CS 105C: Lecture 10

Last Time…

Smart Pointers for Fun and Profit

Multiple noncommunicating managers is an awful idea everywhere

std::unique_ptr: the manager that disallows other managers

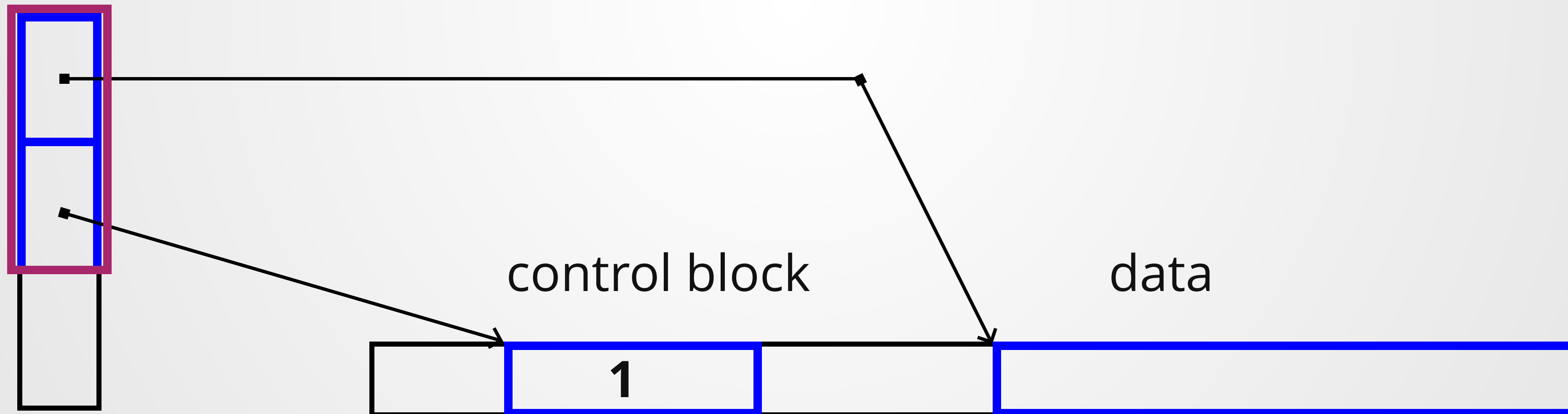
std::shared_ptr: the manager that talks to other managers

Both classes allow us to manage the lifetimes of objects using RAII

unique_ptr works by blocking copies

```
unique_ptr {
  T* ptr;
  ...
  unique_ptr(const unique_ptr& other) = delete;
  unique_ptr& operator=(const unique_ptr& other) = delete;
```

shared_ptr works by reference counting



T&& in a type-deduced context forms a forwarding pointer, which is a reference to the category it was initialized with.

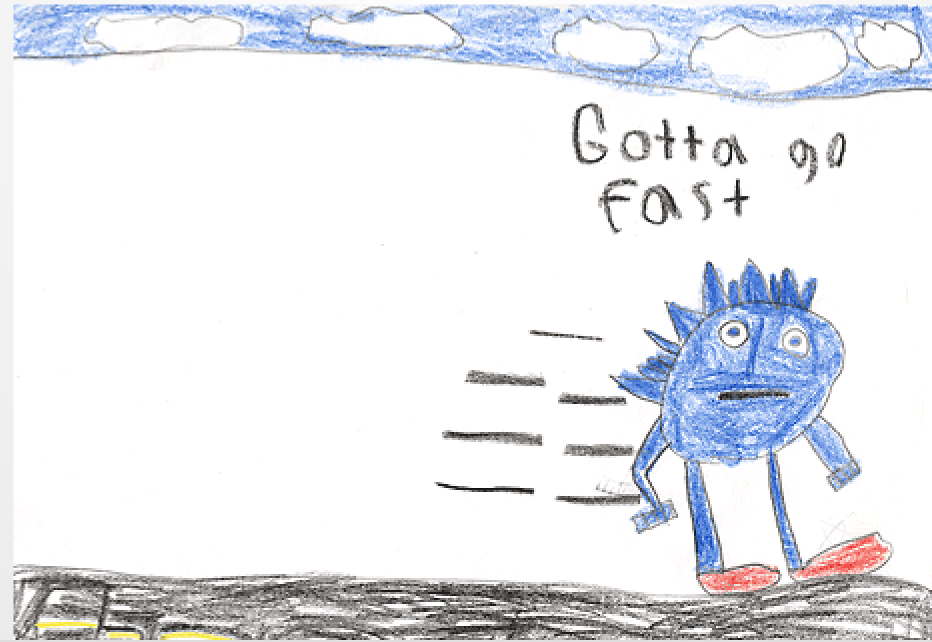
```cpp
int main(){
  int x = 3;
  auto&& rvalue_reference = 3;
  auto&& lvalue_reference = x;
}
```

We can *forward* a forwarding reference with the original value category it had by using std::forward

```cpp
template<typename T>
T make_T(T&& arg){
  T(arg); // Calls copy constructor ONLY
  T(std::forward<T>(arg)); // Calls move OR copy constructor
}
```

CS 105C: Lecture 10

Zero-cost abstractions



There are no zero-cost abstractions

https://www.youtube.com/embed/rHIkrotSwcc?enablejsapi=1

Summary

Okay, for real though

Even abstractions as basic as **function calls** and **loops** can add signficant amounts of overhead!

So much so that we have optimization techniques (inlining and unrolling) to deal with the overheads of these abstractions!

So what the **heck** does it mean when we say that C++ offers zero-cost abstractions?

“ What you don’t use, you don’t pay for. And further: What you do use, you couldn’t hand code any better.

-- Bjarne Stroustrup

In practice, something we strive for more than something we accomplish.

"What you don't use, you don't pay for"

Python: Heterogeneous Objects

```python
a = [1, "1", 1.0]
```

Python has heterogeneous collections: you can add objects of different types into e.g. a list.

How convenient!

Python: Heterogeneous Objects

```
a = [1, "1", 1.0]
```

Since we can put any object into a list, we can put different-sized objects into a list.

How do we store them?

Option 1: Store objects inline

Advantages

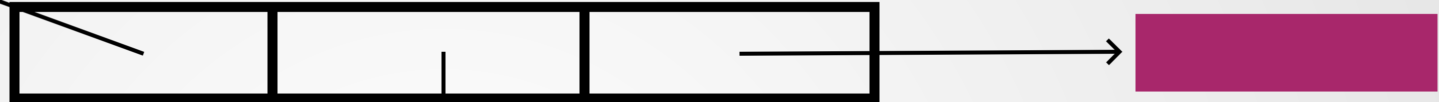
- Saves space!
- Data remains local

Disadvantages

- Indexing is now O(N)

Implications: simple for loops are now $O(N^2)$!

Option 2: Store Objects as Pointers



Advantages

- Indexing is O(1)
- Store as much data as you want

Disadvantages

- Accessing an element of the list costs two pointer lookups (and probably a cache miss or page fault)

There Is No Fundamental Way Around The Storage Problem

Any container storing heterogeneous elements ***must*** either pay an $O(N)$ indexing cost or store elements indirectly.

Heterogeneous lists have a high runtime cost.

What if I don't want to use this functionality?

```
a = [1, 2, 3]
```

Can I just place the elements in-line like I would for a homogeneous array?

What if I don't want to use this functionality?

```
a = [1, 2, 3, "4"]
```

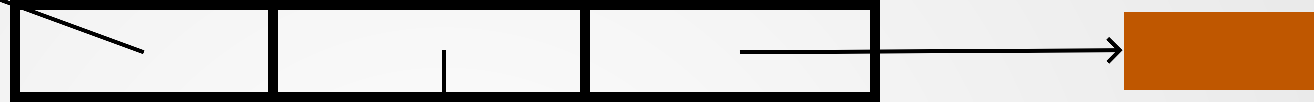
Can I just place the elements in-line like I would for a homogeneous array?

No.

As long as we *can* add elements of a different type, we have to be able to deal with it.

If we want to have a heterogeneous collection, we must pay the runtime cost *even if we don't use it that way.*

Even homogeneous collections are stored in Python indirectly!



This has very real runtime implications for Python!

Let's add 1 to a bunch of numbers to test it out...

```cpp
1 #include<vector>
2 #include<iostream>
3
4 int main(){
5   std::vector<int> a(100'000'000, 1);
6   for(auto& x : a){
7     x++;
8   }
9 }
```

```python
1 def main():
2   x = [1] * 100_000_000
3   list(map(lambda z: z+1, x))
4
5 if __name__ == "__main__":
6   main()
```

Test: Perform x+=1 on 100,000,000 ints in Python

```
~/t/C++ via 🐍 v3.8.0 took 3s
❯ hyperfine "python add.py"
Benchmark #1: python add.py
  Time (mean ± σ):      6.369 s ±  0.138 s    [User: 5.833 s, System: 0.517 s]
  Range (min … max):    6.246 s …  6.596 s    10 runs
```

Test: Perform x+=1 on 100,000,000 ints in C++

```
❯ hyperfine ./a.out
Benchmark #1: ./a.out
  Time (mean ± σ):      1.179 s ±  0.016 s    [User: 1.047 s, System: 0.127 s]
  Range (min … max):    1.159 s …  1.204 s    10 runs
```

Test: Perform x+=1 on 100,000,000 ints in C++, optimized

```
~/t/C++ via 🐍 v3.8.0
❯ hyperfine ./a.out
Benchmark #1: ./a.out
  Time (mean ± σ):     211.9 ms ±  26.7 ms    [User: 67.4 ms, System: 142.6 ms]
  Range (min … max):   188.0 ms … 283.4 ms    15 runs
```

Actually, the add operation in python isn't great either...

```c
case TARGET(BINARY_ADD): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    /* NOTE(haypo): Please don't try to micro-optimize int+int on
       CPython using bytecode, it is simply worthless.
       See http://bugs.python.org/issue21955 and
       http://bugs.python.org/issue10044 for the discussion. In short,
       no patch shown any impact on a realistic benchmark, only a minor
       speedup on microbenchmarks. */
    if (PyUnicode_CheckExact(left) &&
             PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(tstate, left, right, f, next_instr);
        /* unicode_concatenate consumed the ref to left */
    }
    else {
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
```

```c
PyObject *
PyNumber_Add(PyObject *v, PyObject *w)
{
    PyObject *result = binary_op1(v, w, NB_SLOT(nb_add));
    if (result == Py_NotImplemented) {
        PySequenceMethods *m = v->ob_type->tp_as_sequence;
        Py_DECREF(result);
        if (m && m->sq_concat) {
            return (*m->sq_concat)(v, w);
        }
        result = binop_type_error(v, w, "+");
    }
    return result;
}
```

```c
static PyObject *
binary_op1(PyObject *v, PyObject *w, const int op_slot)
{
    PyObject *x;
    binaryfunc slotv = NULL;
    binaryfunc slotw = NULL;

    if (v->ob_type->tp_as_number != NULL)
        slotv = NB_BINOP(v->ob_type->tp_as_number, op_slot);
    if (w->ob_type != v->ob_type &&
        w->ob_type->tp_as_number != NULL) {
        slotw = NB_BINOP(w->ob_type->tp_as_number, op_slot);
        if (slotw == slotv)
            slotw = NULL;
    }
    if (slotv) {
        if (slotw && PyType_IsSubtype(w->ob_type, v->ob_type)) {
            x = slotw(v, w);
            if (x != Py_NotImplemented)
                return x;
            Py_DECREF(x); /* can't do it */
            slotw = NULL;
        }
        x = slotv(v, w);
        if (x != Py_NotImplemented)
            return x;
        Py_DECREF(x); /* can't do it */
    }
    if (slotw) {
        x = slotw(v, w);
        if (x != Py_NotImplemented)
            return x;
        Py_DECREF(x); /* can't do it */
    }
    Py_RETURN_NOTIMPLEMENTED;
}
```

More Examples

Garbage Collection

Recall: Stack allocation/deallocation is lightning fast,
but size of object needs to be known at compile-time.

Unknown-size objects have to be allocated on the
heap and cleaned up later.

```
1 int* read_unknown_data(const std::string& filename){
2   std::istream inf(filename);
3   int num_elements;
4   inf >> num_elements;
5
6   int* data = new int[num_elements]; // Ugh, when can I delete this?
7 }
```

Garbage Collection

Garbage collection makes this fun and easy, at the cost of a little extra time to run the GC algorithm!

```
1  class Program{
2    public int[] readInput(String fname)
3      Scanner scanner = new Scanner(new File(fname));
4      int[] vals = new int[scanner.nextInt()];  // Java will take care of it!
5      for(int i = 0; i < vals.length){
6        vals[i++] = scanner.nextInt();
7      }
8      return vals;
9  }
```

But there's a small cost for *every single value* we use, and that can add up.

Garbage Collection

Can I tell the garbage collector to only worry about certain data in my heap?

No.

Garbage Collection

If I'm working in a garbage collected language, I have to pay the GC cost even if I know ahead of time exactly when all my memory can be deallocated!

I can manually force garbage collection to occur, but I can't tell the language "trust me, drop that memory now" without violating gc safety requirements.

Green Threads

A lightweight threading system which relies on the **language runtime** to schedule threads instead of the OS. Found in:

- Java (pre-v1.1)
- PyPy/Stackless Python
- Erlang/Elixir
- Go (goroutines)
- Julia (tasks)
- Haskell
- Ruby (pre-v1.9)
- Rust (pre-v1.0)

Green Threads

```go
func f(from string) {
	for i := 0; i < 3; i++ {
		fmt.Println(from, ":", i)
	}
}

func main() {
	go f("goroutine")
	go func(msg string) {
		fmt.Println(msg)
	}("going")
}
```

```
goroutine : 0
going
goroutine : 1
goroutine : 2
```

Threads managed by Go, **not** by the OS!

example from https://gobyexample.com/goroutines

Green Threads

Can I tell my programming language that I don't want to use green threads?

Yes. Just don't use green threads.

Can I remove the code dealing with threads from the language runtime to reduce the complexity/binary size of the language?

No.

A warning about performance

“ Use your intuition to ask questions, not answer them

--John Osterhout, inventor of Tcl

Software engineers are full of horror stories of someone who spent weeks optimizing the {memory access, program logic, parallelization} only to realize that they didn't actually make the program run any faster!

“ Use your intuition to ask questions, not answer them

--John Osterhout, inventor of Tcl

We can look at some code and say "it looks like this should be slower"...

...but we **must** back this up with measurements showing it to be the case!

C++ Zero-Cost Abstractions

std::unique_ptr

std::unique_ptr

Do we have to pay a cost for unique_ptr if we don't use it?

Runtime cost	Compile time cost	Code Complexity Cost
No.	No: no template is instantiated so there's no extra stuff to compile	No: unique_ptr (in its simplest form) just wraps new and delete

std::unique_ptr

Could we write it better ourselves?

If we use handwritten allocation instead of letting unique_ptr allocate stuff for us, do we lose any performance?

Does it use more memory?

```cpp
#include<memory>
#include<iostream>

int main(){
  int* databuf = new int[10000];
  int* rp = databuf;
  std::unique_ptr<int> up(databuf);

  std::cout << "Size of unique ptr = " << sizeof(up) << '\n';
  std::cout << "Size of raw ptr = " << sizeof(rp) << '\n';
}
```

```
❯ ./a.out
Size of unique ptr = 8
Size of raw ptr = 8
```

Apparently not.

Does it take more time to run?

Allocate + Deallocate 100,000,000 ints
using smart pointers and raw pointers

Compiler	Optimization	new	std::shared_ptr	std::make_shared	std::unique_ptr	std::make_unique
GCC	no	3.03	13.48	30.47	8.74	9.09
GCC	yes	3.03	6.42	3.24	3.07	3.04
cl.exe	no	8.79	25.17	18.75	11.94	13.00
cl.exe	yes	7.42	17.29	9.40	7.58	7.68

Less than a 0.1% difference in a program
that spends all of its time doing allocation!

If your program spends 5% of its time
doing allocation, this is a 0.05% difference

Conclusion: not really enough to matter.

No extra cost if unused

Just as fast as handwritten code

std::unique_ptr is a zero-cost abstraction

C++ Zero-Cost Abstractions

std::vector

std::vector

Do we have to pay a cost for vector if we don't use it?

Runtime cost	Compile time cost	Code Complexity Cost
No.	No: no template is instantiated so there's no extra stuff to compile	Minimal: the code for the abstraction has to exist, but we don't have to put it into *our* program.

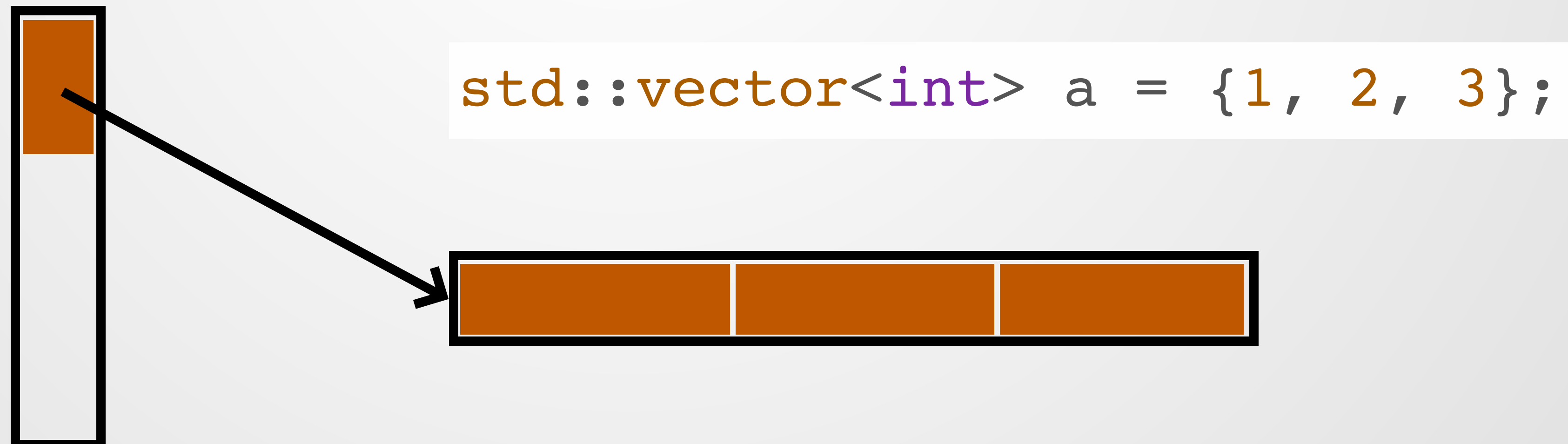
What you do use, you couldn't hand code any better.

Do we need to pay the indirect storage cost like we do in Python?

After all, we can store any type of data in a C++ vector…

```
1 std::vector<int>
2 std::vector<float>
3 std::vector<Cow>
```

Since any given vector only stores one type of data, we can store all of them in-line in the heap!



Element Access

Element access in a raw array is a simple procedure:

- Find the array address
- Add the appropriate offset
- Load the element from offset

```
1  _Z8access_NPPii:
2  .LFB854:
3      .cfi_startproc
4      movq    (%rdi), %rax
5      movslq  %esi, %rsi
6      movl    (%rax,%rsi,4), %eax
7      ret
8      .cfi_endproc
```

__In principle__, access in a vector is harder:

- Find operator[] in vector method
- Resolve operator [] overloads
- Access data buffer within class
- Add appropriate offset
- Load element from offset

```
1  _Z8access_NRSt6vectorIiSaIiEEi:
2  .LFB853:
3      .cfi_startproc
4      movq    (%rdi), %rax
5      movslq  %esi, %rsi
6      movl    (%rax,%rsi,4), %eax
7      ret
8      .cfi_endproc
```

Vector Copy

How do I make a copy of a vector in C++?
(Without using the built-in copy constructor)

```
1  template <typename T>
2  std::vector<T> makeCopy(const std::vector<T>& in){
3      std::vector<T> out(in.size());
4      for(int i = 0; i < in.size(); i++){
5          out[i] = in[i];
6      }
7      return out;
8  }
```

Time for 10 million elements:

97 milliseconds

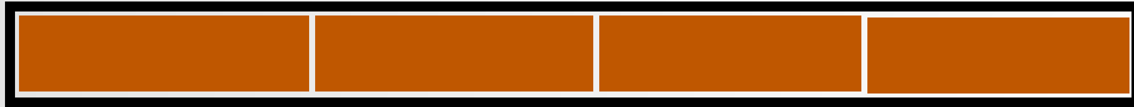
```
1  vector<int> makeCopySTL(const vector<int> &a) {
2    vector<int> x = a;
3    return x;
4  }
```

Time for 10 million elements:

13 milliseconds

How is the copy constructor so much faster??

Most modern processors provide **vector units** which can load/store *multiple addresses at once*.



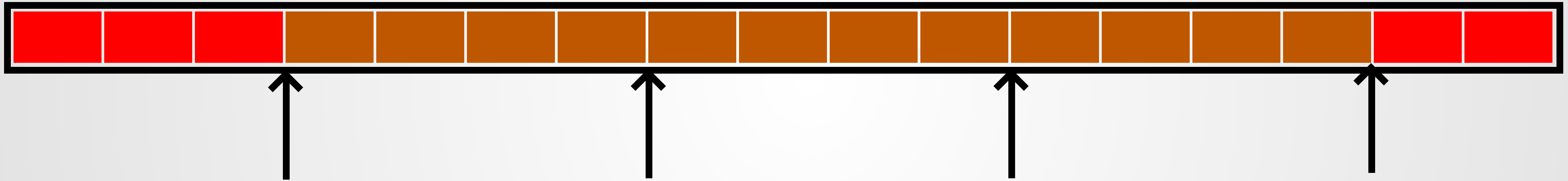
AVX2, per CPU cycle

load/store, per CPU cycle

But there's a catch: the memory block has to be *aligned* to a multiple of 16! Most memory allocated in a program doesn't have this property!

Memory Alignment

The vector units require that memory is *aligned* (e.g. the beginning address is a multiple of 16). An actual data buffer may not have this property.

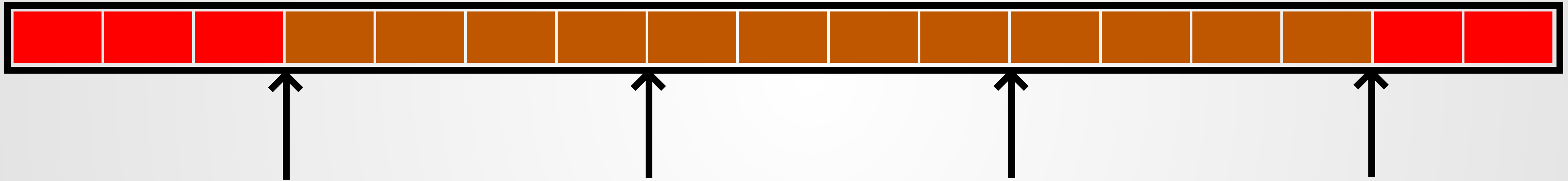


Orange blocks can be copied via aligned vector instructions--red blocks cannot.

Should we give up on using vector instructions because some of the memory is uneligible?

Solution

Copy orange blocks via vector instructions, then clean up red blocks with manual copying!



Determining which blocks can be copied and which cannot at runtime is a little annoying....

The vector move constructor does all of this by default!

```
1  _Z11makeCopySTLRKSt6vectorIiSaIiEE:
2  .LFB853:
3      .cfi_startproc
4  ;; blah blah blah
5  .L3:
6          ; Associated code does "patch-up" on red blocks which can't be copied by aligned memmove
7      movq    %rcx, %xmm0
8      addq    %rcx, %rbx
9      punpcklqdq  %xmm0, %xmm0
10     movq    %rbx, 16(%r12)
11     movups  %xmm0, (%r12)
12     movq    8(%rbp), %rax
13     movq    0(%rbp), %rsi
14     movq    %rax, %rbx
15     subq    %rsi, %rbx
16     cmpq    %rsi, %rax
17     je  .L6
18     movq    %rcx, %rdi
19     movq    %rbx, %rdx
20     call    memmove@PLT     ; Does accelerated memory copies using vector instructions!
21     movq    %rax, %rcx
```

Reality is much more complicated than presented here--sometimes both your source and destination need to be aligned!

Can you code the vector copy better by hand?

…maybe you could. I probably couldn't.

Summary

Zero-Cost Abstractions

Broadly try to fulfill two goals:

- Do not affect the runtime/complexity of the language if unused
- Are nearly as efficient as handwritten versions of the same code

Some abstractions that don't qualify:

Heterogeneous Collections

Force layout changes in data structure that affect all usages of that collection

Garbage Collection

All data must be collected by the GC, even if we know exactly when it can be safely collected

Green Threads

Can opt not to use them, but implementation must remain in the language runtime

Zero-Cost Abstractions in C++

std::vector

Uses compile-time information to make operations just as fast as handwritten counterparts

Vector copy scheme is *faster* than naive handwritten code!

std::unique_ptr

Uses no more memory and is essentially no slower than a raw pointer.

Notecards

- Name and EID
- One thing you learned today (can be "nothing")
- One question you have about the material. **If you leave this blank, you will be docked points.**

  If you do not want your question to be put on Piazza, please write the letters **NPZ** and circle them.