Breaking `std::vector`'s ABI

for performance gains

A horror story

What is `std::vector`?

What happens when we call `vector::push_back`?

こんにちは、LLVM

```
1 auto v = std::vector<char32_t>{};
2 v.push_back(U'こ');
3 v.push_back(U'ん');
4 v.push_back(U'に');
5 v.push_back(U'ち');
6 v.push_back(U'は');
```

```
auto v = std::vector<char32_t>{}
```

```
Capacity: 0
  Size: 0
```

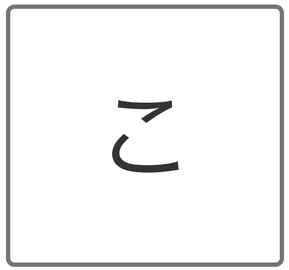
```
v.push_back(U'こ')
```

```
Capacity: 0
  Size: 0
```

v.push_back(U'こ')

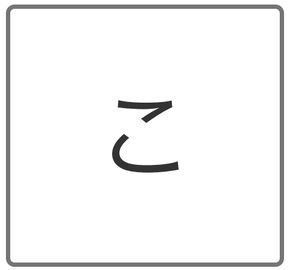
Capacity: 1
Size: 0

v.push_back(U'こ')



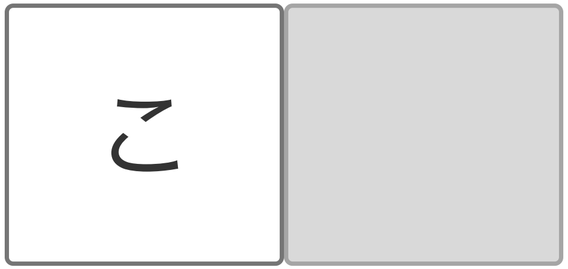
Capacity: 1
Size: 1

```
v.push_back(U'ん')
```




```
Capacity: 1
  Size: 1
```

v.push_back(U'ん')



Capacity: 2
Size: 1

v.push_back(U'ん')

Capacity: 2
Size: 2

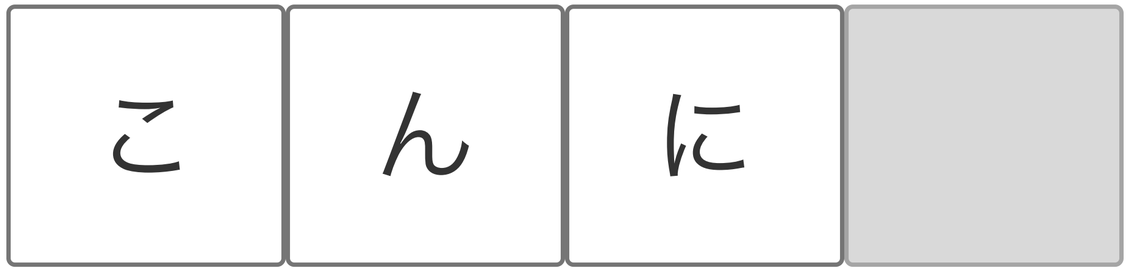
v.push_back(U'に')

Capacity: 2
Size: 2

v.push_back(U'に')

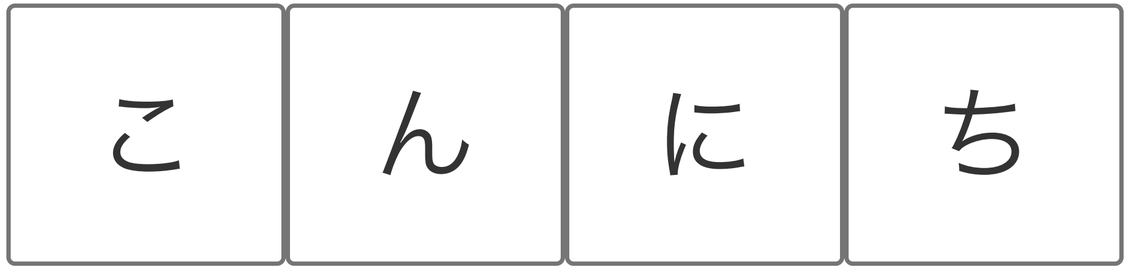
Capacity: 4
Size: 2

v.push_back(U'に')



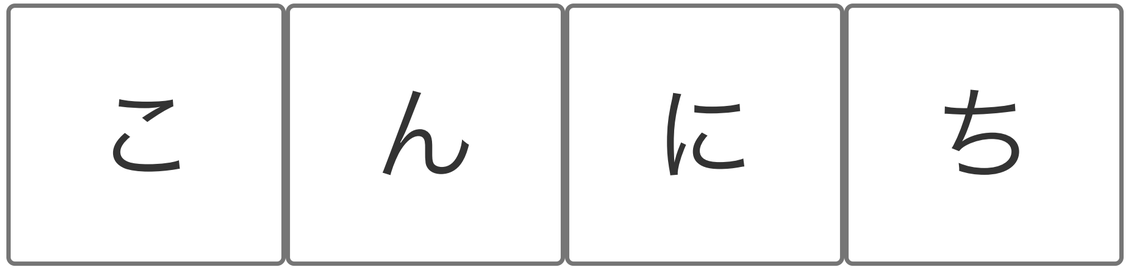
Capacity: 4
Size: 3

v.push_back(U'ち')



Capacity: 4
Size: 4

v.push_back(U'は')

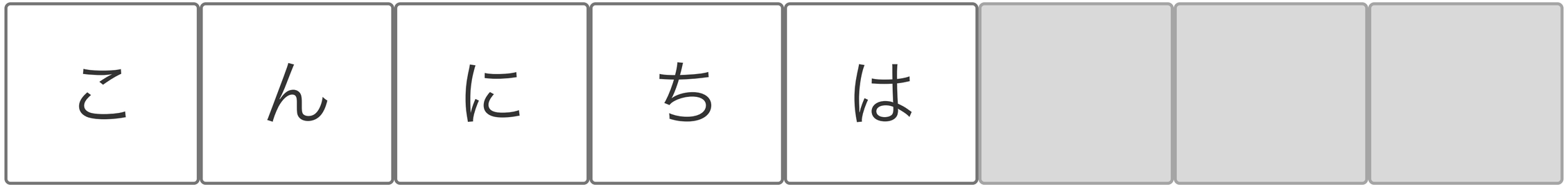


Capacity: 4
Size: 4

v.push_back(U'は')

Capacity: 8
Size: 4

v.push_back(U'は')



Capacity: 8
Size: 5

What's the most intuitive way to implement this?

```
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return data_[i]; }

  size_type size() const     { return size_; }
  size_type capacity() const { return capacity_; }

  iterator begin() { return iterator(data_); }
  iterator end()   { return iterator(data_ + size_); }

  // ...
private:
  T*        data_;
  size_type size_;
  size_type capacity_;
};
```

```
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return data_[i]; }

  size_type size() const     { return size_; }
  size_type capacity() const { return capacity_; }

  iterator begin() { return iterator(data_); }
  iterator end()   { return iterator(data_ + size_); }

  // ...
private:
  T*        data_;
  size_type size_;
  size_type capacity_;
};
```

```cpp
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return data_[i]; }

  size_type size() const     { return size_; }
  size_type capacity() const { return capacity_; }

  iterator begin() { return iterator(data_); }
  iterator end()   { return iterator(data_ + size_); }

  // ...
private:
  T*        data_;
  size_type size_;
  size_type capacity_;
};
```

```
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return data_[i]; }

  size_type size() const     { return size_; }
  size_type capacity() const { return capacity_; }

  iterator begin() { return iterator(data_); }
  iterator end()   { return iterator(data_ + size_); }

  // ...
private:
  T*        data_;
  size_type size_;
  size_type capacity_;
};
```

```
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return data_[i]; }

  size_type size() const     { return size_; }
  size_type capacity() const     { return capacity_; }

  iterator begin() { return iterator(data_); }
  iterator end()   { return iterator(data_ + size_); }

  // ...
private:
  T*        data_;
  size_type size_;
  size_type capacity_;
};
```

How does libc++ implement it?

```
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return begin_[i]; }

  size_type size() const     { return end_ - begin_; }
  size_type capacity() const { return capacity_ - begin_; }

  iterator begin() { return iterator(begin_); }
  iterator end()   { return iterator(end_); }

  // ...
private:
  T* begin_;
  T* end_;
  T* capacity_;
};
```

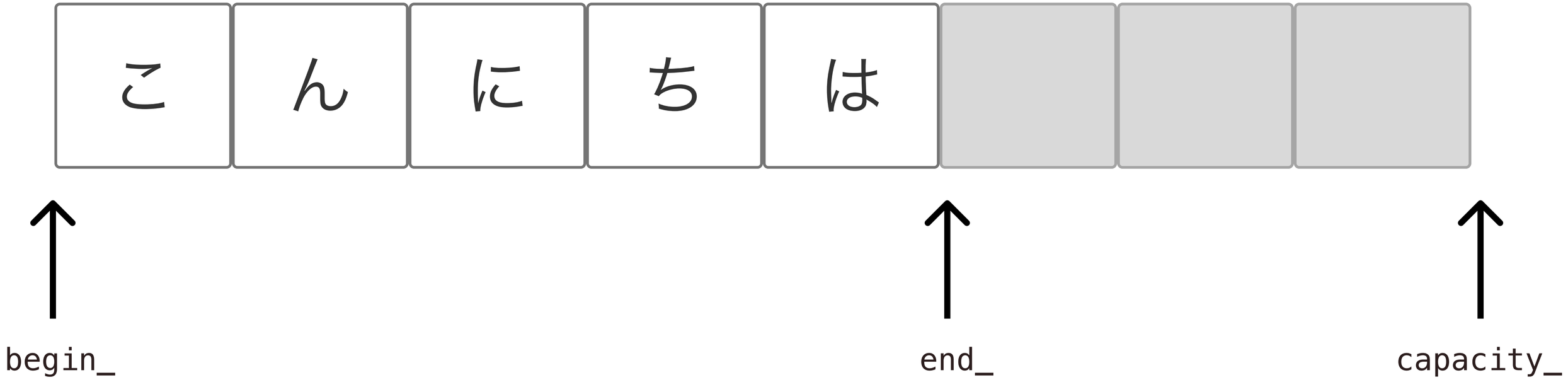
```
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return begin_[i]; }

  size_type size() const     { return end_ - begin_; }
  size_type capacity() const { return capacity_ - begin_; }

  iterator begin() { return iterator(begin_); }
  iterator end()   { return iterator(end_); }

  // ...
private:
  T* begin_;
  T* end_;
  T* capacity_;
};
```

```
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return begin_[i]; }

  size_type size() const     { return end_ - begin_; }
  size_type capacity() const { return capacity_ - begin_; }

  iterator begin() { return iterator(begin_); }
  iterator end()   { return iterator(end_); }

  // ...
private:
  T* begin_;
  T* end_;
  T* capacity_;
};
```

```cpp
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return begin_[i]; }

  size_type size() const     { return end_ - begin_; }
  size_type capacity() const { return capacity_ - begin_; }

  iterator begin() { return iterator(begin_); }
  iterator end() { return iterator(end_); }

  // ...
private:
  T* begin_;
  T* end_;
  T* capacity_;
};
```

```
template<class T>
class vector {
public:
  // ...

  T& operator[](size_type i) { return begin_[i]; }

  size_type size() const     { return end_ - begin_; }
  size_type capacity() const { return capacity_ - begin_; }

  iterator begin() { return iterator(begin_); }
  iterator end() { return iterator(end_); }

  // ...
private:
  T* begin_;
  T* end_;
  T* capacity_;
};
```

libc++'s vector, graphically represented



Why this way?

Compare the pair

Case 1:

`sizeof(vector::value_type) == 2^n`

Assembly diff (armv8-a)

	Pointer-based	Size-based

`size()`	`ldp x9, x8, [x0]`
	`sub x8, x8, x9`
	`asr x0, x8, #4`
	`ret`
		`ldr x0, [x0, #8]`
		`ret`

`end()`	`ldr x0, [x0, #8]`
	`ret`
		`ldp x8, x9, [x0]`
		`add x0, x8, x9, lsl #4`
		`ret`

Case 2:

`sizeof(vector::value_type) != 2^n`

Assembly diff (armv8-a)

Pointer-based

`size()`

```
ldp     x9, x8, [x0]
sub     x8, x8, x9
mov     x9, #-6148914691236517206
asr     x8, x8, #3
movk    x9, #43691
mul     x0, x8, x9
ret
```

`end()`

```
ldr     x0, [x0, #8]
ret
```

Size-based

```
ldr     x0, [x0, #8]
ret
```

```
ldp     x10, x8, [x0]
mov     w9, #24
madd    x0, x8, x9, x10
ret
```

Assembly diff (armv8-a)

Pointer-based `size()`

```
ldp     x9, x8, [x0]
sub     x8, x8, x9
mov     x9, #-6148914691236517206
asr     x8, x8, #3
movk    x9, #43691
mul     x0, x8, x9
ret
```

Size-based `end()`

```
ldp     x10, x8, [x0]
mov     w9, #24
madd    x0, x8, x9, x10
ret
```

Assembly diff (x86_64)

Pointer-based `size()`

```
mov     rcx, qword ptr [rdi + 8]
sub     rcx, qword ptr [rdi]
sar     rcx, 3
movabs  rax, -6148914691236517205
imul    rax, rcx
ret
```

Size-based `end()`

```
mov     rax, qword ptr [rdi + 8]
lea     rax, [rax + 2*rax]
shl     rax, 3
add     rax, qword ptr [rdi]
ret
```

It becomes more important with hardening enabled

Hardened libc++

```
T& vector<T>::operator[](size_type i) const {
  HARDENED_ASSERT(i < size(), "vector[] index out of bounds");
  return begin_[i];
}

auto const v = std::vector<T>{0};

// somewhere else in the code...
std::println("{}", v[0]); // okay
std::println("{}", v[1]); // program crashes with message
                          // "vector[] index out of bounds"
```

Hardened libc++

```
T& vector<T>::operator[](size_type i) const {
  HARDENED_ASSERT(i < size(), "vector[] index out of bounds");
  return begin_[i];
}

auto const v = std::vector<T>{0};

// somewhere else in the code...
std::println("{}", v[0]); // okay
std::println("{}", v[1]); // program crashes with message
                          // "vector[] index out of bounds"
```

Hardened libc++

```
T& vector<T>::operator[](size_type i) const {
  HARDENED_ASSERT(i < size(), "vector[] index out of bounds");
  return begin_[i];
}

auto const v = std::vector<T>{0};

// somewhere else in the code...
std::println("{}", v[0]); // okay, but unconditionally calls size()
std::println("{}", v[1]); // program crashes with message
                          // "vector[] index out of bounds"
```

How much does this actually matter?

How much does this actually matter?

A lot.

Microbenchmarks are unreliable

Macrobenchmarking on load tests

Servers	0.2–0.5% additional queries per second
Most important servers	0.12% decrease in operational costs

These are all predictions... we want *real* data

Prod results

- Server CPU time spent in:
  - `vector::size`: down from 0.6% to 0.04%
  - `vector::end`: unchanged
- Overall: 0.15~0.2% decrease in time spent in vector operations, company-wide

Deploying at scale 👹

A tale of Hyrum's Law

What is “Hyrum's law”?

> *With a sufficient number of users of an API,*
> *it does not matter what you promise in the contract:*
> *all observable behaviors of your system*
> *will be depended on by somebody.*

—Hyrum Wright, https://www.hyrumslaw.com

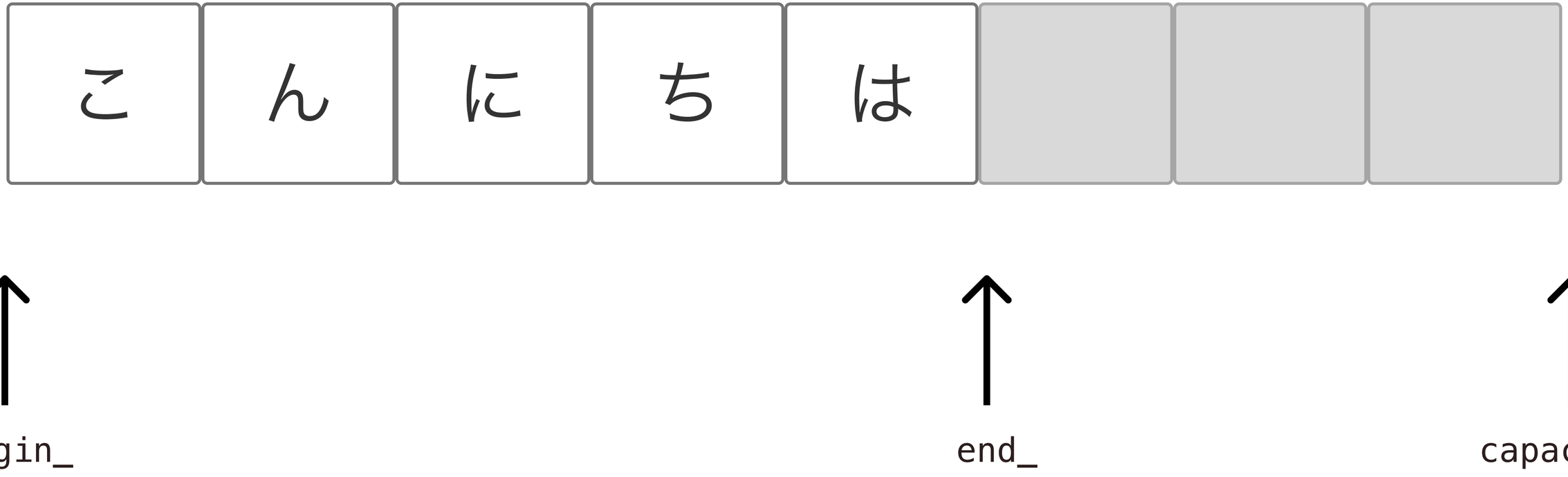
```
void resize_type_erased_vector(void* obj, size_t len, int flags) {
  auto*  v           = reinterpret_cast<std::vector<uint8_t>*>(obj);
  size_t object_size = EXTRACT_SIZE(flags);

  switch(object_size) {
  case 1:
    reinterpret_cast<std::vector<uint8_t>*>(v)->resize(len);
    return;
  case 2:
    reinterpret_cast<std::vector<uint16_t>*>(v)->resize(len);
    return;
  // ...
  }
}
```

```
void resize_type_erased_vector(void* obj, size_t len, int flags) {
  auto*  v           = reinterpret_cast<std::vector<uint8_t>*>(obj);
  size_t object_size = EXTRACT_SIZE(flags);

  switch(object_size) {
  case 1:
    reinterpret_cast<std::vector<uint8_t>*>(v)->resize(len);
    return;
  case 2:
    reinterpret_cast<std::vector<uint16_t>*>(v)->resize(len);
    return;
  // ...
  }
}
```

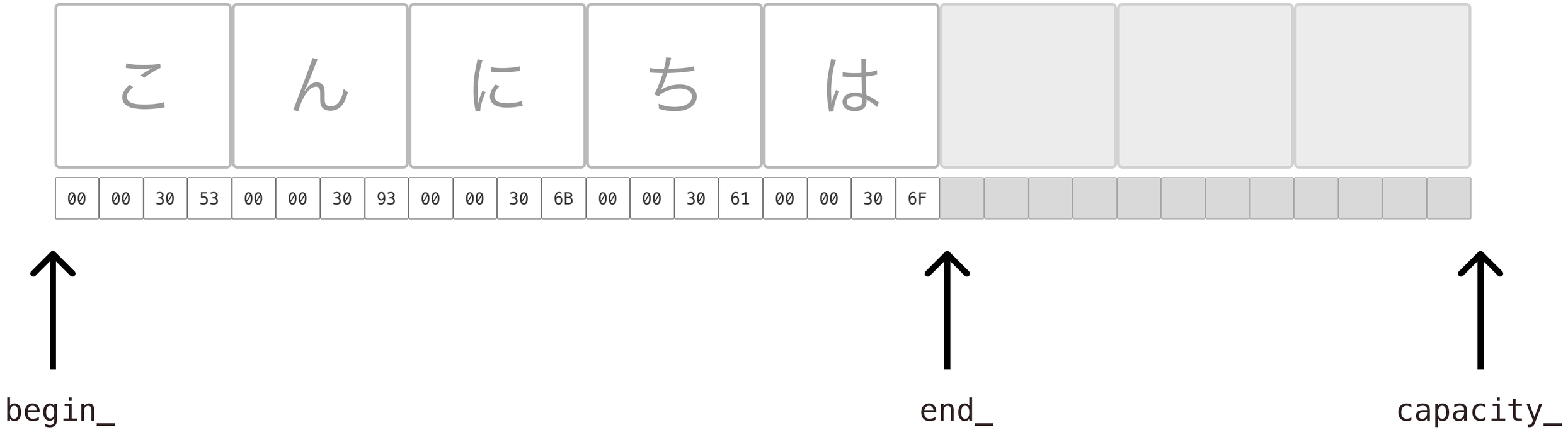
```
void resize_type_erased_vector(void* obj, size_t len, int flags) {
  auto*  v           = reinterpret_cast<std::vector<uint8_t>*>(obj);
  size_t object_size = EXTRACT_SIZE(flags);

  switch(object_size) {
  case 1:
    reinterpret_cast<std::vector<uint8_t>*>(v)->resize(len);
    return;
  case 2:
    reinterpret_cast<std::vector<uint16_t>*>(v)->resize(len);
    return;
  // ...
  }
}
```

```
void resize_type_erased_vector(void* obj, size_t len, int flags) {
  auto*  v           = reinterpret_cast<std::vector<uint8_t>*>(obj);
  size_t object_size = EXTRACT_SIZE(flags);

  switch(object_size) {
  case 1:
    reinterpret_cast<std::vector<uint8_t>*>(v)->resize(len);
    return;
  case 2:
    reinterpret_cast<std::vector<uint16_t>*>(v)->resize(len);
    return;
  // ...
  }
}
```

libc++'s vector, graphically represented

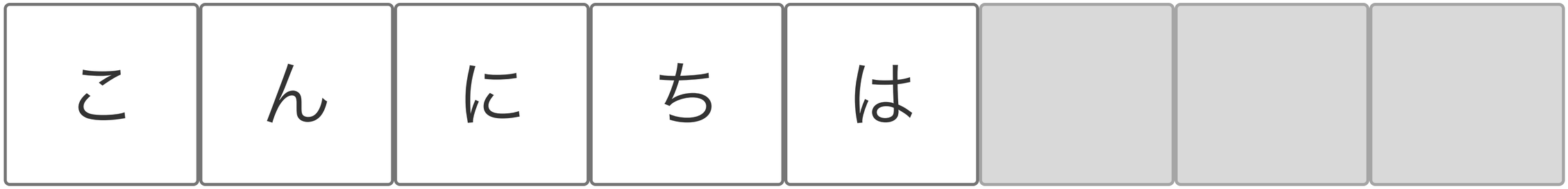


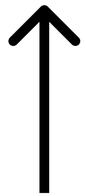
reinterpret_cast<vector<uint8_t>>(v)



```
reinterpret_cast<vector<uint8_t>>(v)
```

Size-based vector, graphically represented





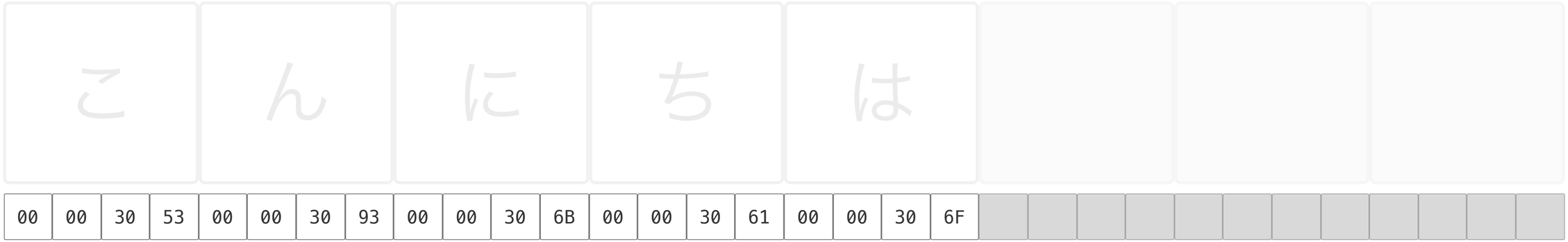
`begin_`

`begin_`
`+ size_`

`begin_`
`+ capacity_`

`Capacity: 8`
`Size: 5`

reinterpret_cast<vector<uint8_t>>(v)



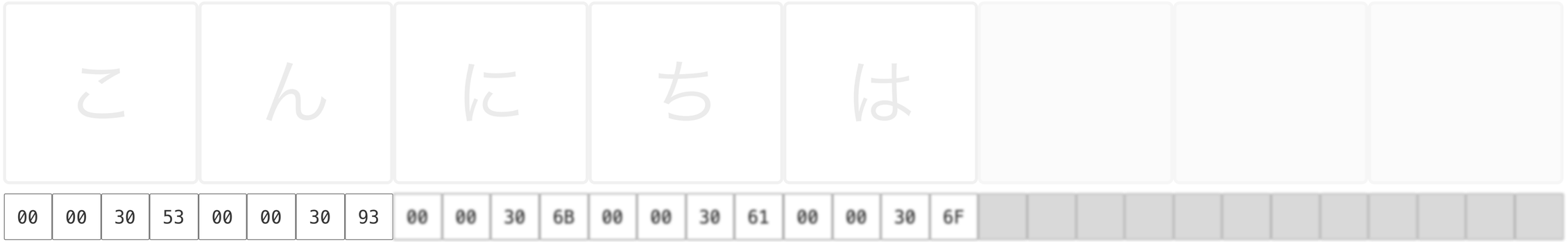
begin_

begin_
+ size_

begin_
+ capacity_

Capacity: 8
Size: 5

reinterpret_cast<vector<uint8_t>>(v)



Capacity: 8
Size: 5

reinterpret_cast<vector<uint8_t>>(v)

begin_ + size_ + capacity_

Capacity: 8
Size: 5

```
=================================================================
==2812247==ERROR: AddressSanitizer: new-delete-type-mismatch on 0x7d12c31e0040 in thread
  object passed to delete has wrong type:
  size of the allocated type:   320 bytes;
  size of the deallocated type: 80 bytes.
    #0 0x55f1639da0f2 in operator delete(void*, unsigned long) /tmp/llvm-project/compile
    #1 0x55f1639debec in void std::__2::__libcpp_operator_delete[abi:fe210000]<unsigned
    #2 0x55f1639deb8d in void std::__2::__libcpp_deallocate[abi:fe210000]<unsigned char>
    #3 0x55f1639deb25 in std::__2::allocator<unsigned char>::deallocate[abi:fe210000](un
    #4 0x55f1639de914 in std::__2::allocator_traits<std::__2::allocator<unsigned char>>:
    #5 0x55f1639dd99a in std::__2::__size_based_split_buffer<unsigned char, std::__2::al
    #6 0x55f1639dcc95 in std::__2::vector<unsigned char, std::__2::allocator<unsigned ch
    #7 0x55f1639db18b in std::__2::vector<unsigned char, std::__2::allocator<unsigned ch
    #8 0x55f1639daaac in main (/tmp/example+0x116aac)
    #9 0x7ff2c3df0ca7 in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main
    #10 0x7ff2c3df0d64 in __libc_start_main csu/../csu/libc-start.c:360:3
    #11 0x55f1638f2480 in _start (/tmp/example+0x2e480)

0x7d12c31e0040 is located 0 bytes inside of 320-byte region [0x7d12c31e0040,0x7d12c31e01
allocated by thread T0 here:
    #0 0x55f1639d948d in operator new(unsigned long) /tmp/llvm-project/compiler-rt/lib/a
    #1 0x55f1639dbf64 in void* std::__2::__libcpp_operator_new[abi:fe210000]<unsigned lo
    #2 0x55f1639dbee5 in int* std::__2::__libcpp_allocate[abi:fe210000]<int>(std::__2::_
    #3 0x55f1639dbe44 in std::__2::allocator<int>::allocate[abi:fe210000](unsigned long)
    #4 0x55f1639dba3c in std::__2::__allocation_result<std::__2::allocator_traits<std::_
    #5 0x55f1639db407 in std::__2::vector<int, std::__2::allocator<int>>::__vallocate[ab
    #6 0x55f1639dad89 in std::__2::vector<int, std::__2::allocator<int>>::vector[abi:fe2
```

```
class VectorBase {
  virtual void resize_impl(size_t len, int flags) = 0;
};

template<class T>
class VectorT : public VectorBase {
  std::vector<T> data_;

  void resize_impl(size_t len, int) override {
    data_.resize(len);
  }
};

class NonVector : public VectorBase {
  void resize_impl(size_t len, int flags) override {
    // some other impl here...
  }
};
```

Upstreaming

- RFC on Discourse: 'Adding a size-based vector to libc++'s unstable ABI'
- GitHub PR#139632

Recap

- `std::vector` is historically implemented using three pointers
- Tracking size and capacity as integers significantly improves performance
- Changing the representation breaks ABI
- Hyrum's law is always lurking