Dynamic Storage Exercise

Dynamic Storage Exercise

```
int i;
while (std::cin >> i) ...
```

reads inputs as long as there are more available.

Write a code snippet which reads inputs as described above, and which then stores these inputs in an array. For this exercise you are not allowed to use the Standard Library (i.e. **no** `std::vector`).

To achieve this you will have to use `new[]` and `delete[]`.

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full,

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full,

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full,

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range
  4. Delete initial range (using `delete[]`)

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range
  4. Delete initial range (using `delete[]`)
  5. Go back to 2. with newly generated memory

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range
  4. Delete initial range (using `delete[]`)
  5. Go back to 2. with newly generated memory

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range
  4. Delete initial range (using `delete[]`)
  5. Go back to 2. with newly generated memory

(From: Script Exercise 158.a)

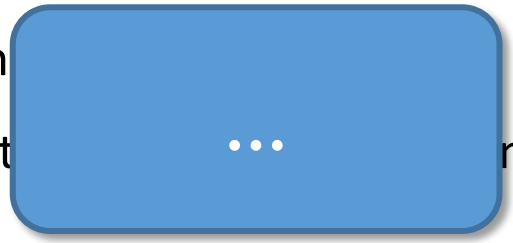
Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range
  4. Delete initial range (using `delete[]`)
  5. Go back to 2. with newly generated memory

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Idea:
  1. Allocate some range (using `new[]`)
  2. As soon as range full, allocate larger range (using `new[]`)
  3. Copy over initial range
  4. Delete initial ran
  5. Go back to 2. wit mory



(From: Script Exercise 158.a)

Dynamic Storage Solution

- And the code…

```
int n = 1; // current array size
int k = 0; // number of elements read so far

// dynamically allocate array
int* a = new int[n]; // this time, n is NOT a constant

// read into the array
while (std::cin >> a[k]) {
  if (++k == n) {
    // next element wouldn't fit; replace the array a by
    // a new one of twice the size
    int* b = new int[n*=2];  // get pointer to new array
    for (int i=0; i<k; ++i)  // copy old array to new one
        b[i] = a[i];
    delete[] a;              // delete old array
    a = b;                   // let a point to new array
  }
}

...

delete[] a; // don't forget to delete after use
```

New Range - How Much Larger?

Dynamic Storage Solution

- "Much" larger?
  - Pro: ranges less often full → copy less often
  - Con: larger memory consumption

- Important: Larger by a factor, not by a constant…
  - ```
    length_n = length_o * 2
    length_n = length_o + 2
    ```

(From: Script Exercise 158.a)
    ```

Dynamic Storage Solution

- Larger by: a) factor 2 b) constant 2

elements	Case a)	Case b)
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Larger by: a) factor 2 b) constant 2

elements	Case a)	Case b)
2	2	2
3	4	4
4	4	4
5	8	6
6	8	6
7	8	8
8	8	8
9	16	10
10	16	10
11	16	12
12	16	12
13	16	14
14	16	14
15	16	16
16	16	16
17	32	18

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Larger by: a) factor 2 b) constant 2

elements	Case a)	Case b)
2	2	2
3	4	4
4	4	4
5	8	6
6	8	6
7	8	8
8	8	8
9	16	10
10	16	10
11	16	12
12	16	12
13	16	14
14	16	14
15	16	16
16	16	16
17	32	18

Case a):

Significantly fewer resizings.

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Larger by: a) factor 2 b) constant 2

elements	Case a)	Case b)
2	2	2
3	4	4
4	4	4
5	8	6
6	8	6
7	8	8
8	8	8
9	16	10
10	16	10
11	16	12
12	16	12
13	16	14
14	16	14
15	16	16
16	16	16
17	32	18

arbitr. chosen

Each resizing means:

Copy WHOLE array.

Case a):

Significantly fewer resizings.

(From: Script Exercise 158.a)

Dynamic Storage Solution

- Larger by: a) factor 2 b) c

elements	Case a)	
2	2	
3	4	
4		
5		
		8
		8
		10
	16	10
	16	12
	16	12
13	16	14
14	16	14
15	16	16
16	16	16
17	32	18

hosen

Each resizing means:

Copy WHOLE array

icantly fewer resizings.

Factor 2 is an arbitrary, but good choice.

(From: Script Exercise 158.a)

Vectors

Vectors

- Vectors can grow!

```
std::vector<int> vec (2,0); // 0 0
vec.push_back(7);           // 0 0 7
vec.push_back(2);           // 0 0 7 2
vec.push_back(6);           // 0 0 7 2 6
```

- This works as discussed before!

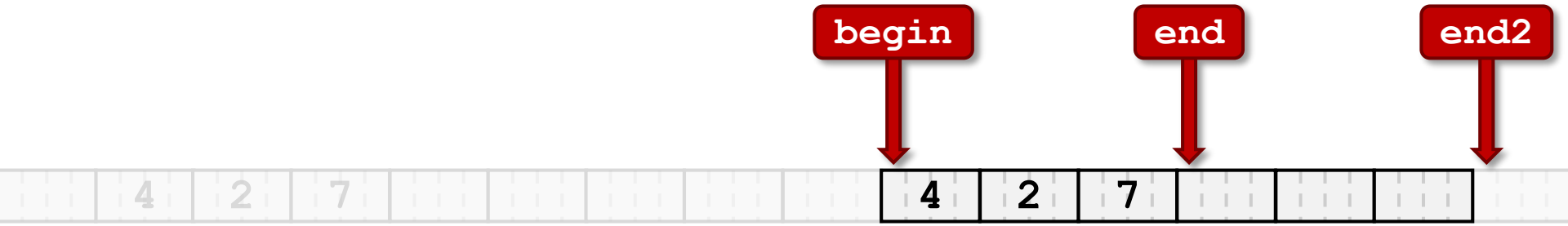
Dynamic Storage in Vectors

- Vectors store 3 pointers:

  `begin`: begin of memory

  `end`: end of *user-accessible* part

  `end2`: end of allocated part



Dynamic Storage in Vectors

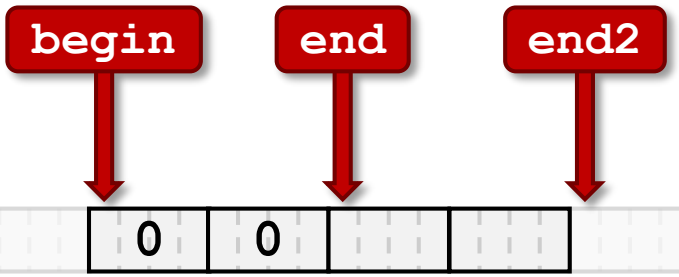
- Example:

```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```

Dynamic Storage in Vectors

- Example:

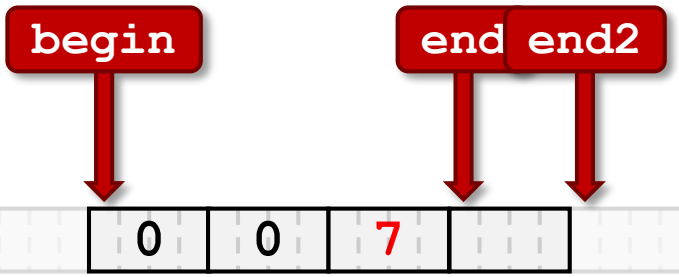
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

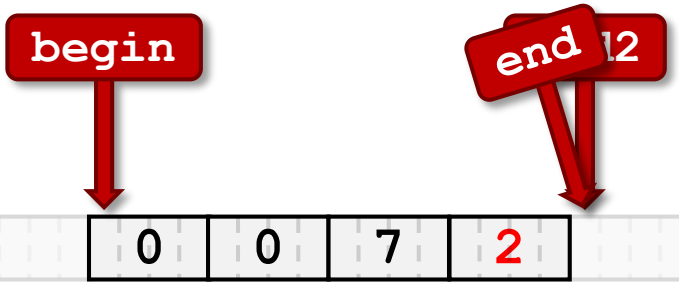
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

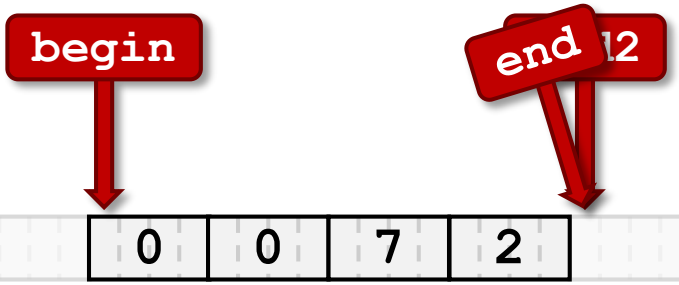
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



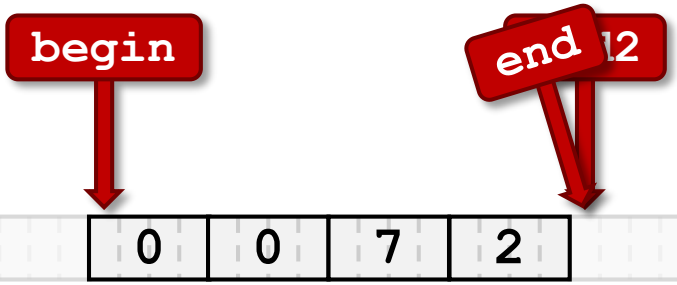
Dynamic Storage in Vectors

- Example:

```
ifmp::vector<i                    // 0 0
vec.push_ba                       0 0 7
vec.push_b                        0 0 7 2
vec.push_b                        0 7 2 6
```

Space full

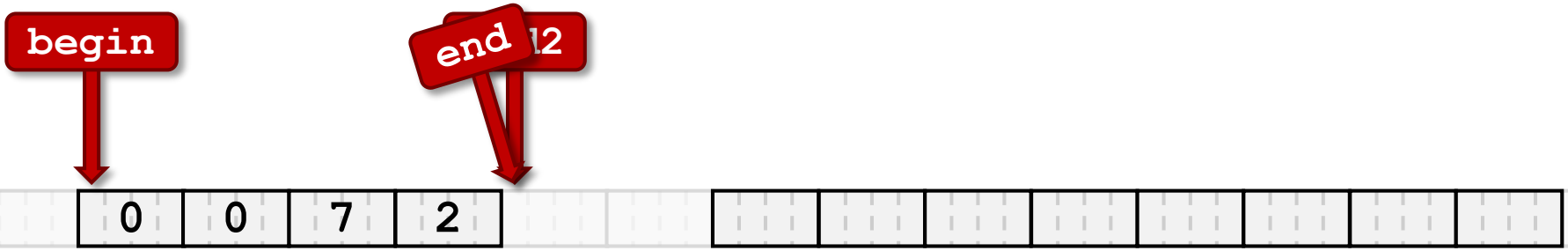
Now:
copy range



Dynamic Storage in Vectors

- Example:

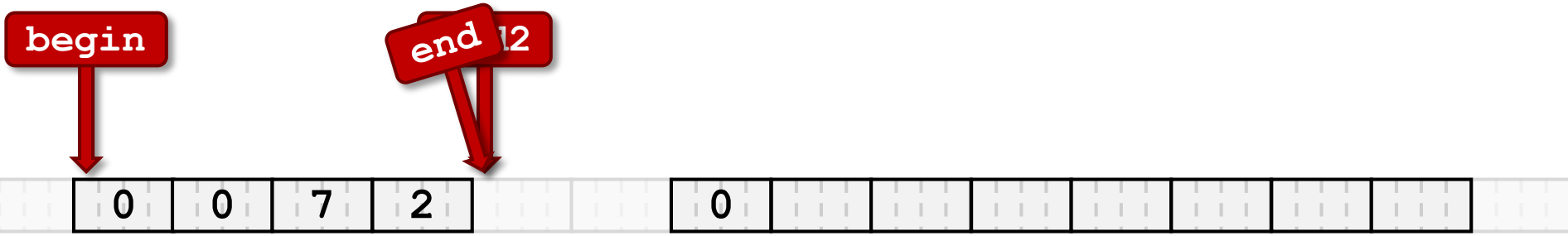
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

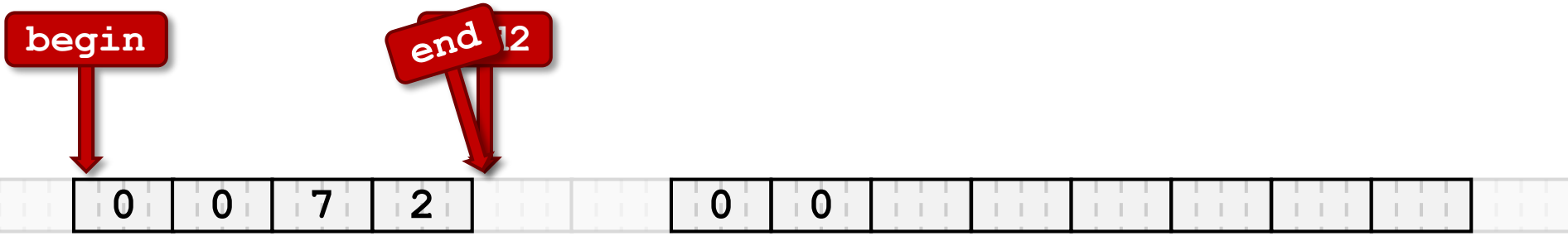
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

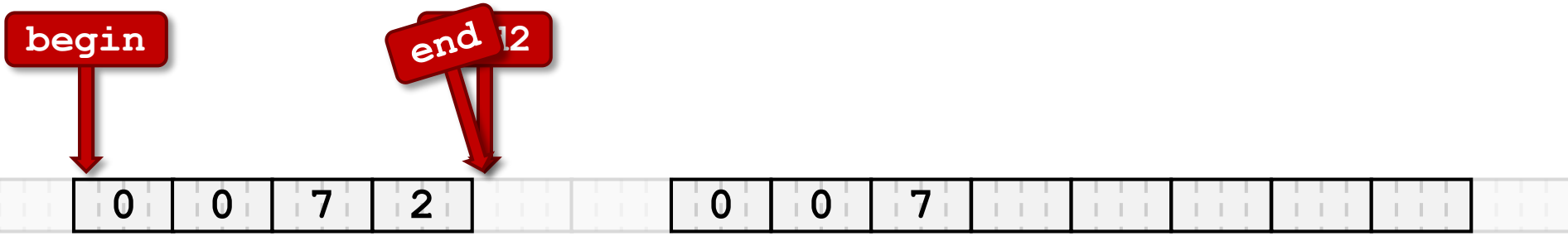
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

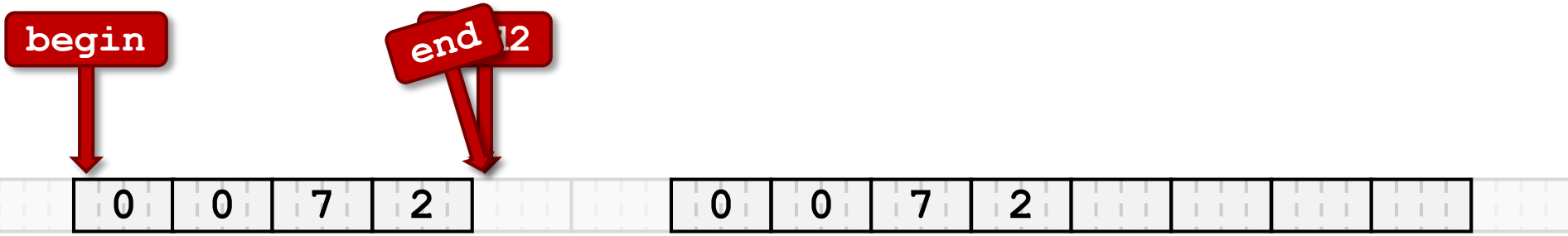
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

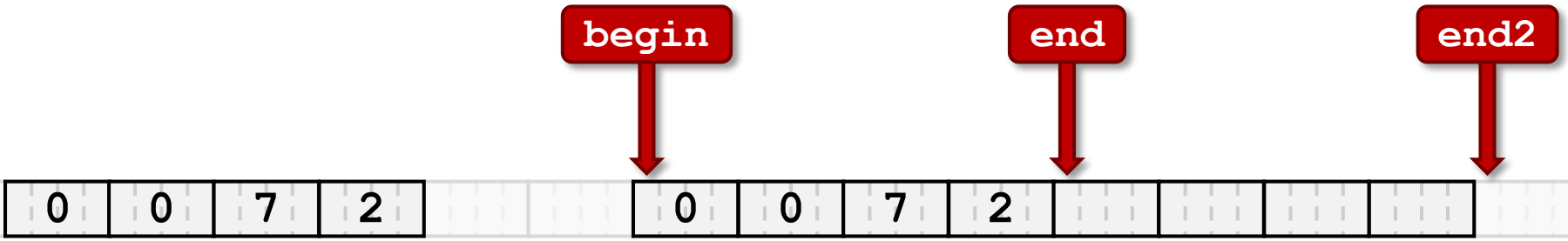
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

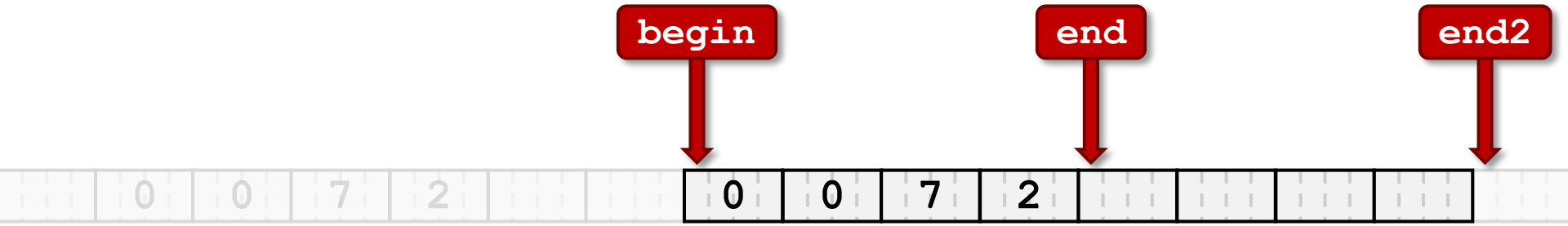
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

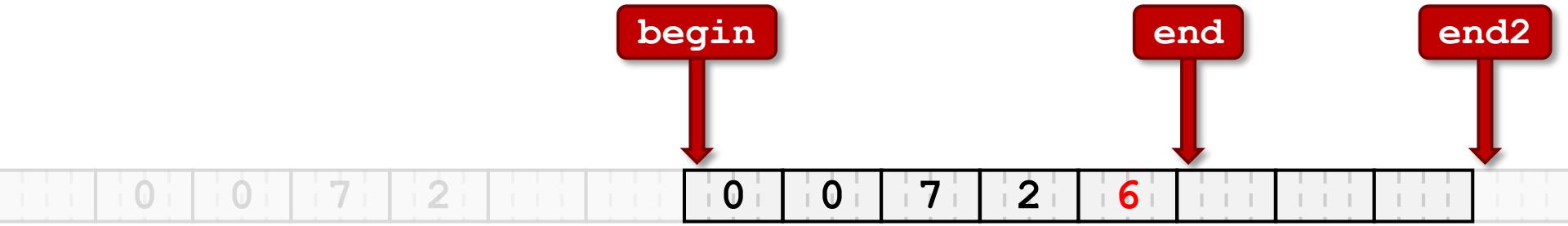
```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Example:

```
ifmp::vector<int> vec (2,0); // 0 0
vec.push_back(7);            // 0 0 7
vec.push_back(2);            // 0 0 7 2
vec.push_back(6);            // 0 0 7 2 6
```



Dynamic Storage in Vectors

- Exercise sheet 12: implement your own vector type.

- Important:
  - **In constructor** — Set initial range
  - **In copy-constructor** — Don't copy just pointers; i.e. copy the ranges behind them
  - **In `operator=`** — Like copy-constructor, in addition:
    i) prevent self-assignments
    ii) don't forget to delete old range