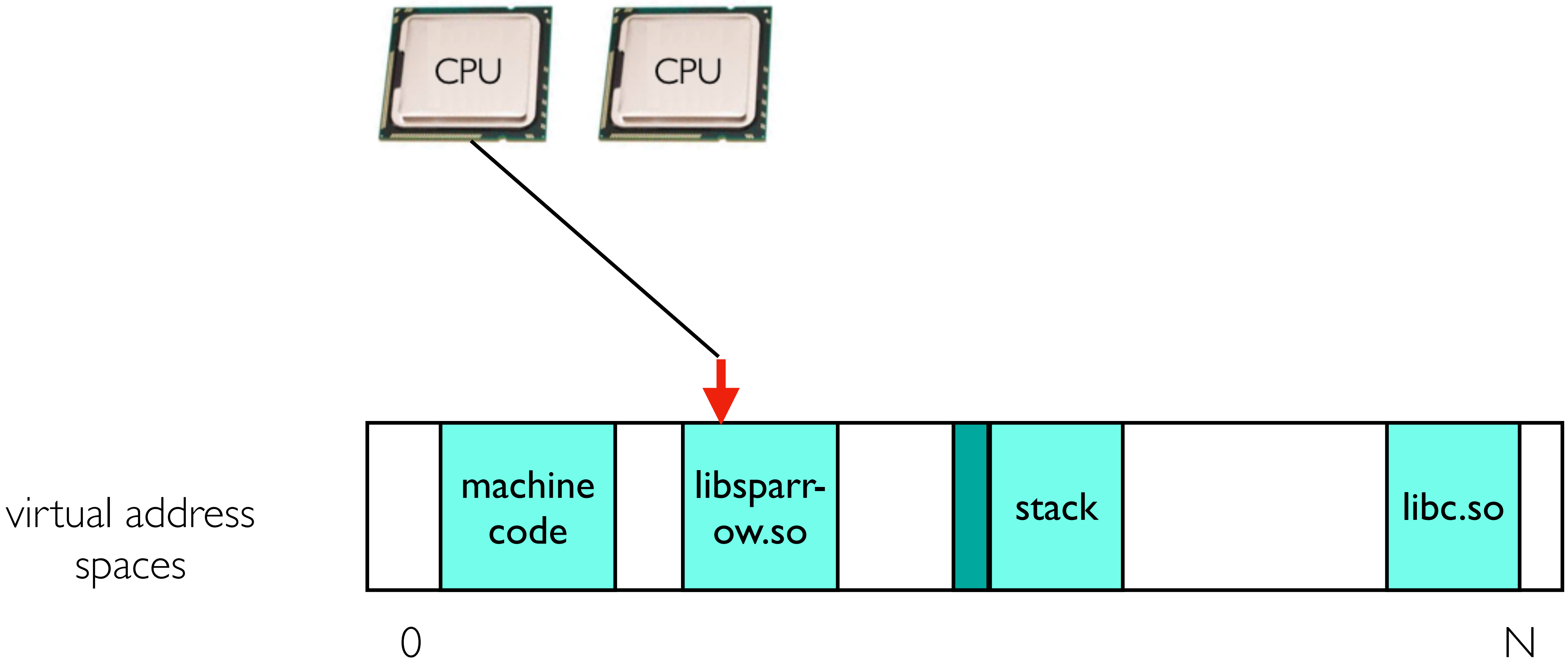
- CPUs are attached to at most one instruction pointer at any given time
- they run code by executing instructions and advancing the instruction pointer
- CPU moves instruction pointer as code executes



How does code execute?

CPUs

- function called: stack frame added to stack (for new vars, params)
- function returns: stack frame popped (to free up that memory)



CPU

CPU

virtual address spaces

machine code

libsparrow.so

stack

libc.so

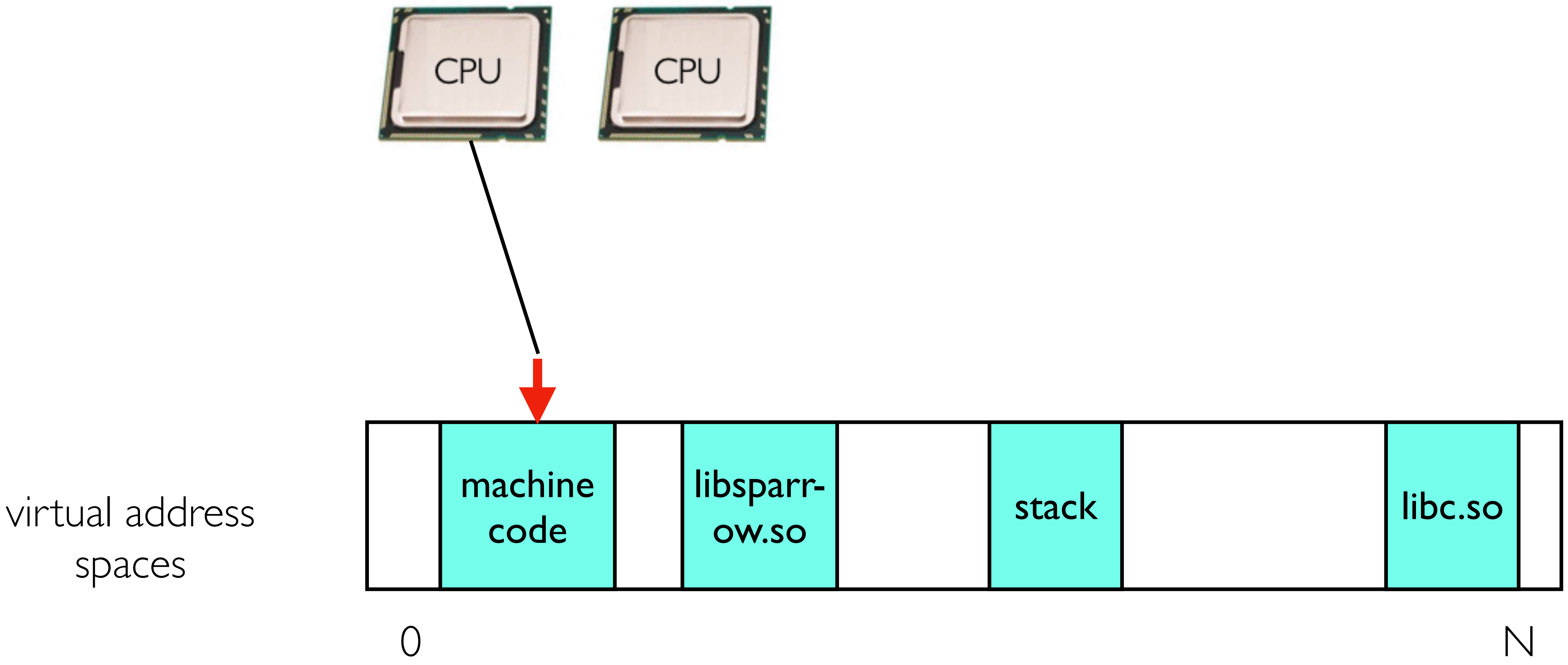
0

N

How does code execute?

CPUs

- function called: stack frame added to stack (for new vars, params)
- function returns: stack frame popped (to free up that memory)



machine code

libsparrow.so

stack

libc.so

virtual address spaces

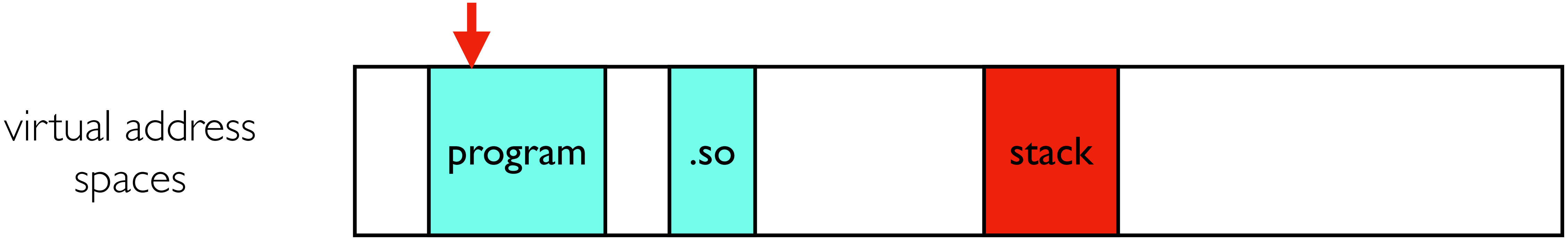
0

N

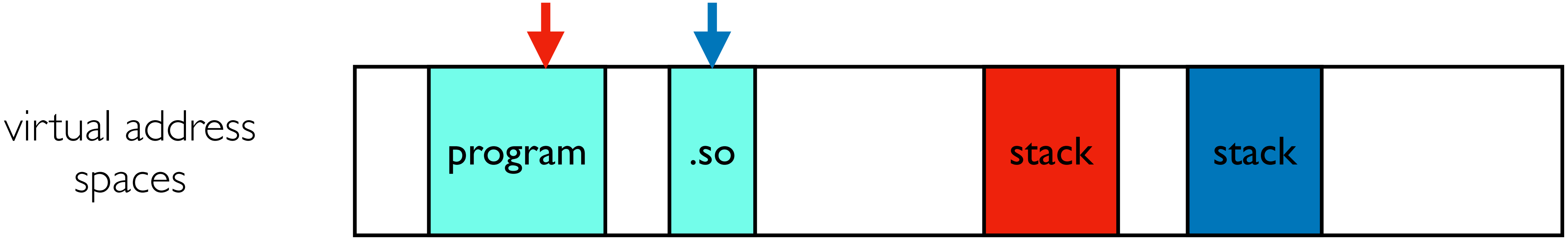
Threads

Threads have their own instruction pointers and stacks.
Multiple threads let us use multiple CPU cores at the same time!

Single-threaded process:



Multi-threaded process:



Stack: Benefits and Limitations

Benefit 1: cleanup happens automatically when function returns!

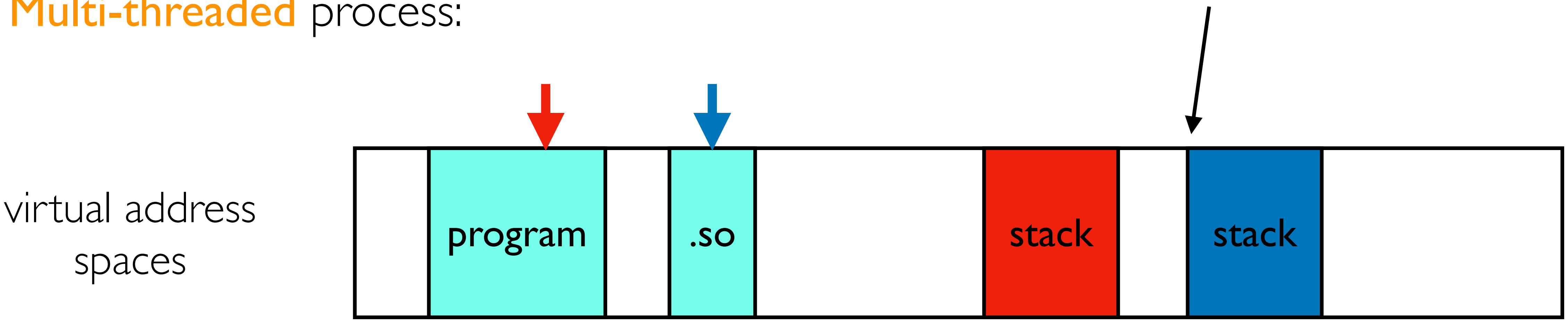
Benefit 2: allocating/dealocating stack memory is FAST.

Limitation 1: what if we want data shared across threads?

Limitation 2: what if we want the data to stay around after function returns?

Limitation 2:
limited room
to grow

Multi-threaded process:

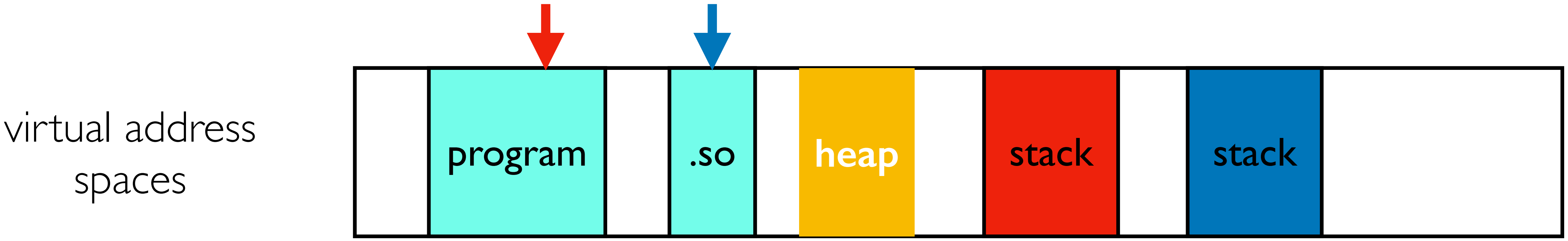


Heap

Characteritics

- explicitly control memory lifetime with new/delete
- shared across threads
- non-contiguous, can use more memory

Multi-threaded process:



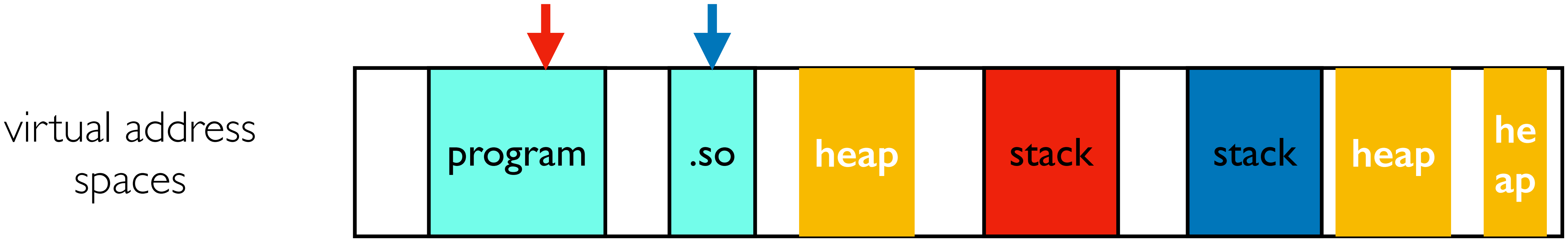
Note: anonymous mmaps grab pages of memory from operating system

Heap

Characteritics

- explicitly control memory lifetime with new/delete
- shared across threads
- non-contiguous, can use more memory

Multi-threaded process:



Note: anonymous mmaps grab pages of memory from operating system

Outline

TopHat and Worksheet

Memory Layout: Code/Stack/Heap

new/delete

arrays

Worksheet

Safety

Motivation: Stack (with bug)

```
int* mult2(int x) {
   int y = x * 2;
   return &y;
}

int main() {
   int* result = mult2(3);
   cout << *result << "\n";
}
```

as soon as mult2 returns,
memory for y is no longer valid

result points to invalid memory

if we want memory to stay valid
after function return, we should
use the heap, not the stack

Heap (with leak bug)

```
int* mult2(int x) {
  int* y = new int{x*2};
  return y;
}

int main() {
  {
    int* result = mult2(3);
    cout << *result << "\n";
  }

  ...
}
```

new: y will point to 4 bytes of heap memory

result points to valid memory!

but it will never be released...

Heap (with double free bug)

```
int* mult2(int x) {
  int* y = new int{x*2};
  return y;
}

int main() {
  {
    int* result = mult2(3);
    cout << *result << "\n";
    delete result;
    delete result;
  }
  ...
}
```

```
malloc: *** error for object 0x600000ce8040:
        pointer being freed was not allocated
```

Heap (with dangling pointer bug)

```
int* mult2(int x) {
  int* y = new int{x*2};
  return y;
}

int main() {
  {
    int* result = mult2(3);
    delete result;
    cout << *result << "\n";  // -1243955136, or some
  }                           // other garbage value
  ...
}
```

The Heap is Tricky

Still need to worry about memory corruptions and segfaults.

"Exciting" new kinds of memory bugs too!

- leaks
- double frees
- dangling pointers

Every "new" call needs a corresponding "delete" call, at the right time: after no more pointers will be followed to that memory. Ideally as soon as possible after that!

Getting this right is hard and complex! Common to have extra data for bookkeeping (for example, an int that keeps track of how many active pointers reference a variable).

Big C++ advantage over C: references and smart pointers (which we'll learn soon!) help us avoid many common mistakes.

Memory API Comparison

	allocate	deallocate	notes
C++	`new`	`delete`	built on malloc, but returns specific type (no need to cast) and does init/cleanup (constructor/destructor) in addition to basic memory work
C	`malloc`	`free`	any granularity (e.g., 8-byte double); uses mmap
UNIX	`mmap`	`munmap`	page granularity (4 KB)

malloc/free should never appear
in your code this semester!

Outline

TopHat and Worksheet

Memory Layout: Code/Stack/Heap

new/delete

arrays

Worksheet

Safety

Arrays

What if we want more than one value of the same type, contiguous in memory?

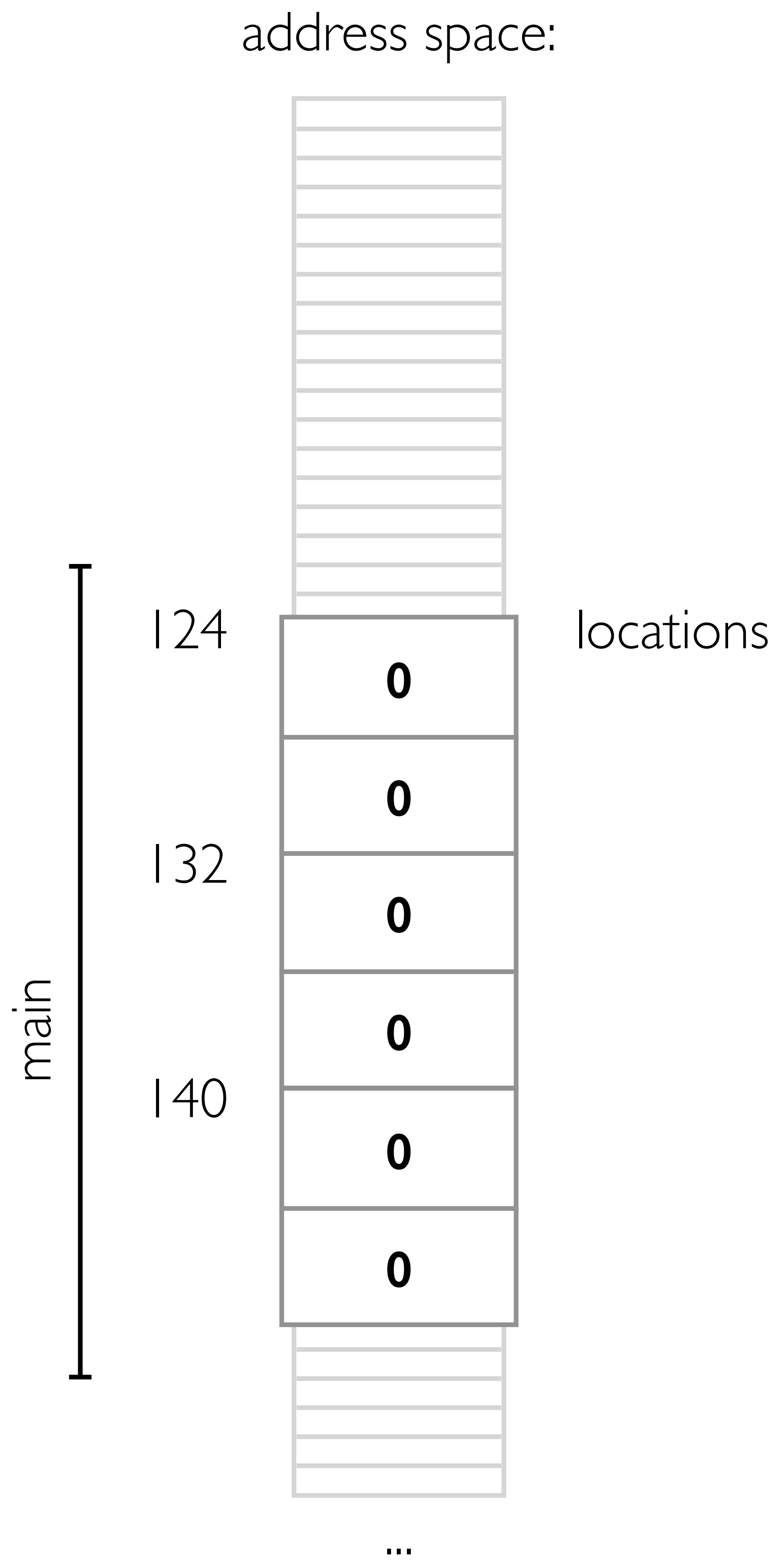
We can use arrays! Arrays are very minimalist. Cannot resize. Number of elements often not even stored anywhere (need separate variable).

Advice

- consider using C++ arrays when building your own data structures
- use vector, STL arrays, or other structs in most case

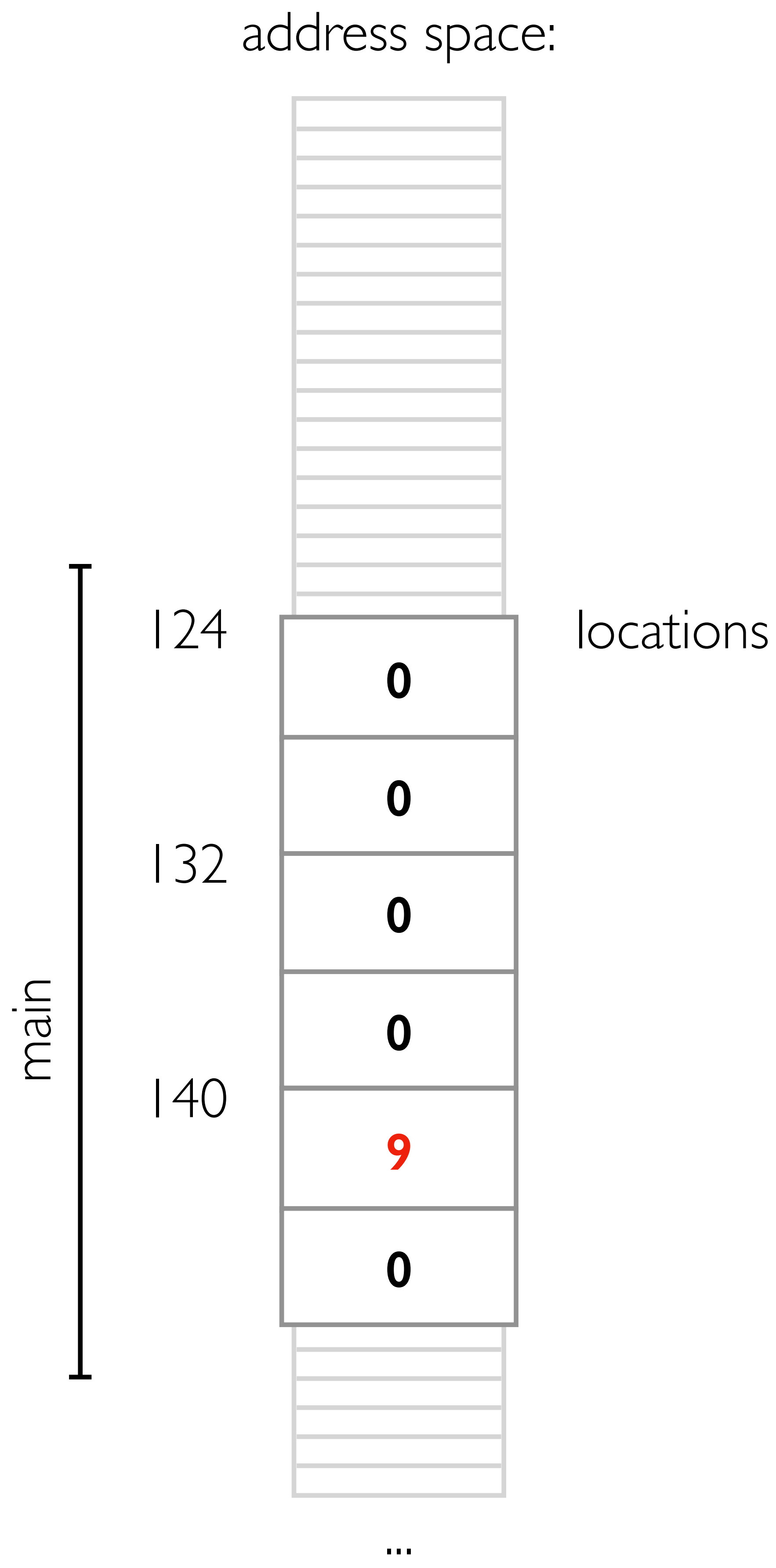
Arrays on the Stack

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

int main() {
  Loc locations[3];
}
```



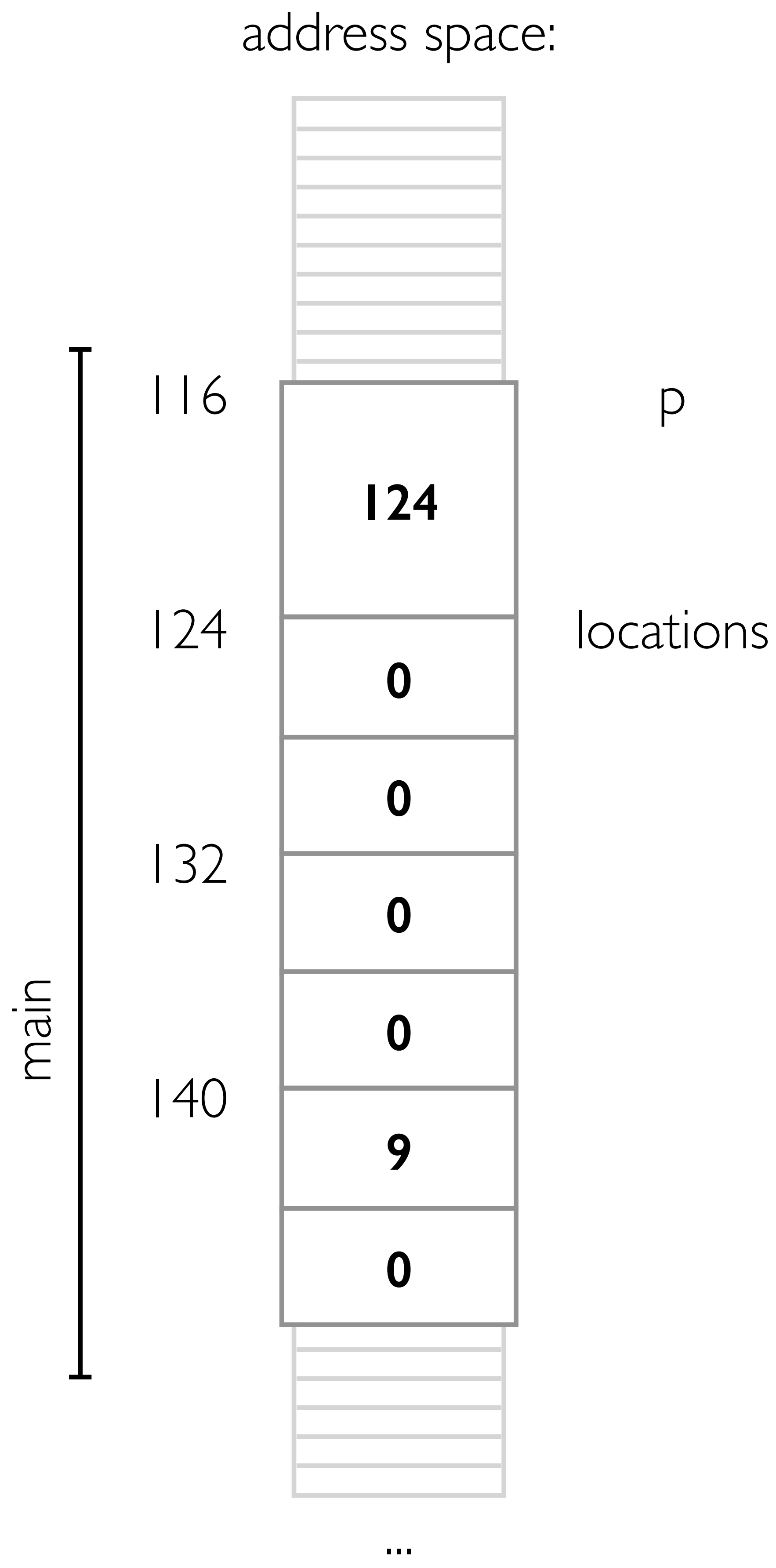
Arrays on the Stack

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

int main() {
  Loc locations[3];
  locations[2].x = 9;
}
```



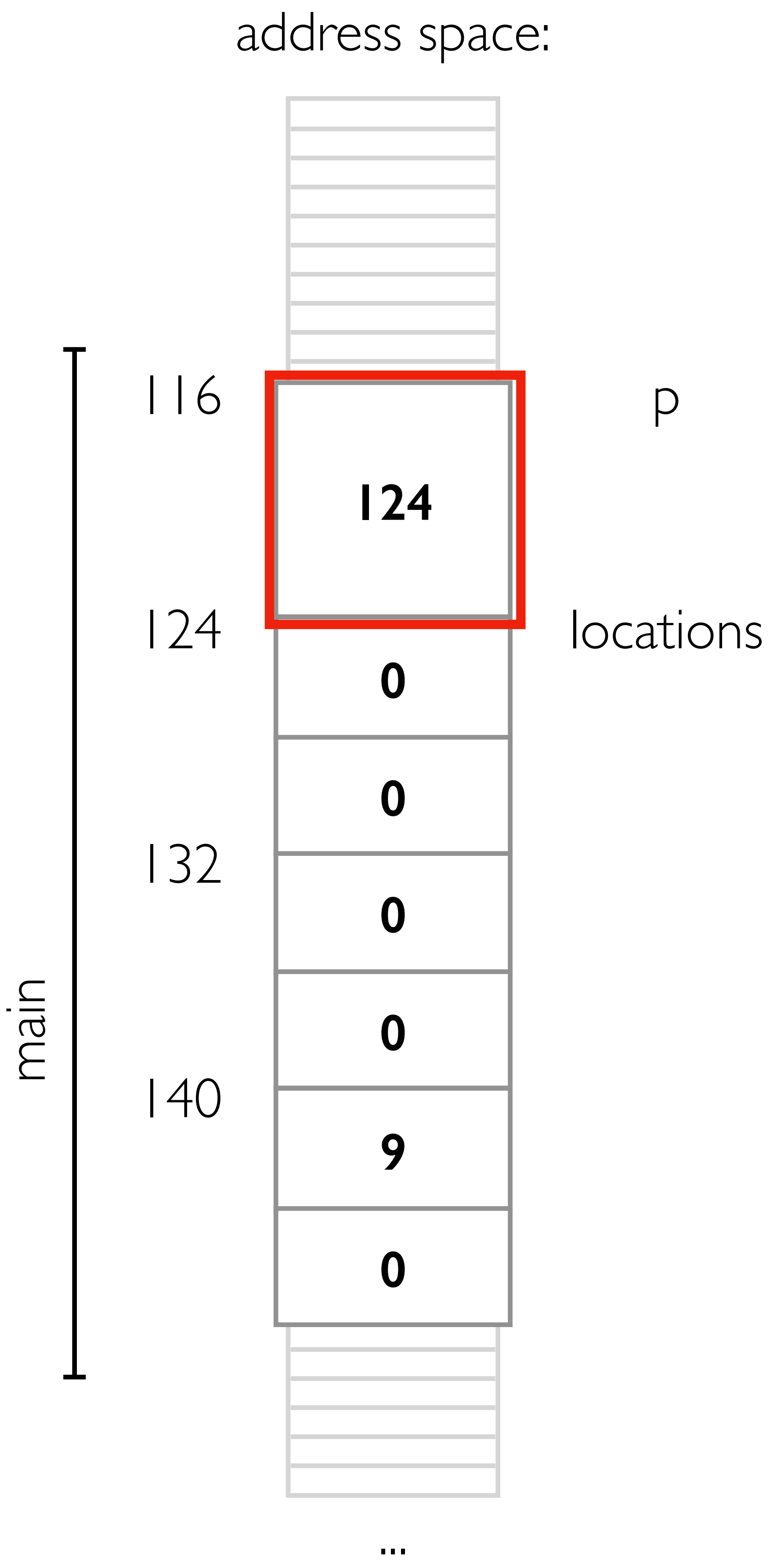
Arrays "Decay" to Pointers

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

int main() {
  Loc locations[3];
  locations[2].x = 9;
  Loc *p = locations;
}
```



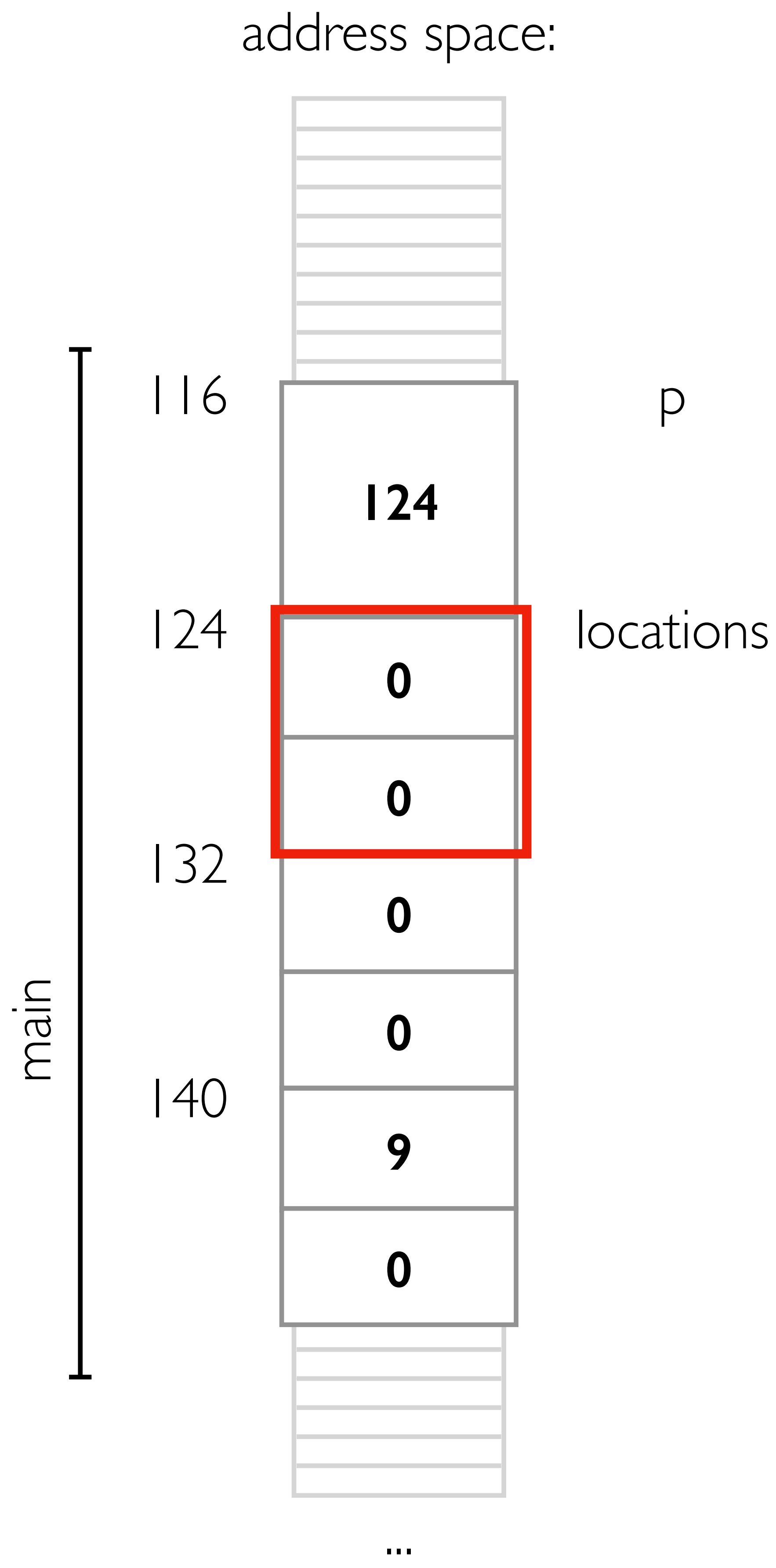
Arrays "Decay" to Pointers

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

int main() {
  Loc locations[3];
  locations[2].x = 9;
  Loc *p = locations;
  p
}
```



Arrays "Decay" to Pointers

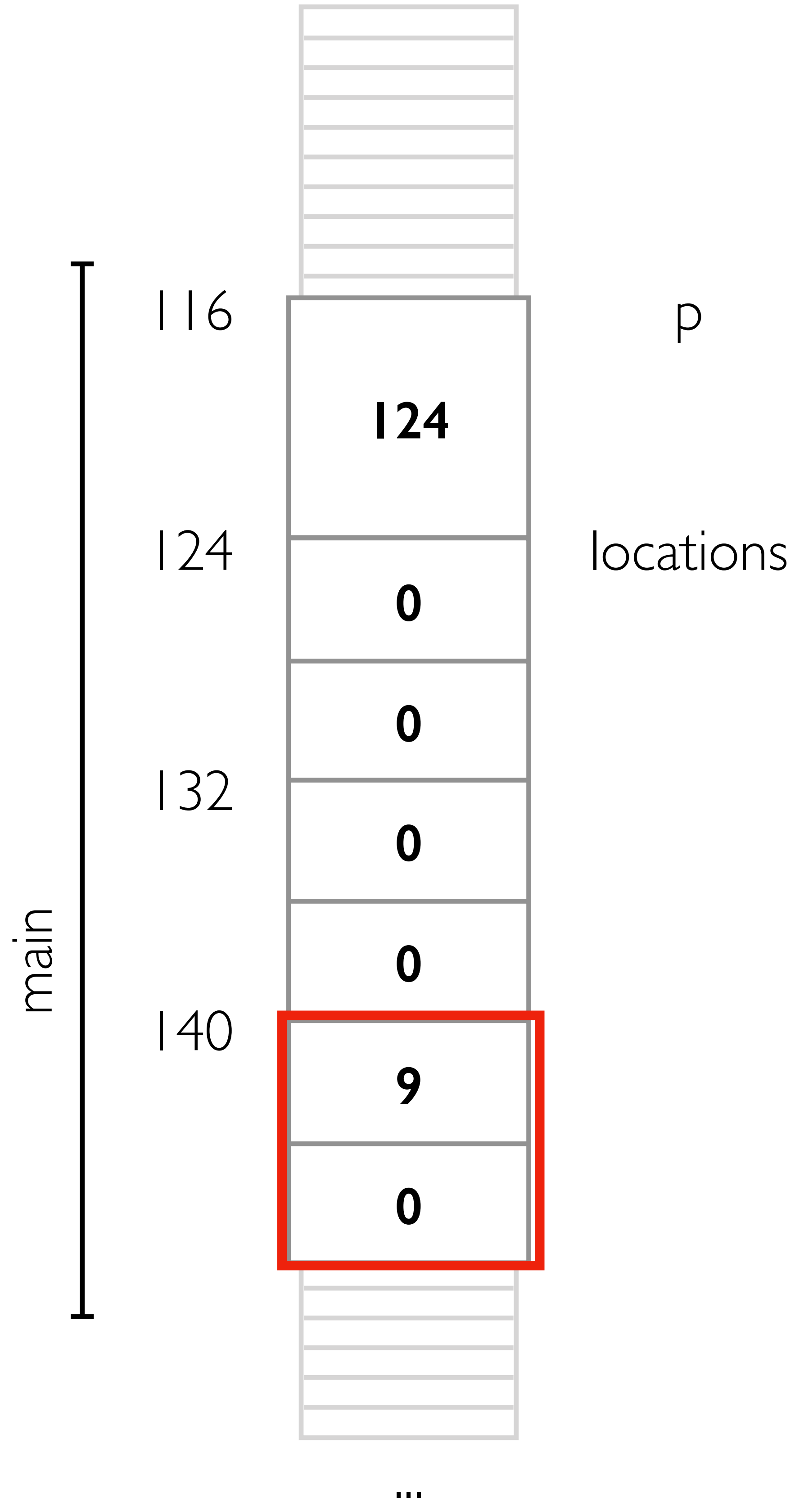
```
struct Loc {
  int x = 0;
  int y = 0;
};

int main() {
  Loc locations[3];
  locations[2].x = 9;
  Loc *p = locations;
  *p
}
```



Arrays "Decay" to Pointers

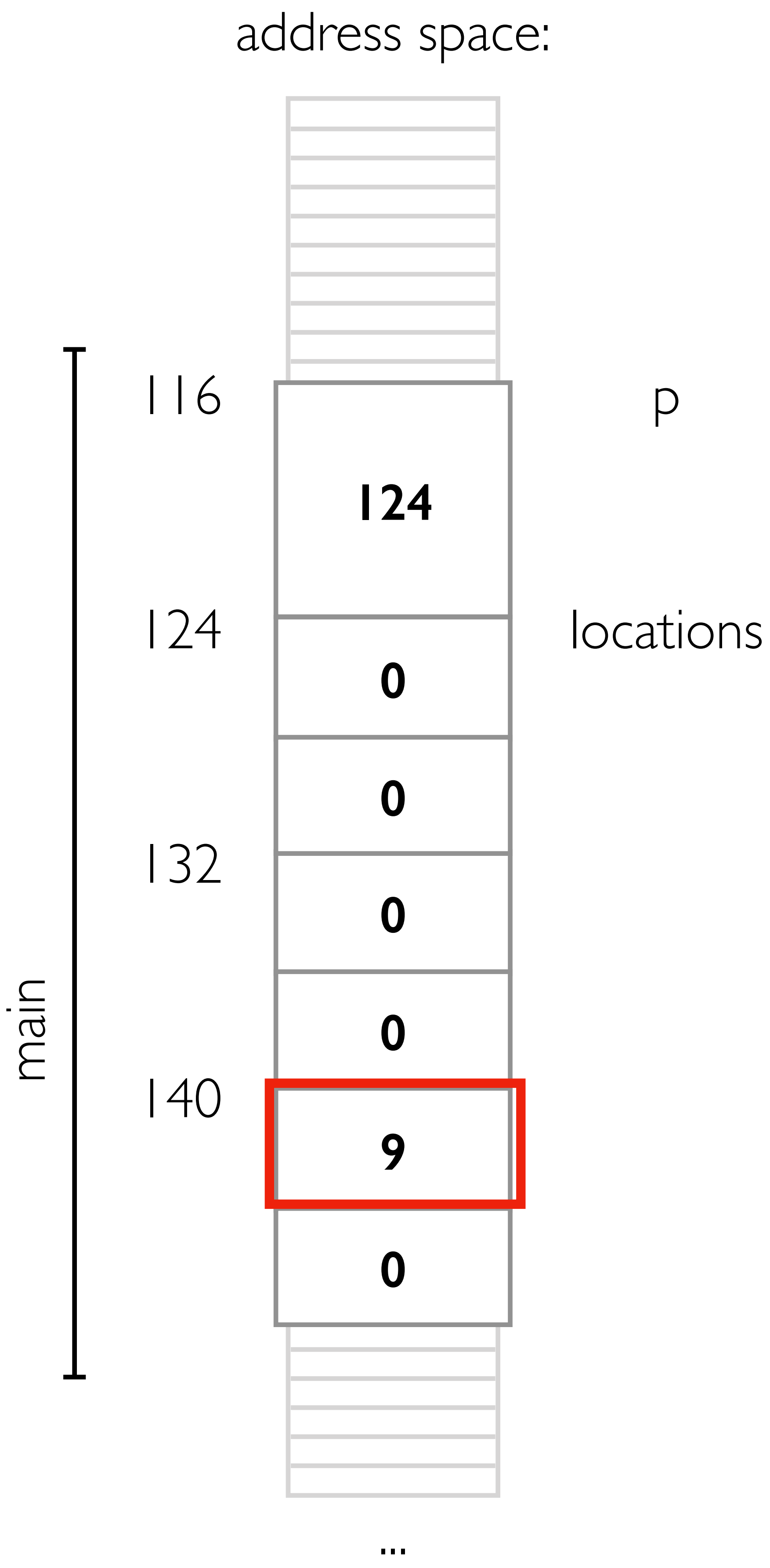
```
struct Loc {
  int x = 0;
  int y = 0;
};

int main() {
  Loc locations[3];
  locations[2].x = 9;
  Loc *p = locations;
  p[2]
}
```

address space:



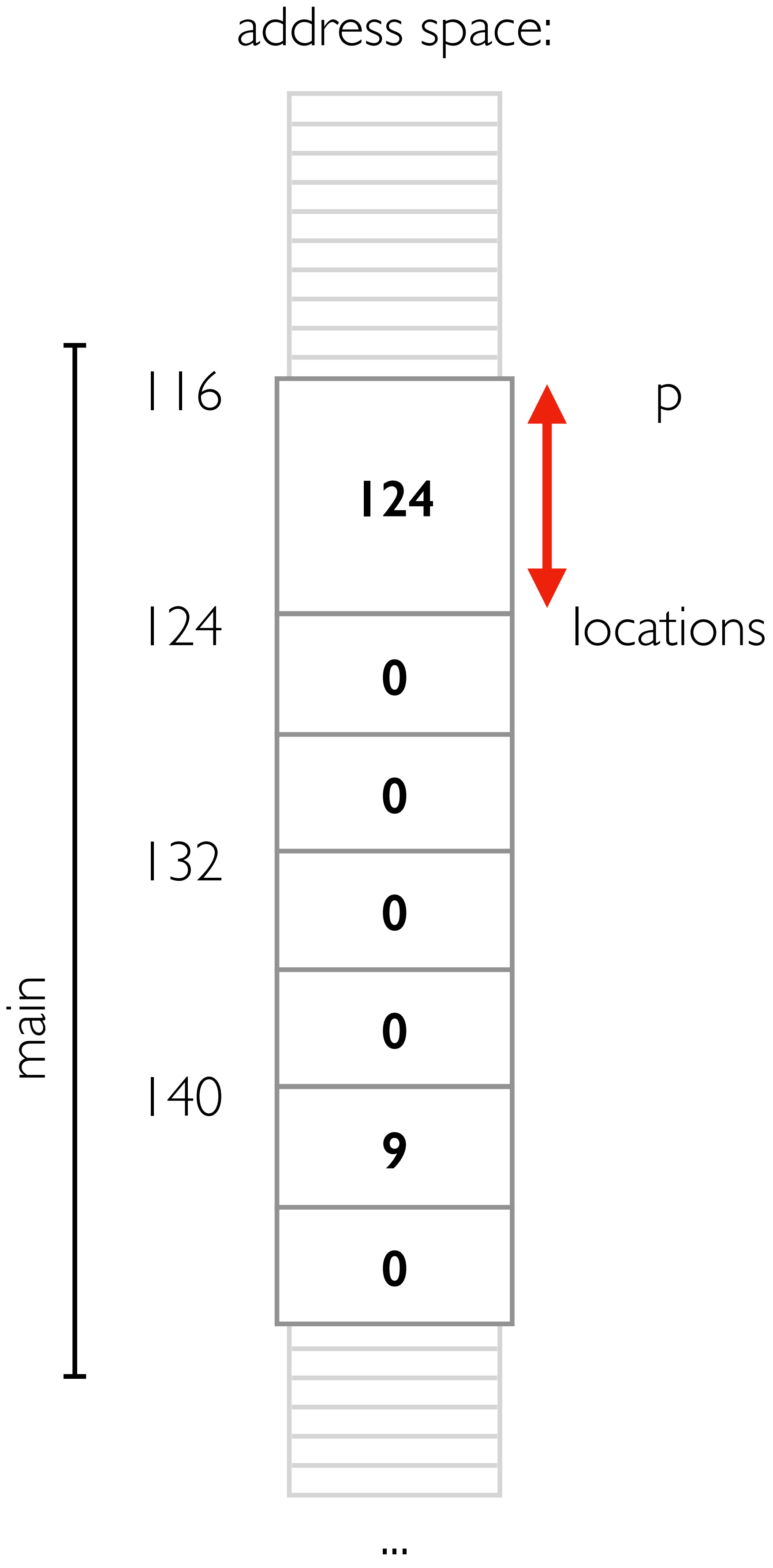
Arrays "Decay" to Pointers

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

int main() {
  Loc locations[3];
  locations[2].x = 9;
  Loc *p = locations;
  p[2].x
}
```



sizeof behavior

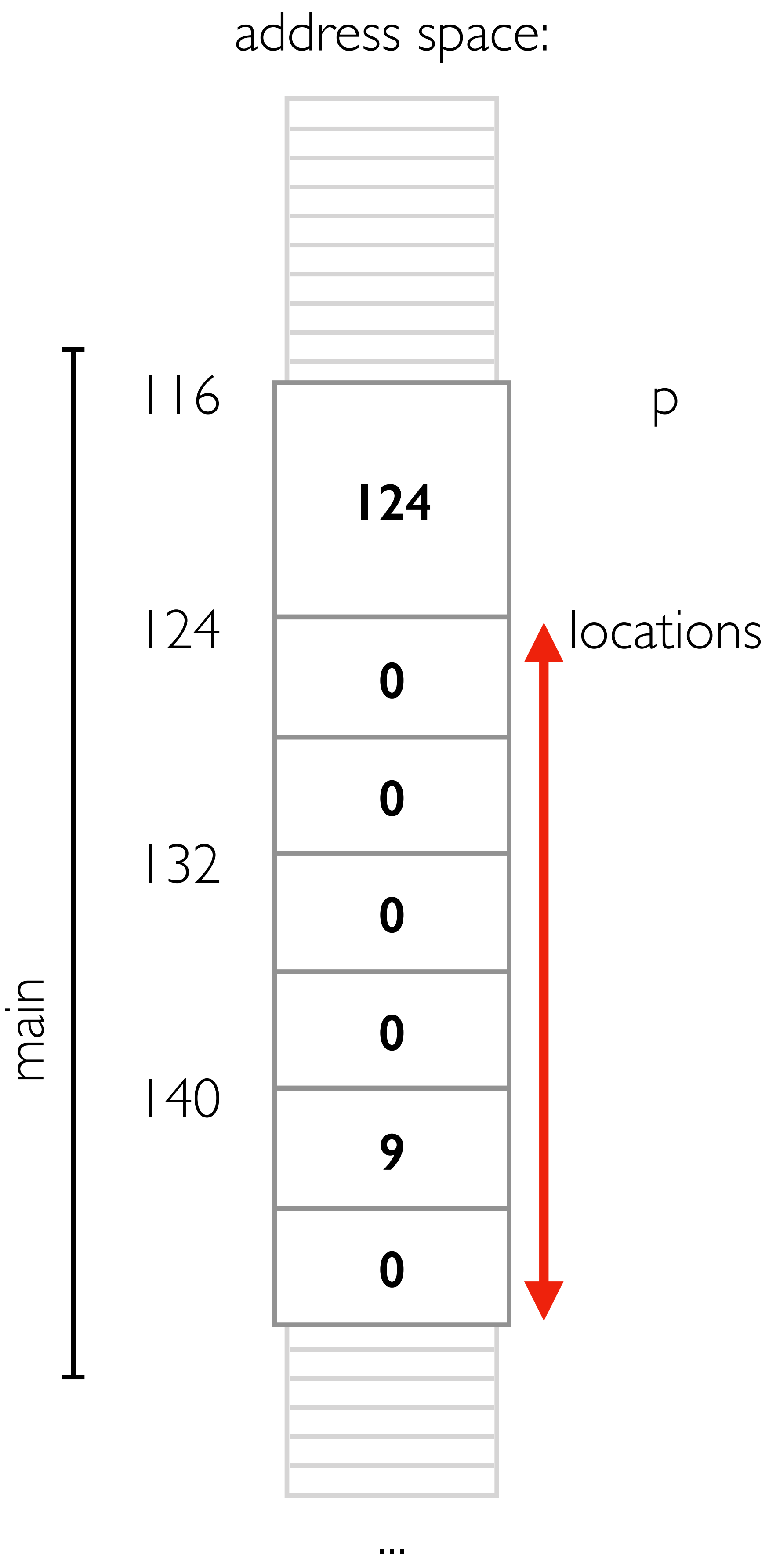
```
struct Loc {
   int x = 0;
   int y = 0;
};

int main() {
   Loc locations[3];
   locations[2].x = 9;
   Loc *p = locations;
   sizeof(p)     8 bytes
}
```



sizeof behavior

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

int main() {
   Loc locations[3];
   locations[2].x = 9;
   Loc *p = locations;
   sizeof(locations)
}
```

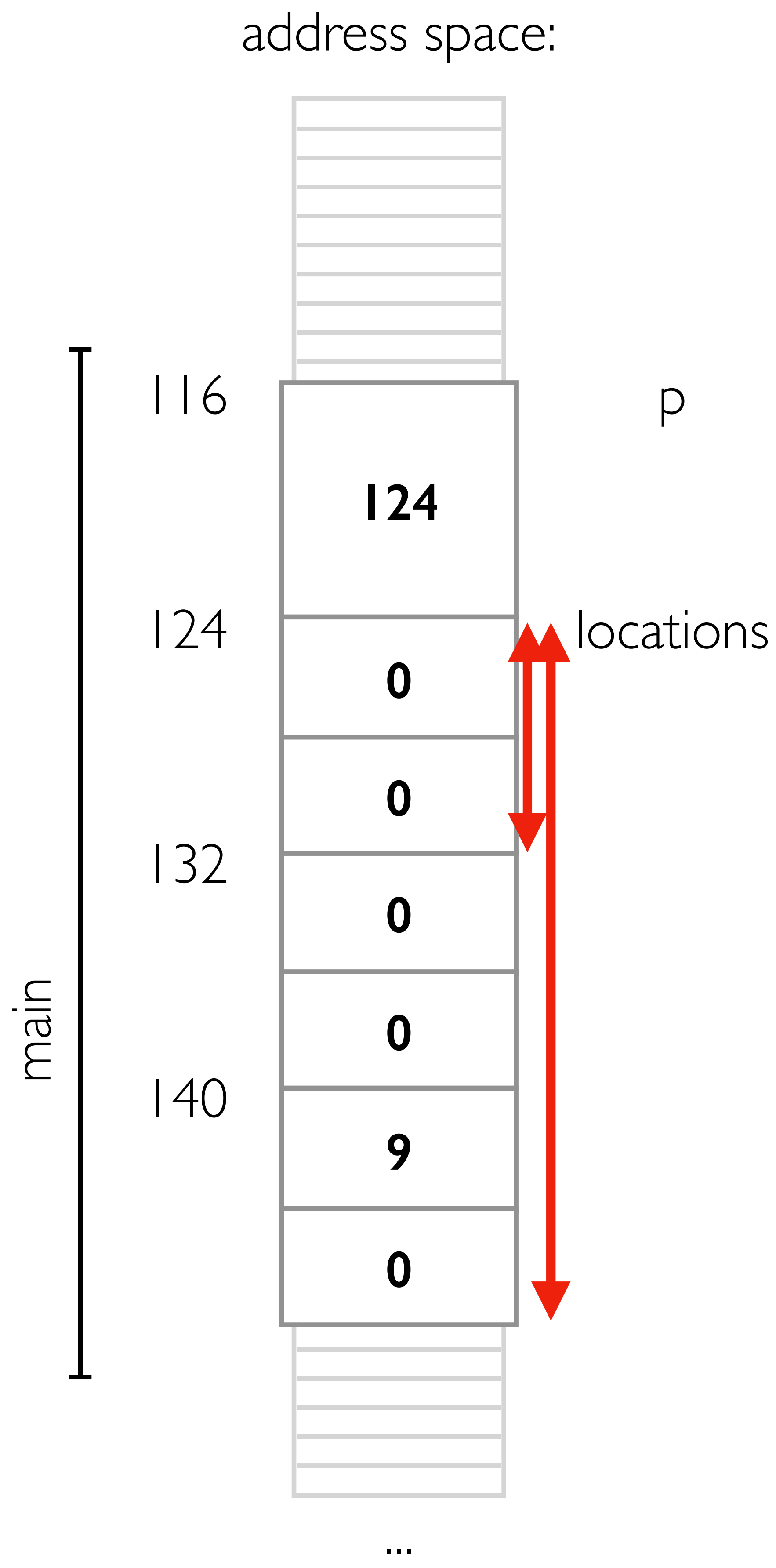
24 bytes



sizeof behavior

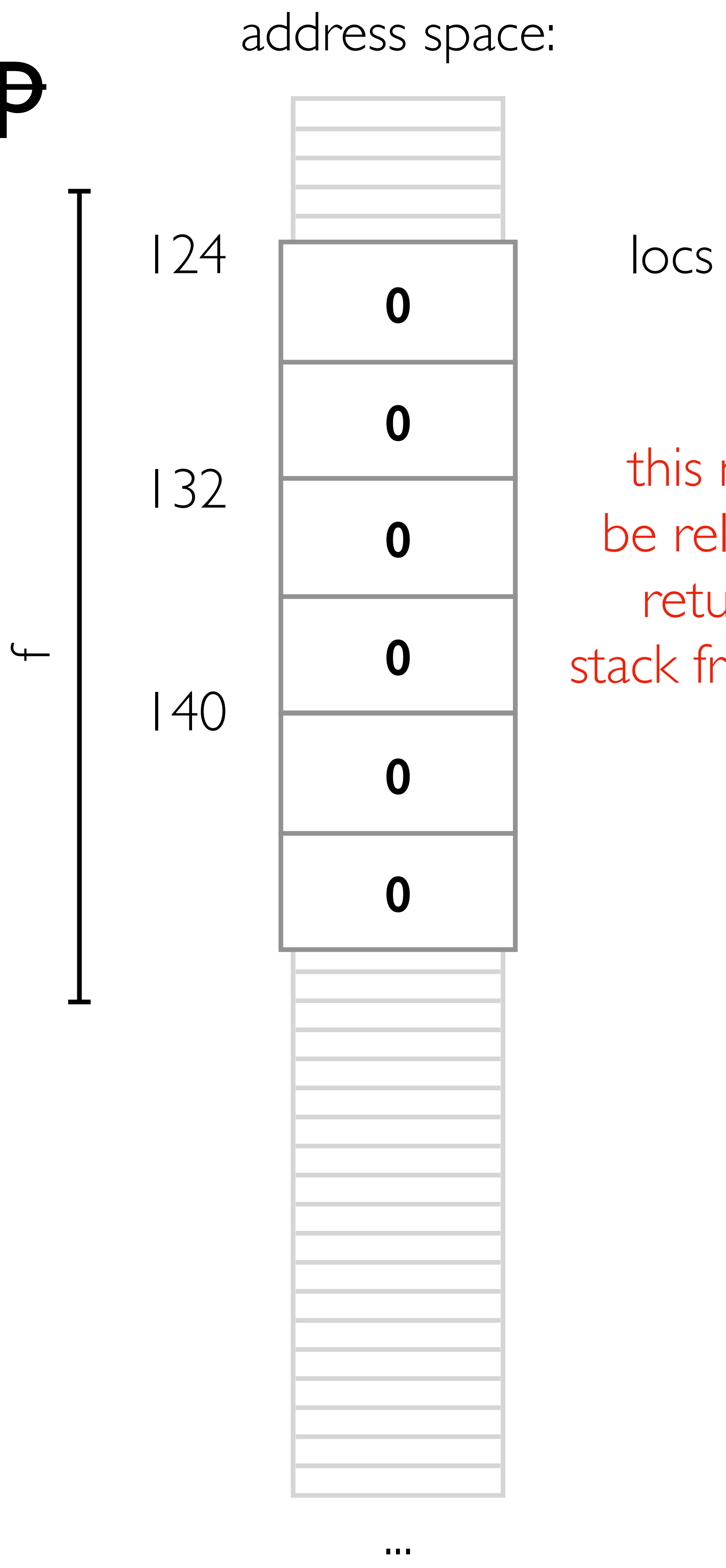
```
struct Loc {
   int x = 0;
   int y = 0;
};

int main() {
   Loc locations[3];
   locations[2].x = 9;
   Loc *p = locations;
   (sizeof(locations) /
   sizeof(locations[0])
}
```

3 elements



Arrays on the Stack ~~Heap~~

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

int f() {
  Loc locs[3];
}
```

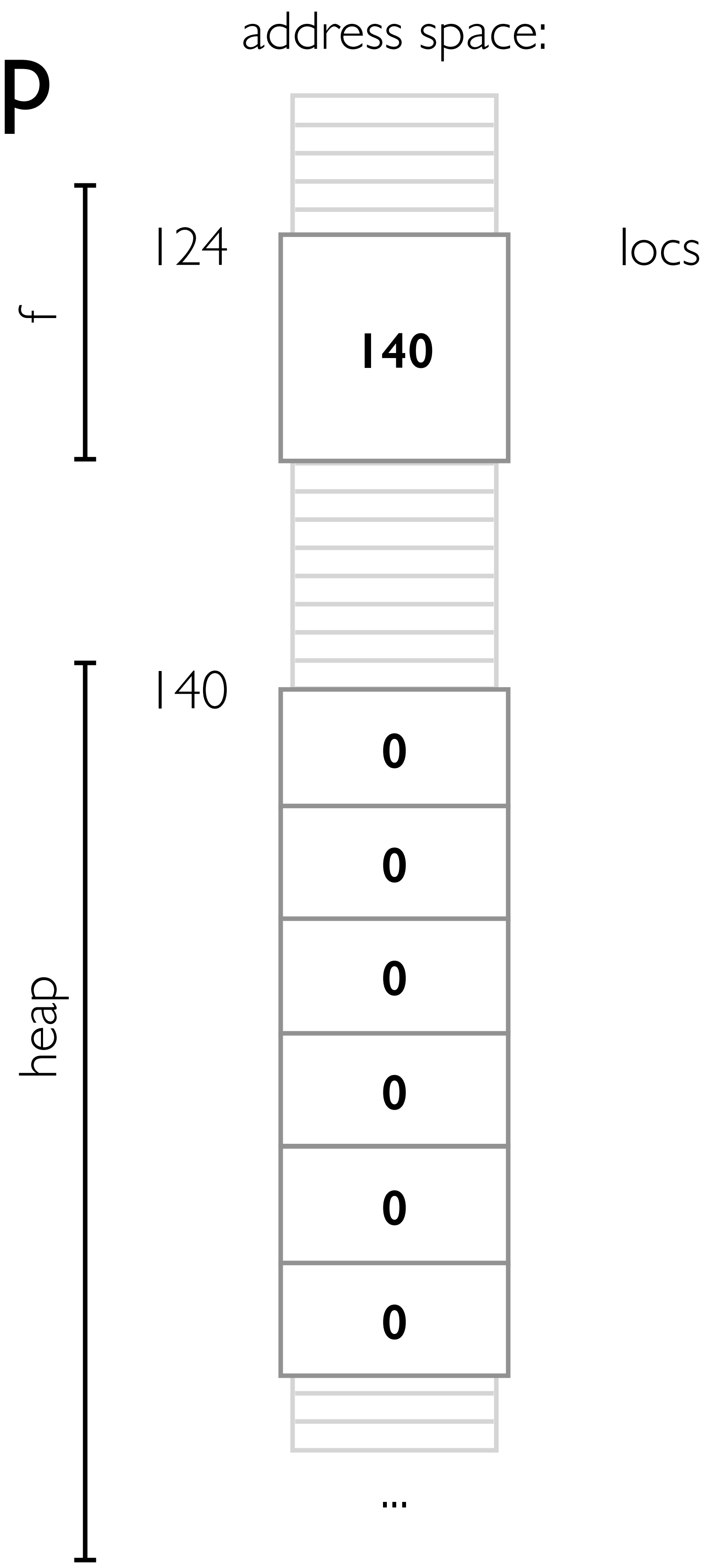


this memory will be released when f returns and the stack frame is popped

Arrays on the ~~Stack~~ Heap

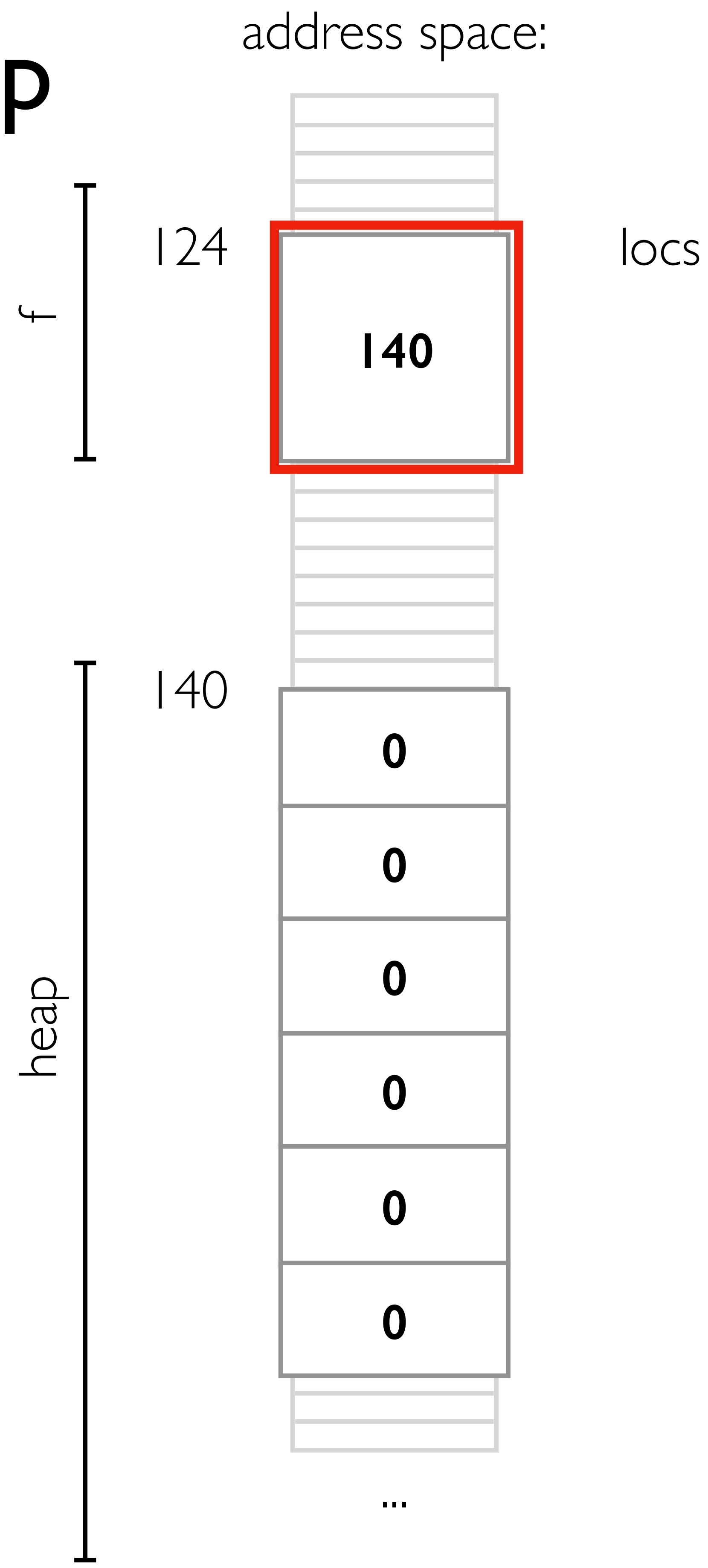
```
struct Loc {
   int x = 0;
   int y = 0;
};

int f() {
   Loc* locs = new Loc[3];
}
```

new can be used in combination with brackets



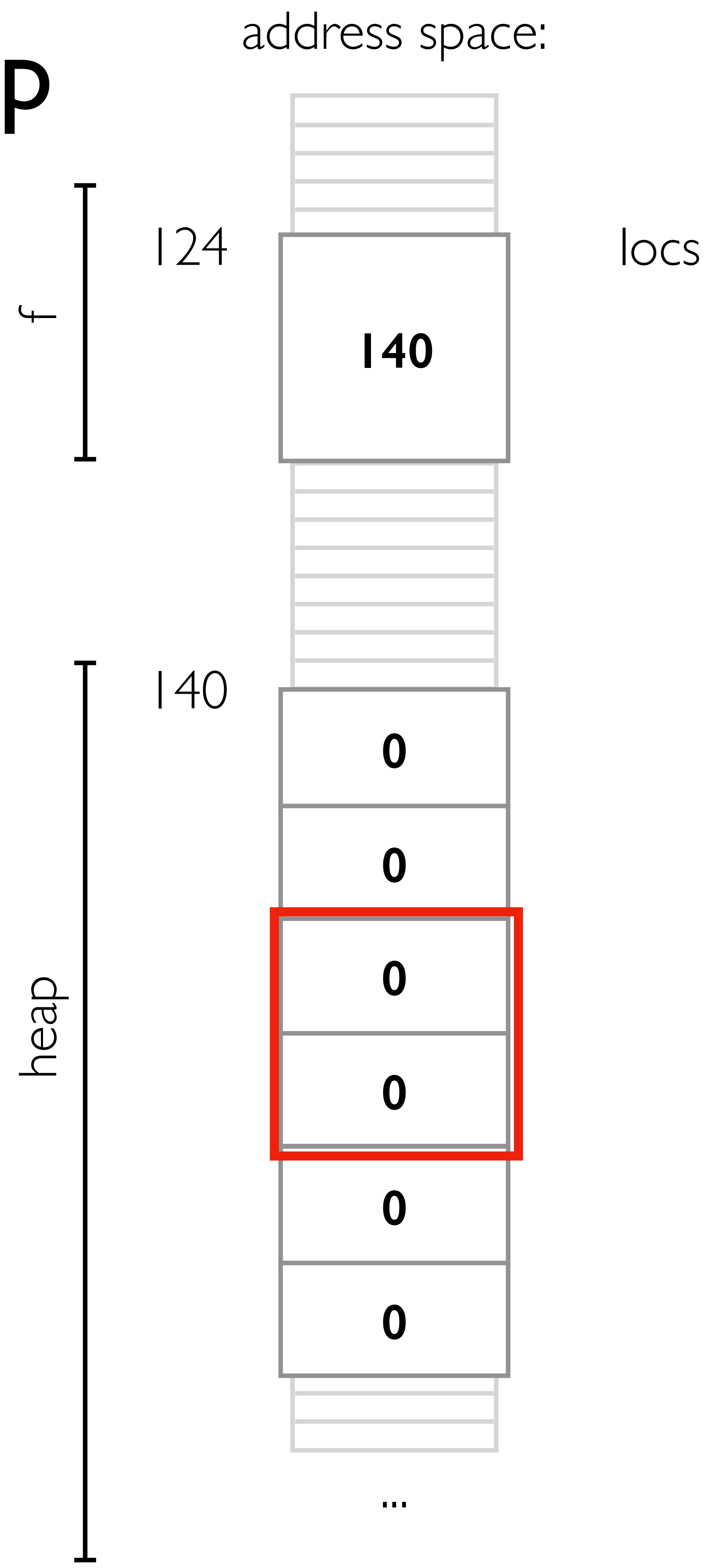
Arrays on the ~~Stack~~ Heap

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

int f() {
  Loc* locs = new Loc[3];
  locs
}
```



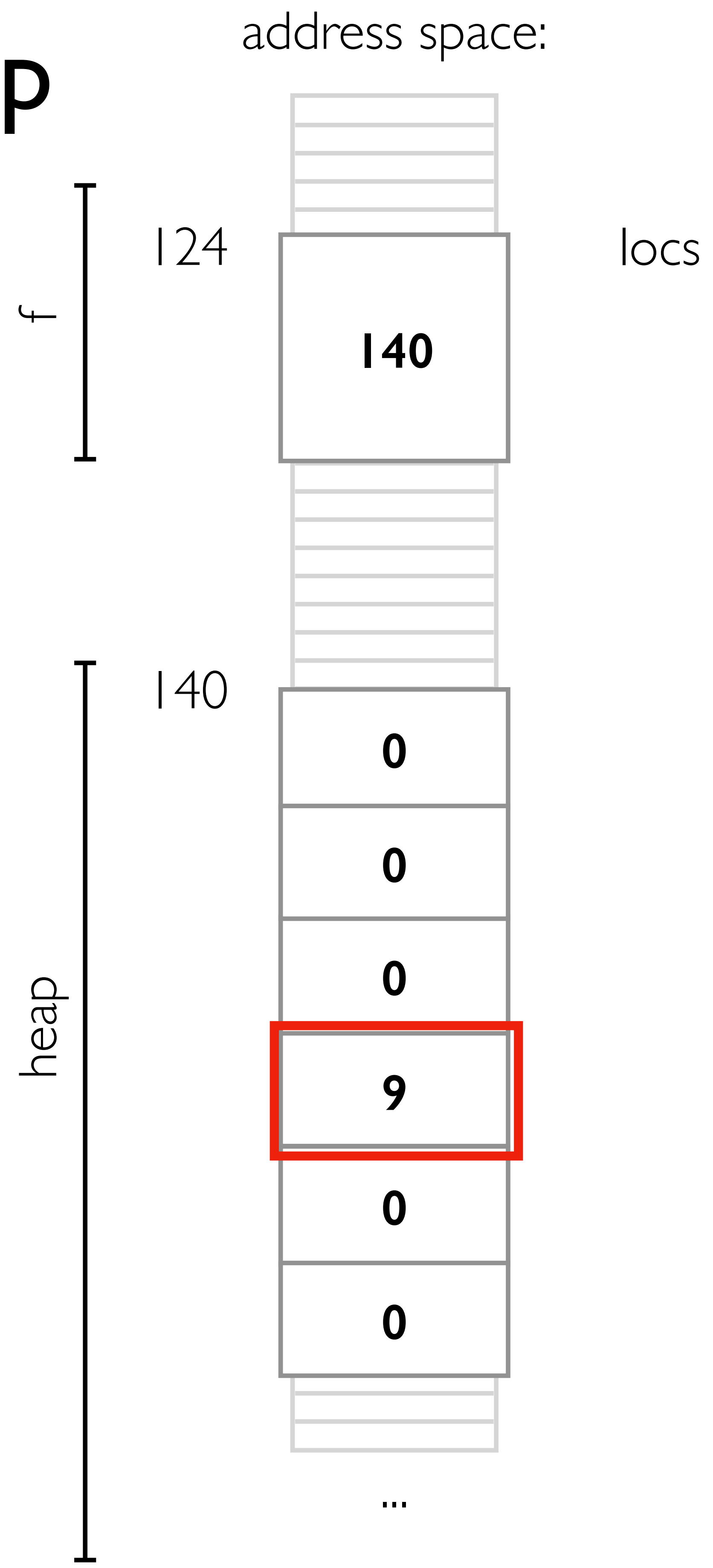
Arrays on the ~~Stack~~ Heap

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

int f() {
   Loc* locs = new Loc[3];
   locs[1]
}
```



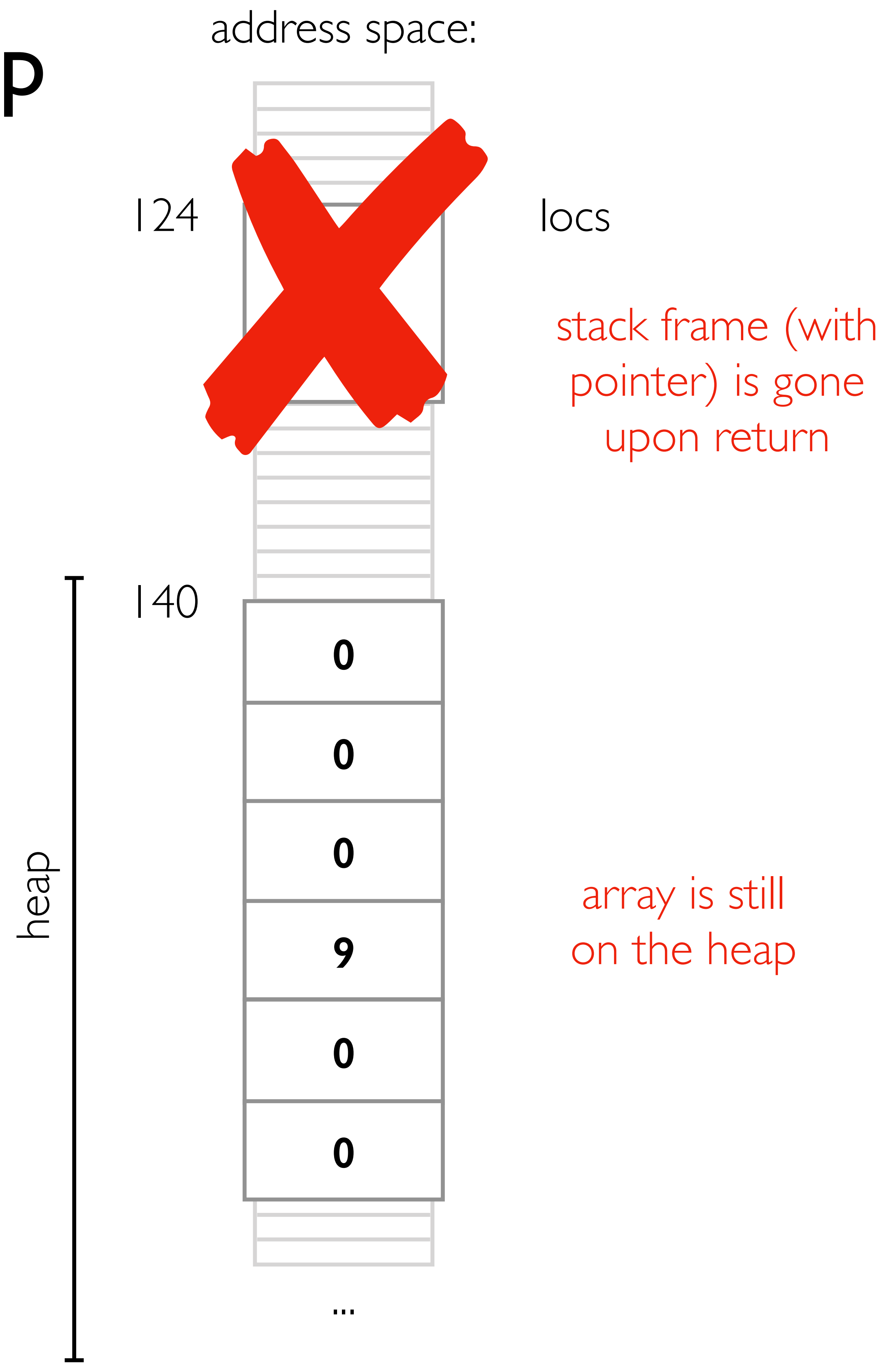
Arrays on the ~~Stack~~ Heap

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

int f() {
  Loc* locs = new Loc[3];
  locs[1].y = 9;
}
```



Arrays on the ~~Stack~~ Heap

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

int f() {
  Loc* locs = new Loc[3];
  locs[1].y = 9;
  return 123;
}
```

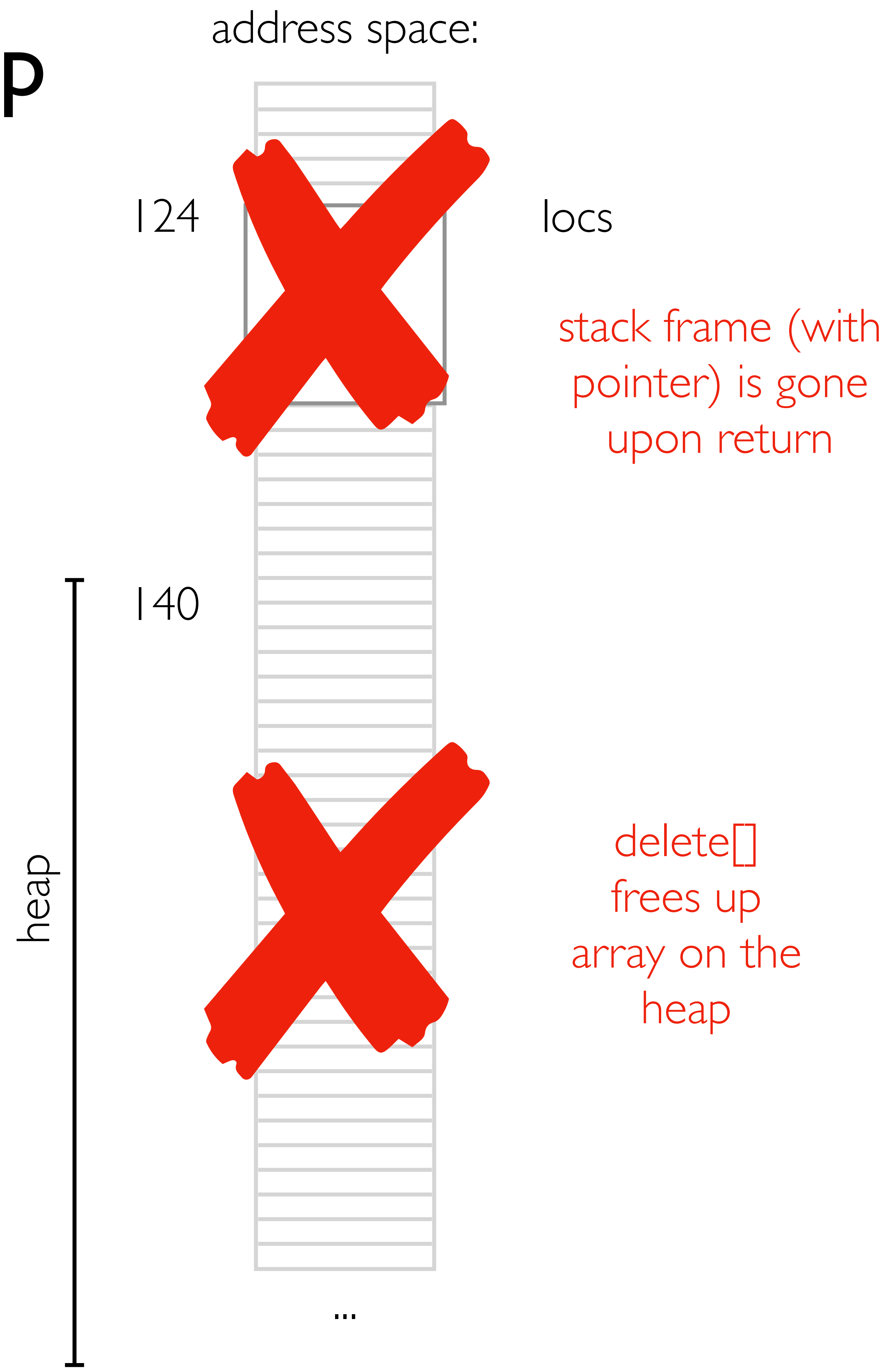


stack frame (with pointer) is gone upon return

array is still on the heap

Arrays on the ~~Stack~~ Heap

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

int f() {
   Loc* locs = new Loc[3];
   locs[1].y = 9;
   delete[] locs;
   return 123;
}
```



stack frame (with pointer) is gone upon return

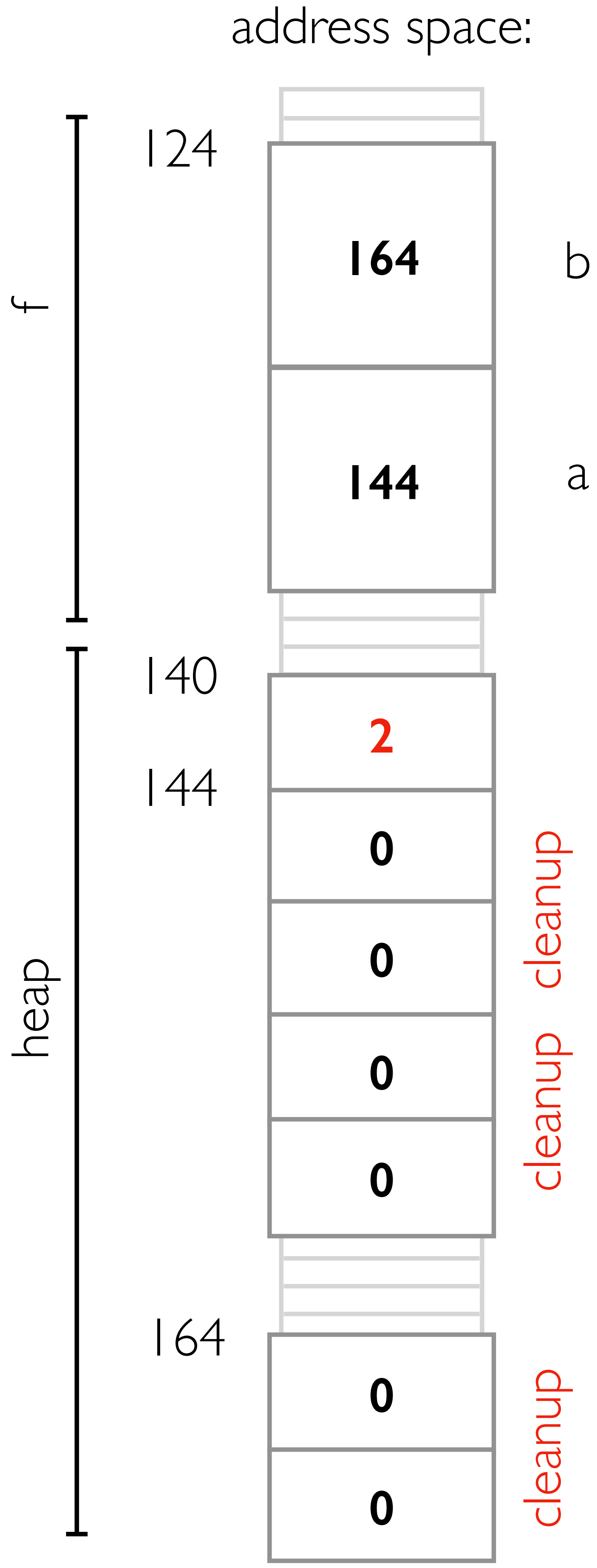
delete[] frees up array on the heap

delete vs. delete[]

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

int f() {
  Loc* b = new Loc;
  Loc* a = new Loc[2];
  delete b;
  delete[] a;
}
```

for complicated types, C++ needs to cleanup (i.e., "destroy") each object.

delete[] can tell C++ it needs to look just before the array pointer to get the size and know how many items need cleanup



Outline

TopHat and Worksheet

Memory Layout: Code/Stack/Heap

new/delete

arrays

Worksheet

Safety

Outline

TopHat and Worksheet

Memory Layout: Code/Stack/Heap

new/delete

arrays

Worksheet

Safety

- const
- references

const Motivation

Reasons to use pointers

- avoid copy
- let function modify our value

What if we DO NOT want to let a function modify our value (can lead to bugs), but still want to avoid a copy?

const lets us indicate that we may not modify a value (after it is initially set).

Disallow Value Changes

Reasons to use pointers

- avoid copy
- let function modify our value

What if we DO NOT want to let a function modify our value (can lead to bugs), but still want to avoid a copy?

const lets us indicate that we may not modify a value (after it is initially set).

```
int main() {
  int x = 3;
  int y = 4;
  const int* z = &x;

  z = &y; // modify pointer
  ~~*z = 9; // modify value~~      not allowed

  cout << x << " " << y << "\n";
}
```

Disallow Pointer Changes

Reasons to use pointers

- avoid copy
- let function modify our value

What if we DO NOT want to let a function modify our value (can lead to bugs), but still want to avoid a copy?

const lets us indicate that we may not modify a value (after it is initially set).

```
int main() {
  int x = 3;
  int y = 4;
  int* const z = &x;

  ~~z = &y; // modify pointer~~
  *z = 9; // modify value

  cout << x << " " << y << "\n";
}
```

rarely used because
references (up next!)
also offer this

not allowed

Disallow Both Changes

Reasons to use pointers

- avoid copy
- let function modify our value

What if we DO NOT want to let a function modify our value (can lead to bugs), but still want to avoid a copy?

const lets us indicate that we may not modify a value (after it is initially set).

```
int main() {
  int x = 3;
  int y = 4;
  const int* const z = &x;

~~  z = &y; // modify pointer~~   not allowed
~~  *z = 9; // modify value~~     not allowed

  cout << x << " " << y << "\n";
}
```

const Parameters

if we don't want a variable changed,
we're prevented from passing it to
a function that doesn't promise not to change it

```
void f(int *x) {
  cout << *x << "\n";
}

void g(const int *x) {
  cout << *x << "\n";
}

int main() {
  int var = 3;
  const int c = 4;

  f(&var);
~~  f(&c);~~      not allowed
  g(&var);
  g(&c);
}
```

Outline

TopHat and Worksheet

Memory Layout: Code/Stack/Heap

new/delete

arrays

Worksheet

Safety

- const
- references

References Motivation

Pointer disadvantages

- ugly syntax: &, *, ->, .
- error prone (don't forget to check if it is NULL!)

References are pointers with 3 differences:

- nicer syntax (no *, ->)
- cannot be NULL
- one reference can only point to one thing (cannot change later)

Syntax: Pointers vs. References (Diff 1)

```
Coord coord{.x=3,.y=4};    Coord coord{.x=3,.y=4};

Coord* p = &coord;         Coord& r = coord;
f(p->x);                   f(r.x);
g(*p);                     g(r);
```

nullptr (Diff 2)

```
Coord coord{.x=3,.y=4};

Coord* p = nullptr;

...

if (p)
  f(p->x);
```

```
Coord coord{.x=3,.y=4};

Coord& r = ...;  cannot be null

...


f(r.x);  no safety check needed
```

Pointing Elsewhere (Diff 3)

```
Coord coord1{.x=3,.y=4};
Coord coord2{.x=6,.y=7};

Coord* p = &coord1;
p = &coord2;
```

```
Coord coord1{.x=3,.y=4};
Coord coord2{.x=6,.y=7};

Coord& r = coord1;
```

r will always refer to coord1

Reference Recap

References are pointers with 3 differences:

- nicer syntax (no *, ->)
- cannot be NULL
- one reference can only point to one thing (cannot change later)

Places to use pointers:

- might want to change what we point to (e.g., looping over an array)
- might want to represent a missing value (with nullptr)
- return type of "new" is a pointer!

Otherwise you should probably use a reference.