CS 106X
Lecture 17: Implementing Vector

Friday, February 17, 2017

Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Section 12.1

Hermit Crab
(*Pagurus bernhardus*)

Today's Topics

- Logistics
  - Practice Midterms posted
  - Midterm will cover up to and including Linked Lists
  - Review session: Monday 7:30-8:30pm

- Homework 5: Linked Lists / Heap
- More information on **`delete`**
- Destructors
- Implementing the Vector class
  - Header File
  - Implementation
  - focus on expand()

Why do we care about delete?

```
const int INIT_CAPACITY = 1000000;

class Demo {
public:
    Demo(); // constructor
    string at(int i);
private:
    string *bigArray;
};

Demo::Demo()
{
    bigArray = new string[INIT_CAPACITY];
    for (int i=0;i<INIT_CAPACITY;i++) {
        bigArray[i] = "Lalalalalalalalala!";
    }
}

string Demo::at(int i)
{
    return bigArray[i];
}
```

Assignment 5: Linked Lists and Heaps

For parts A and C of the assignment, you will be implementing a data structure called a "priority queue" which allows you to store keys and values based on the "priority" of the key. You will be modeling a hospital emergency room: patients with a higher priority are attended to first, even if they arrive after patients with lower priority:

For example, if the following patients arrive at the hospital in this order:

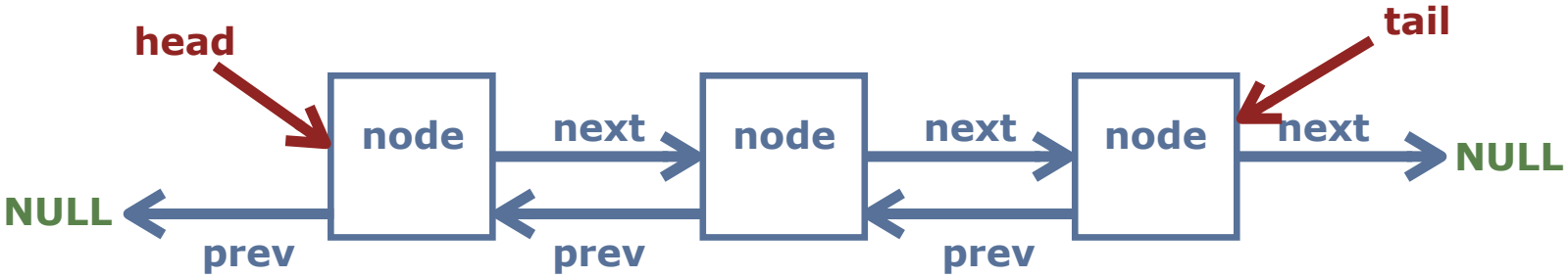
- "Dolores" with priority 5
- "Bernard" with priority 4
- "Arnold" with priority 8
- "William" with priority 5
- "Teddy" with priority 5
- "Ford" with priority 2

Then if you were to dequeue the patients to process them, they would come out in this order: Ford, Bernard, Dolores, William, Teddy, Arnold.

Doubly-linked lists

For part B of the assignment, you will implement "doubly-linked lists":

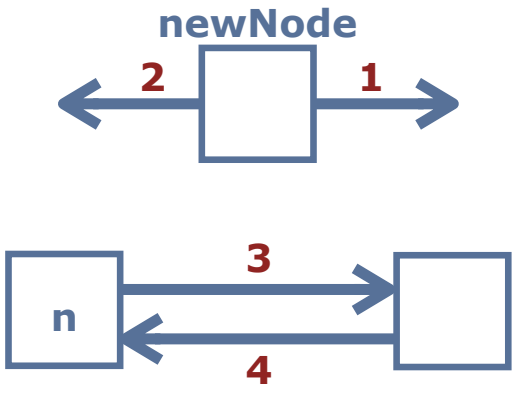
Doubly-linked lists: both next and prev pointers:



Given a node, you can now traverse both forward and backwards, too. *However,* it means that you have to update more pointers when making changes to the list.

Doubly-linked lists

ALWAYS draw this case!!
(numbering is good!)

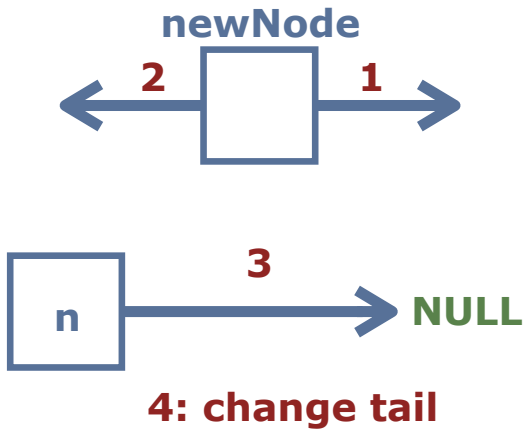



```
void insertNodeAfter(Node *n,Node *newNode){
      if (head == NULL) { // no list yet
            head = newNode;
            newNode->next = NULL;
            newNode->prev = NULL;
      }
      else {
            newNode->next = n->next; // 1
            newNode->prev = n; // 2
            n->next = newNode; // 3
            newNode->next->prev = newNode; // 4
}
```

Not quite complete!

Doubly-linked lists

Don't forget the special cases!




```
void insertNodeAfter(Node *n,Node *newNode){
      if (head == NULL) { // no list yet
            head = newNode;
            newNode->next = NULL;
            newNode->prev = NULL;
      }
      else {
            newNode->next = n->next; // 1
            newNode->prev = n; // 2
            n->next = newNode; // 3
            if (newNode->next == NULL) {
                  tail = newNode; // if using tail
            }
            else {
                  newNode->next->prev = newNode; // 4
            }
      }
}
```

More on part C for HW 5 later!

Why do we care about delete?

```
const int INIT_CAPACITY = 1000000;

class Demo {
public:
    Demo(); // constructor
    string at(int i);
private:
    string *bigArray;
};

Demo::Demo()
{
    bigArray = new string[INIT_CAPACITY];
    for (int i=0;i<INIT_CAPACITY;i++) {
        bigArray[i] = "Lalalalalalalalala!";
    }
}

string Demo::at(int i)
{
    return bigArray[i];
}
```

This is a lot of strings...

1MB array * (20 chars + ~30 bytes of overhead for each string) = 50MB per class instance

Why do we care about delete?

```
const int INIT_CAPACITY = 1000000;

class Demo {
public:
    Demo(); // constructor
    string at(int i);
private:
    string *bigArray;
};

Demo::Demo()
{
    bigArray = new string[INIT_CAPACITY];
    for (int i=0;i<INIT_CAPACITY;i++) {
        bigArray[i] = "Lalalalalalalalala!";
    }
}

string Demo::at(int i)
{
    return bigArray[i];
}
```

```
int main()
{
    for (int i=0;i<10000;i++){
        Demo demo;
        cout << i << ": "
             << demo.at(1234)
             << endl;
    }
    return 0;
}
```

Why do we care about delete?

```
const int INIT_CAPACITY = 1000000;

class Demo {
public:
    Demo(); // constructor
    string at(int i);
private:
    string *bigArray;
};

Demo::Demo()
{
    bigArray = new string[INIT_CAPACITY];
    for (int i=0;i<INIT_CAPACITY;i++) {
        bigArray[i] = "Lalalalalalalalala!";
    }
}

string Demo::at(int i)
{
    return bigArray[i];
}
```

Let's do this 10000 times...after about 40 or so, we will use 2GB of memory...

```
int main()
{
    for (int i=0;i<10000;i++){
        Demo demo;
        cout << i << ": "
             << demo.at(1234)
             << endl;
    }
    return 0;
}
```

Why do we care about delete?

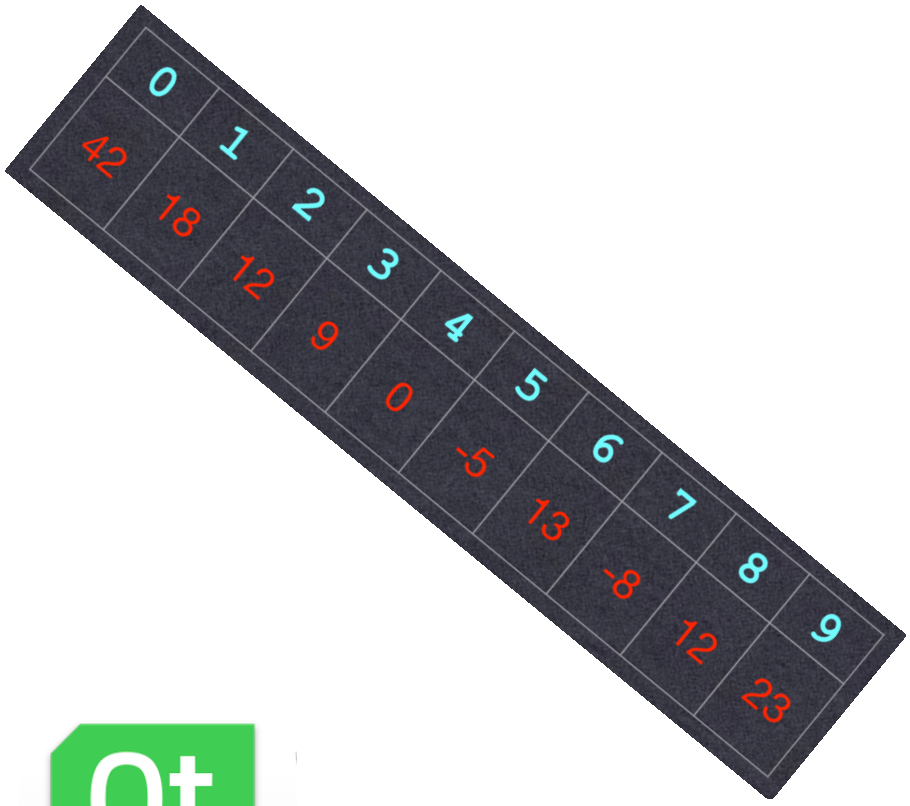
```
const int INIT_CAPACITY = 1000000;

class Demo {
public:
    Demo(); // constructor
    string at(int i);
private:
    string *bigArray;
};

Demo::Demo()
{
    bigArray = new string[INIT_CAPACITY];
    for (int i=0;i<INIT_CAPACITY;i++) {
        bigArray[i] = "Lalalalalalalalala!";
    }
}

string Demo::at(int i)
{
    return bigArray[i];
}
```

Let's see what happens!

```
int main()
{
    for (int i=0;i<10000;i++){
        Demo demo;
        cout << i << ": "
             << demo.at(1234)
             << endl;
    }
    return 0;
}
```

The VectorInt Class: Implementation

- In order to demonstrate how useful (and necessary) dynamic memory is, let's implement a Vector that has the following properties:
  - It can hold **`int`**s (unfortunately, it is beyond the scope of this class to create a Vector that can hold *any* type)
  - It has useful Vector functions: **`add()`, `insert()`, `get()`, `remove()`, `isEmpty()`, `size()`, `<<` overload**
  - We can add as many elements as we would like
  - It cleans up its own memory



Dynamic Memory Allocation: your responsibilities

- Back to Stanford Word.
- The problem we had initially was that Stanford Word can't just pick an array size for the number of pages, because it doesn't know how many pages you want to write.
- But, using a dynamic array, Stanford Word can initially set a low number of pages (say, five), and then ... what can it do?

Expansion Analogy: Hermit Crabs

- Hermit Crabs
  - Hermit crabs are interesting animals. The live in scavenged shells that they find on the sea floor. Once in a shell, this is their lifestyle (with a bit of poetic license):
    - Grow a bit until the shell is outgrown.
    1. Find another shell.
    2. Move all their stuff into the other shell.
    3. Leave the old shell on the sea floor.
    4. Update their address with the Hermit Crab Post Office
    5. Update the capacity of their new shell on their web page.

Expansion Analogy: Hermit Crabs

- Dynamic Arrays
  - We can actually model what we want Microsoft Word to do with the array for its document by the hermit crab model.
  - In essence, when we run out of space in our array, we want to allocate a new array that is bigger than our old array so we can store the new data and keep growing. These "growable arrays" follow a five-step expansion that mirrors the hermit crab model (with poetic license).
  - One question: if we are going to expand our array, how much more memory do we ask for?

double the amount! This is the most efficient.

Expansion Analogy: Hermit Crabs

- Dynamic Arrays
  - There are five primary steps to expanding a dynamic array:

1. Create a new array with a new size (normally twice the size)
2. Copy the old array elements to the new array
3. Delete the old array (understanding what happens here is key!)
4. Point the old array variable to the new array (it is a pointer!)
5. Update the capacity variable for the array

- When do we decide to expand an array?
  - When it is full. How do we know it is full? We keep track!

Expansion Analogy: Hermit Crabs

- Dynamic Arrays
  - There are five primary steps to expanding a dynamic array:

1. Create a new array with a new size (normally twice the size)
2. Copy the old array elements to the new array
3. Delete the old array (understanding what happens here is key!)
4. Point the old array variable to the new array (it is a pointer!)
5. Update the capacity variable for the array

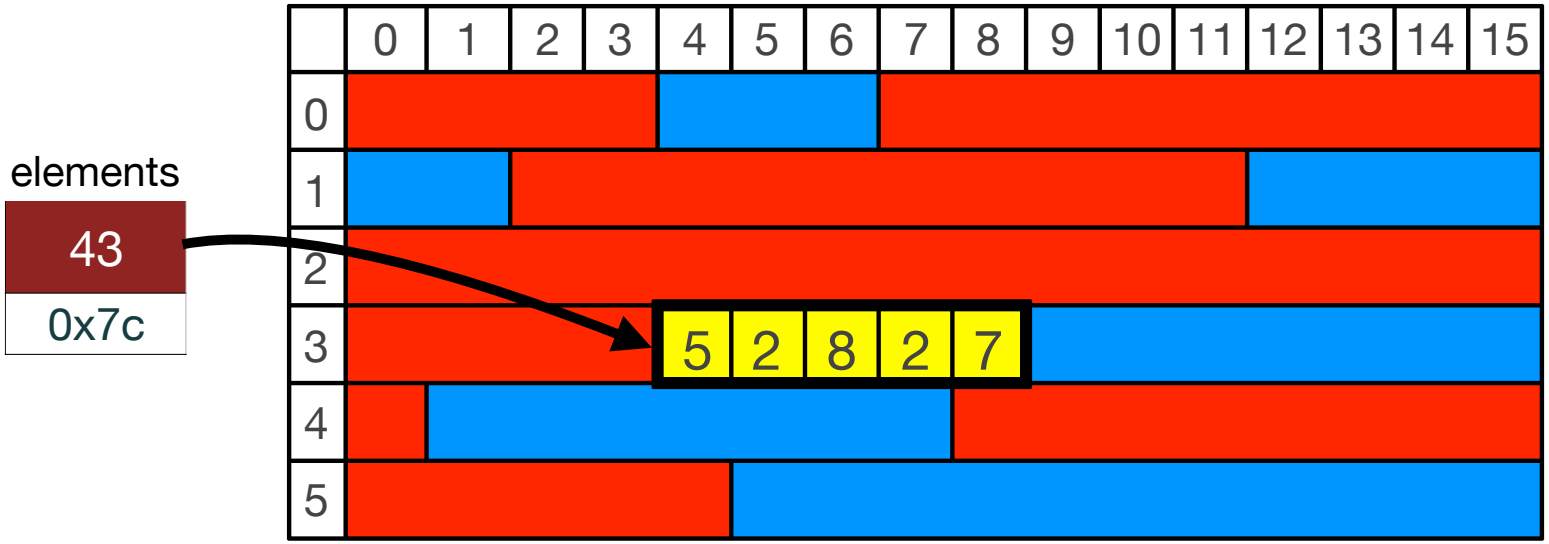
- When do we decide to expand an array?
  - When it is full. How do we know it is full? We keep track!

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

  1.Request space for 10 elements: `int *newElements = new int[capacity * 2];`

What options does the operating system have?

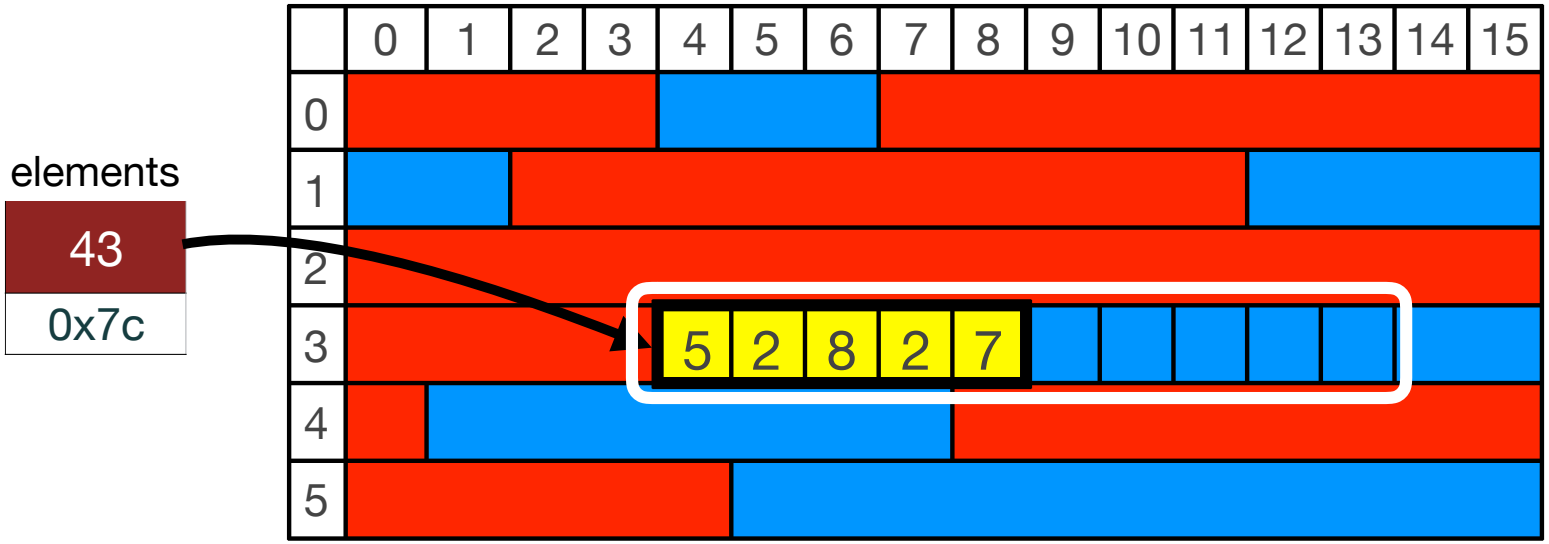


Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

  1.Request space for 10 elements: `int *newElements = new int[capacity * 2];`

What options does the operating system have?



Is this an option?

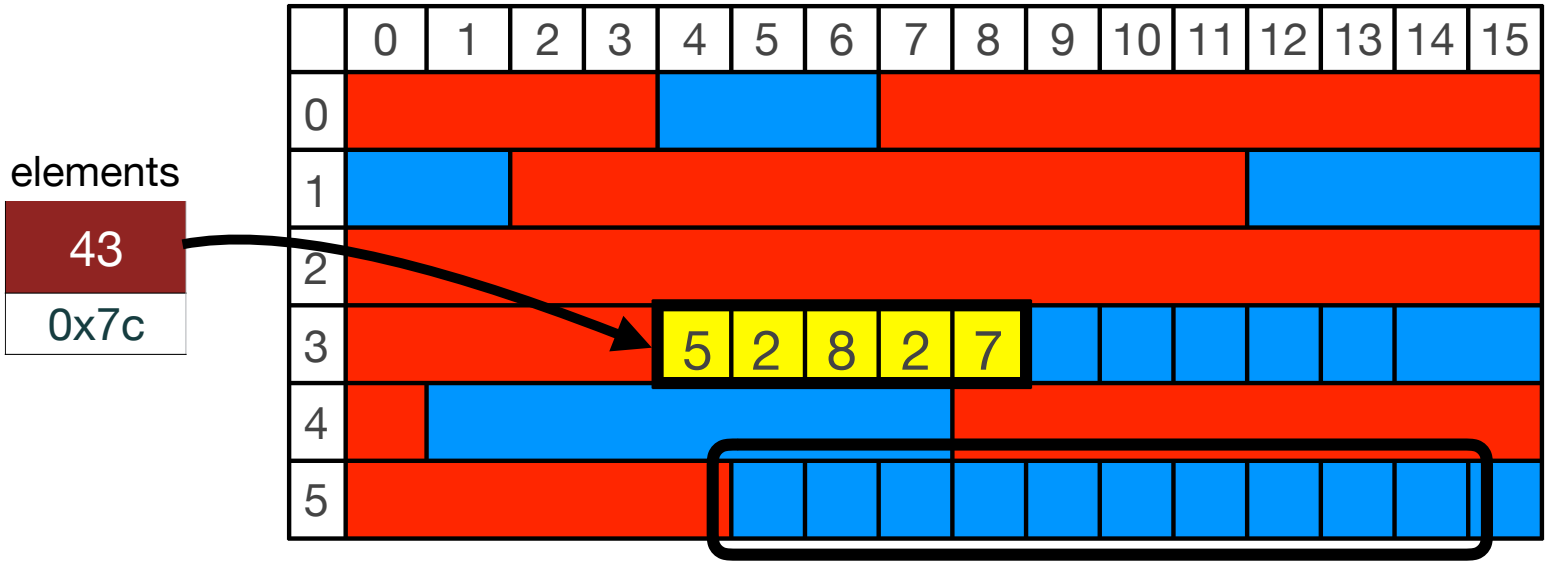
No — you're already using the first five!

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

  1.Request space for 10 elements: `int *newElements = new int[capacity * 2];`

What options does the operating system have?



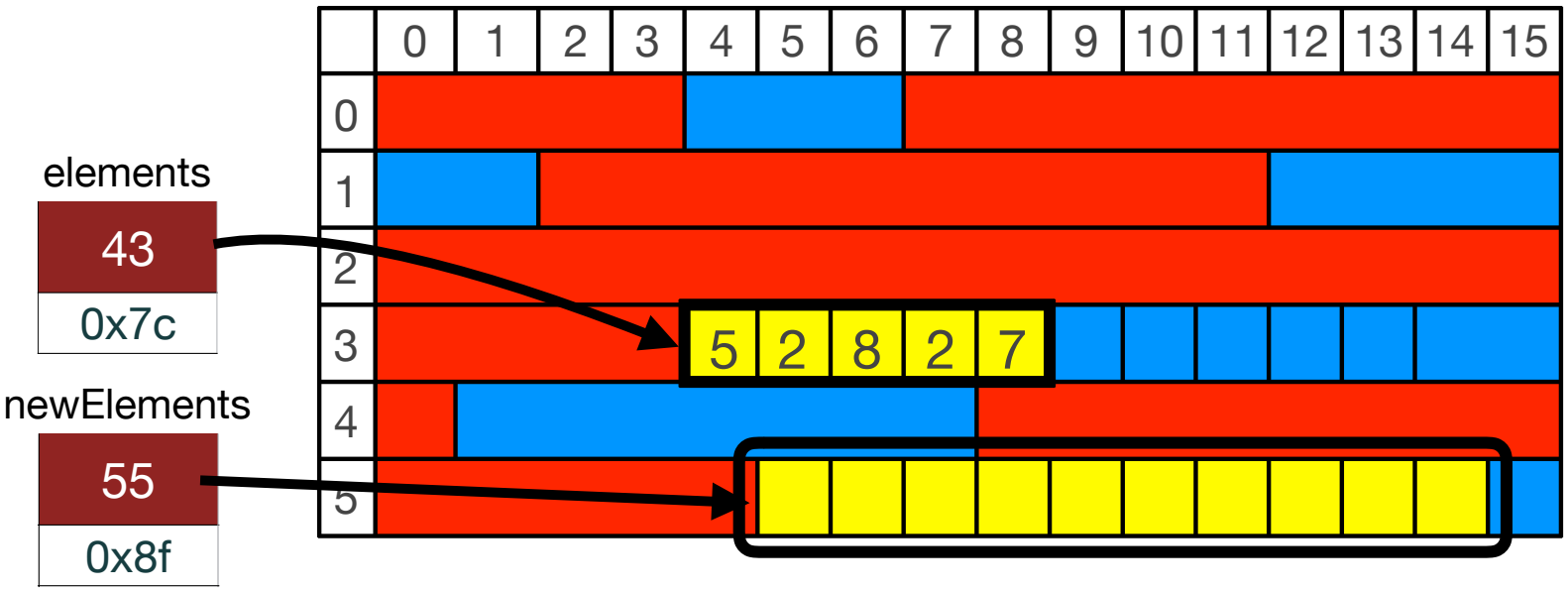
This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

  1.Request space for 10 elements: `int *newElements = new int[capacity * 2];`

What options does the operating system have?



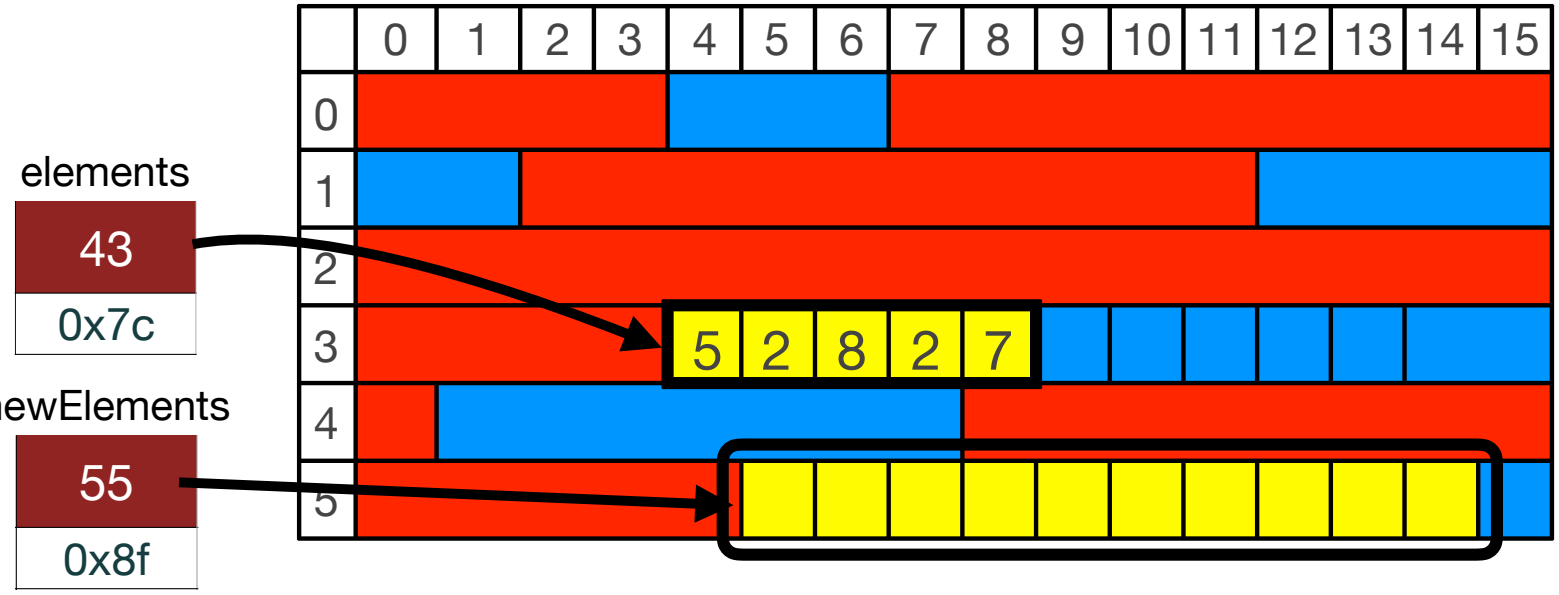
This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

2.Copy:

```
for (int i=0; i < count; i++) {
    newElements[i] = elements[i];
}
```



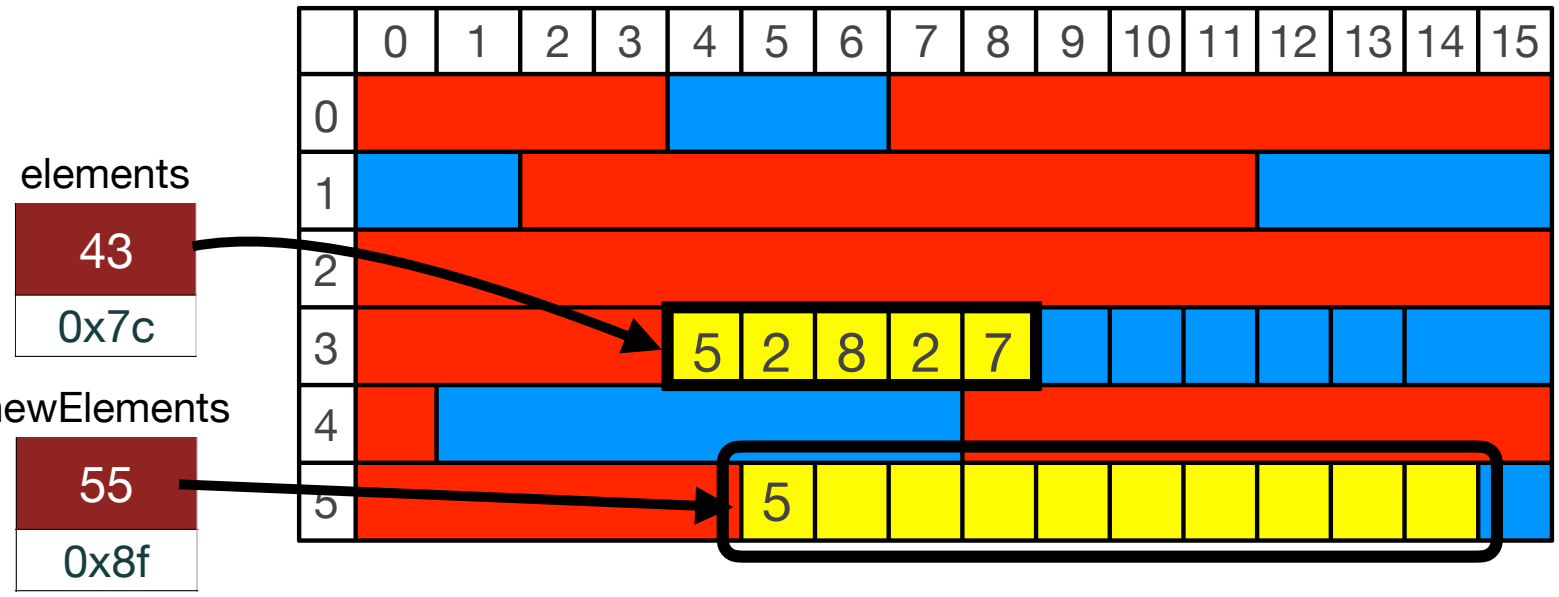
This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

2.Copy:

```
for (int i=0; i < count; i++) {
    newElements[i] = elements[i];
}
```



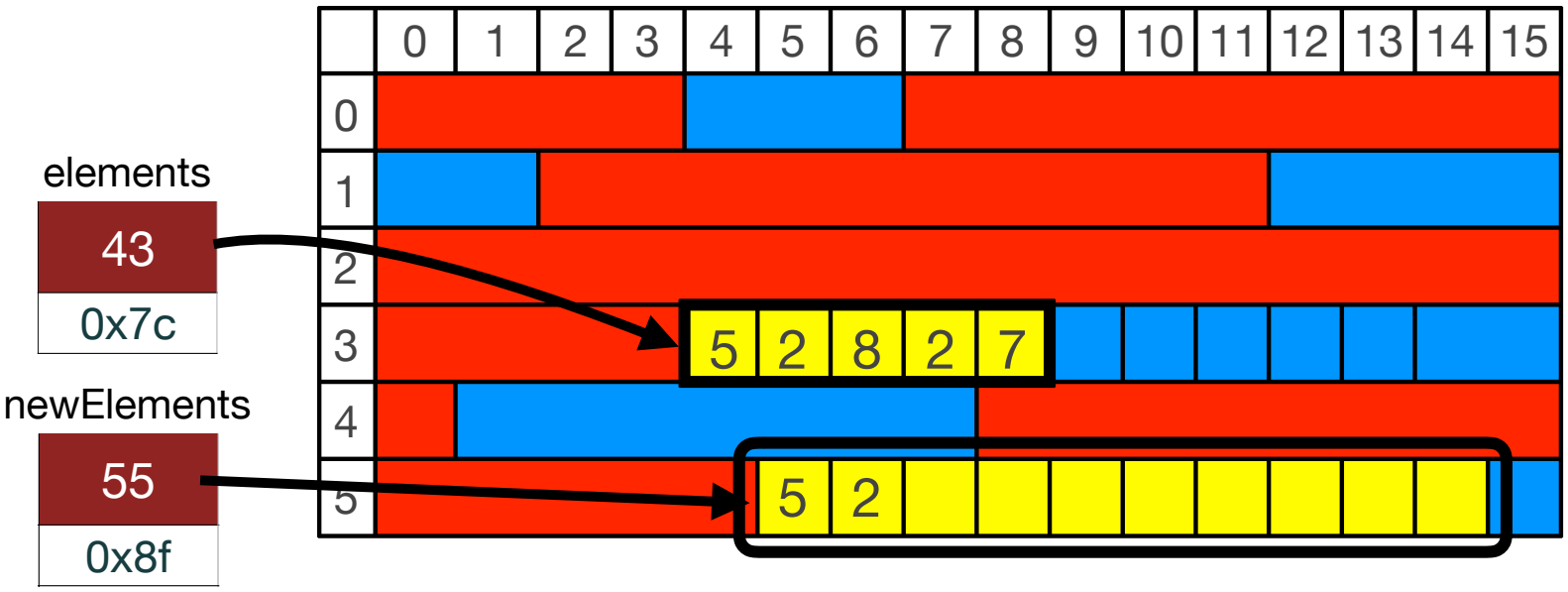
This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

2.Copy:

```
for (int i=0; i < count; i++) {
    newElements[i] = elements[i];
}
```



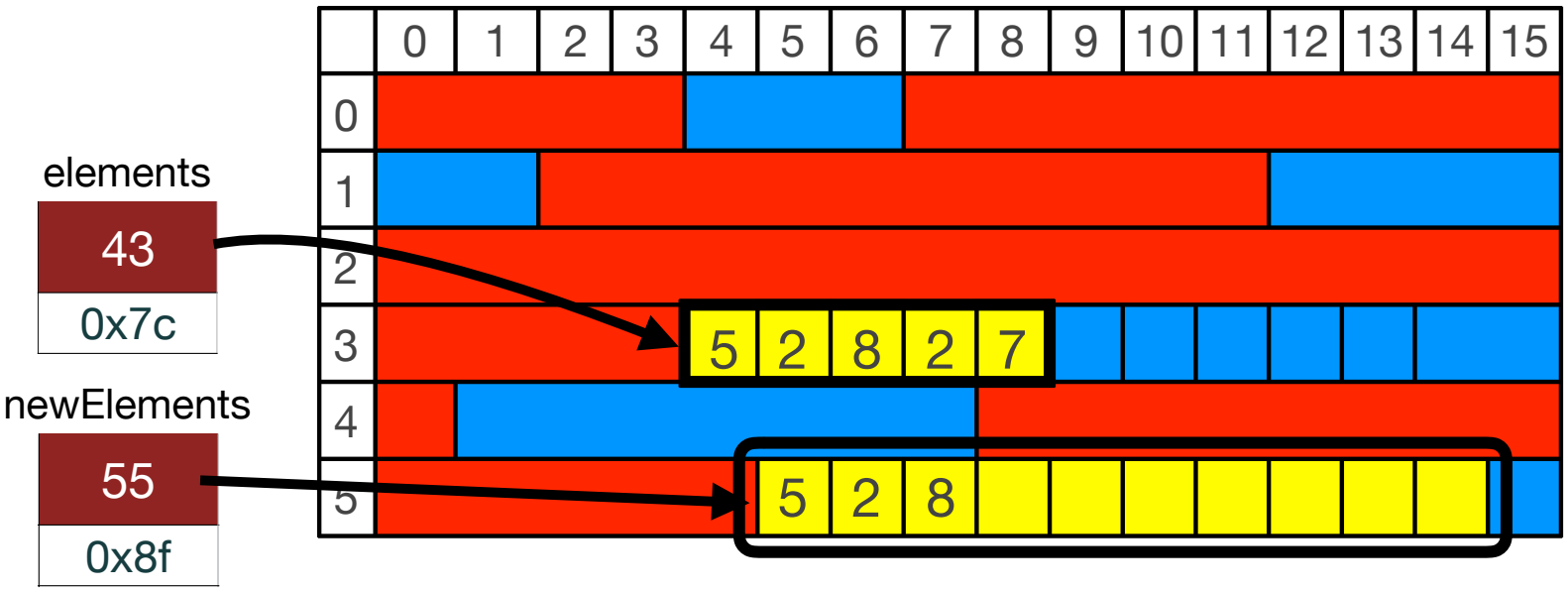
This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

2.Copy:

```
for (int i=0; i < count; i++) {
    newElements[i] = elements[i];
}
```



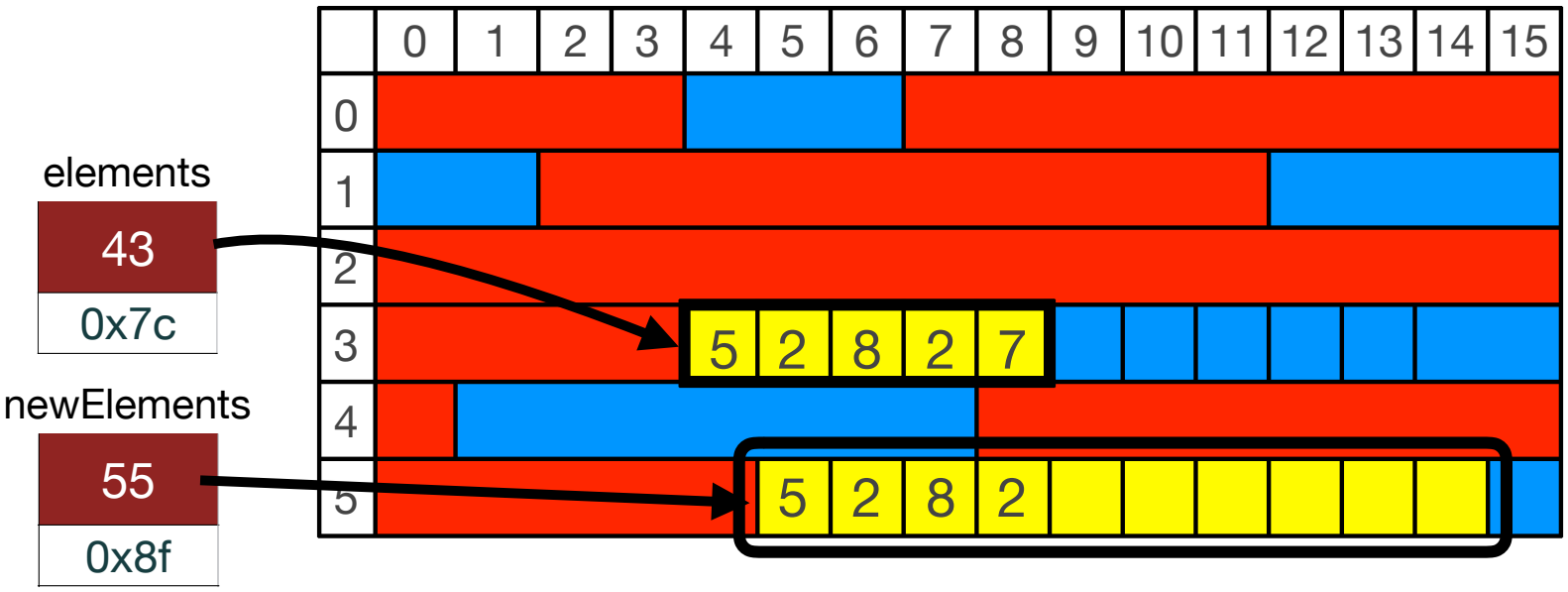
This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

2.Copy:

```
for (int i=0; i < count; i++) {
    newElements[i] = elements[i];
}
```



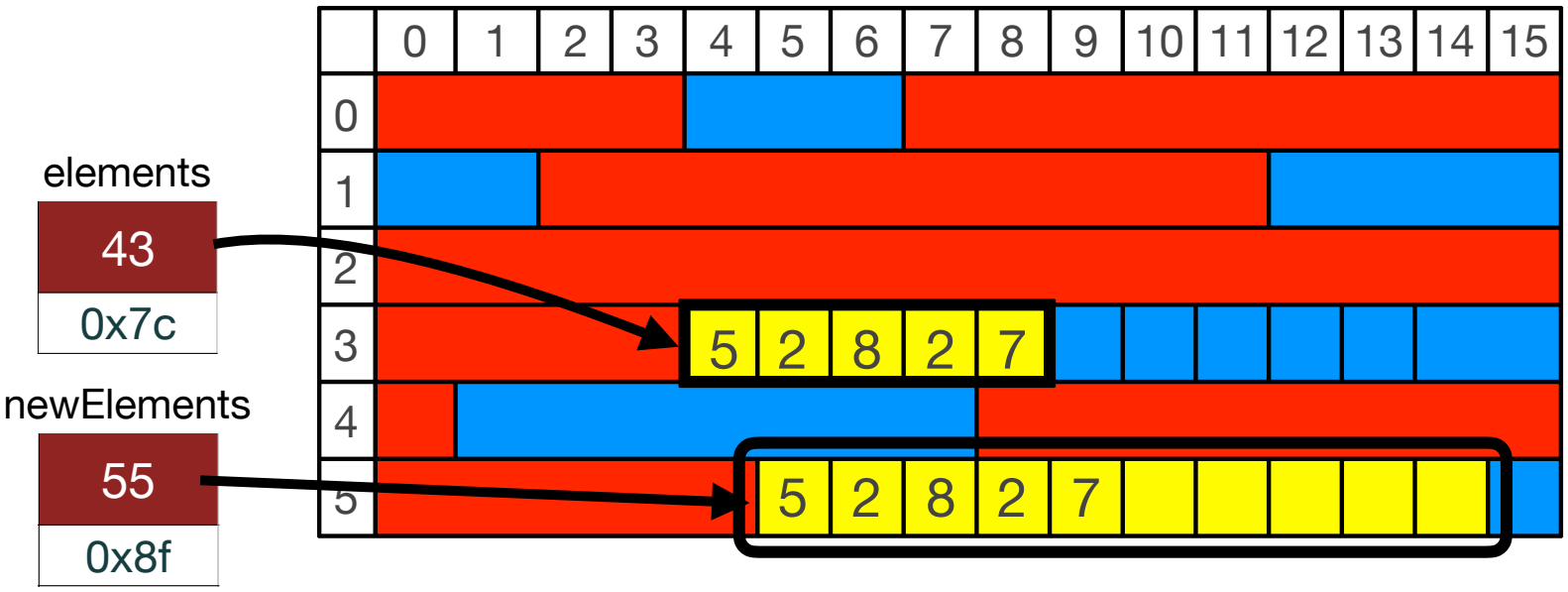
This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

2.Copy:

```
for (int i=0; i < count; i++) {
    newElements[i] = elements[i];
}
```



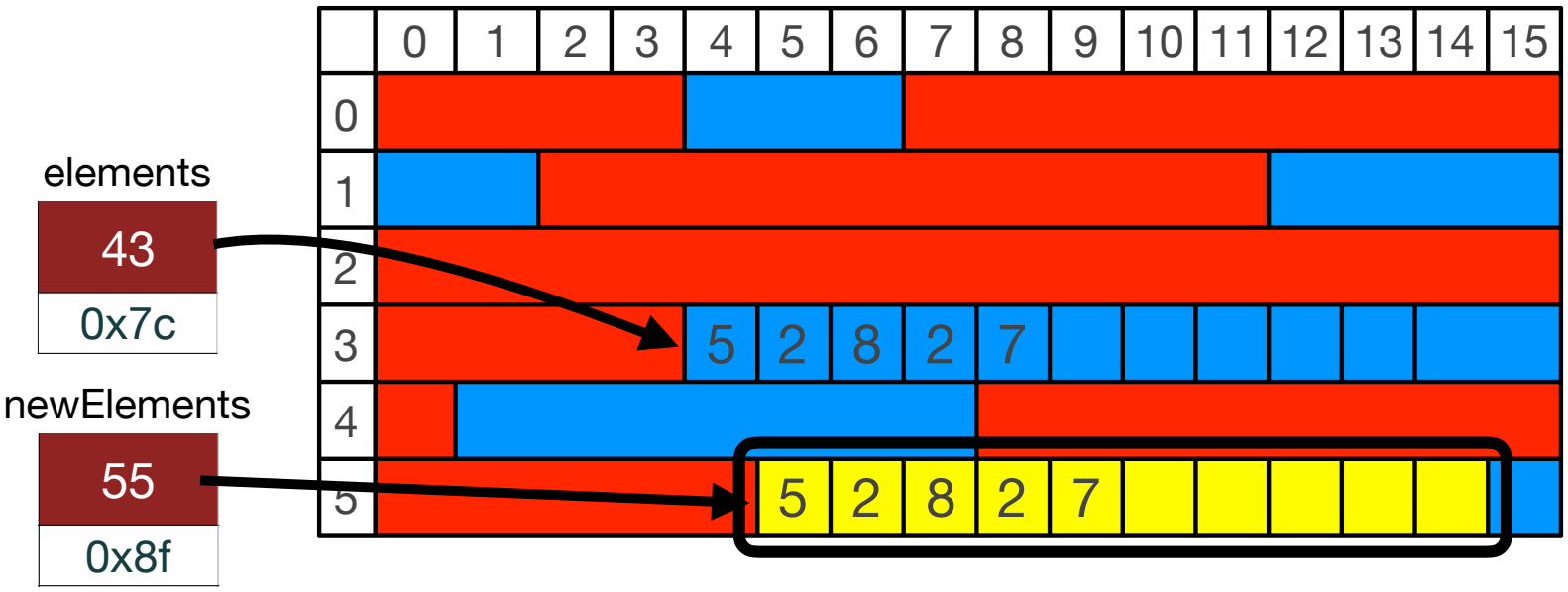
This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:

  3.Delete the original elements: (*you no longer have legitimate access to that memory!*)

```
delete [] elements;
```

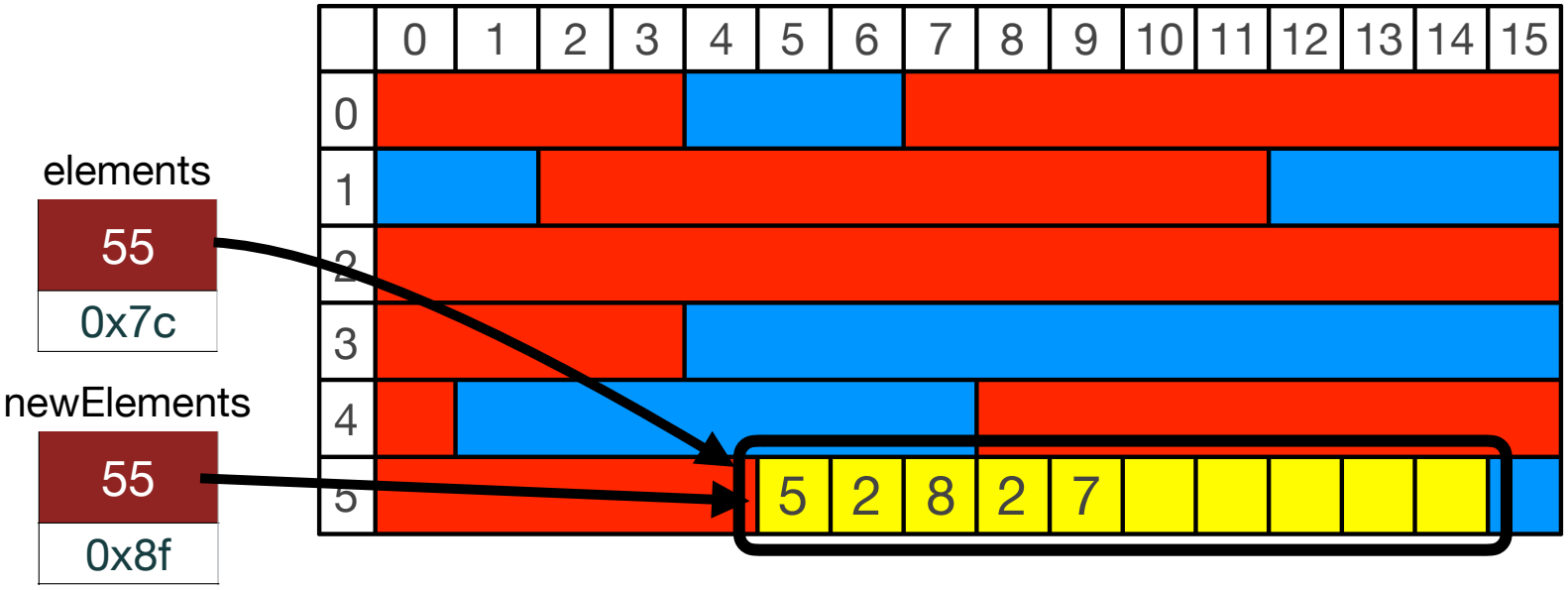


This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:
  4.Assign elements to the new array:

```
elements = newElements;
```

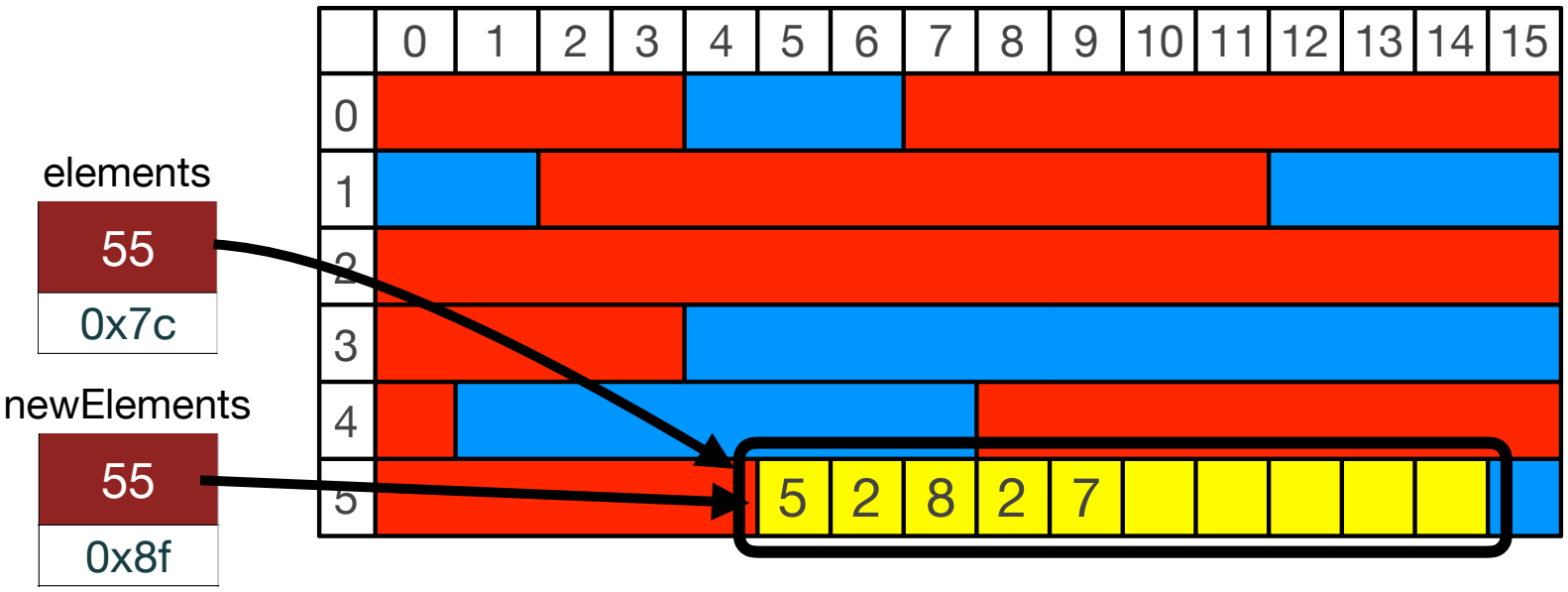


This is the option the OS has to choose.

Vector Expansion: Memory

- Let's say that our Vector's elements pointer points to memory as in the following diagram. capacity = 5, and count = 5 (it is full).
- To expand, we must follow our rules:
  5.(Bookkeeping) update capacity:

```
capacity *= 2;
```



This is the option the OS has to choose.

References and Advanced Reading

- **References:**
  - Dynamic Arrays: https://en.wikipedia.org/wiki/Dynamic_array
  - See the course website for full VectorInt code

- **Advanced Reading:**
  - Vector class with templates: Read textbook, Section 14.4