CS 106B
Lecture 17: Implementing Vector

Wednesday, November 2, 2016

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Section 12.1

Hermit Crab
(*Pagurus bernhardus*)

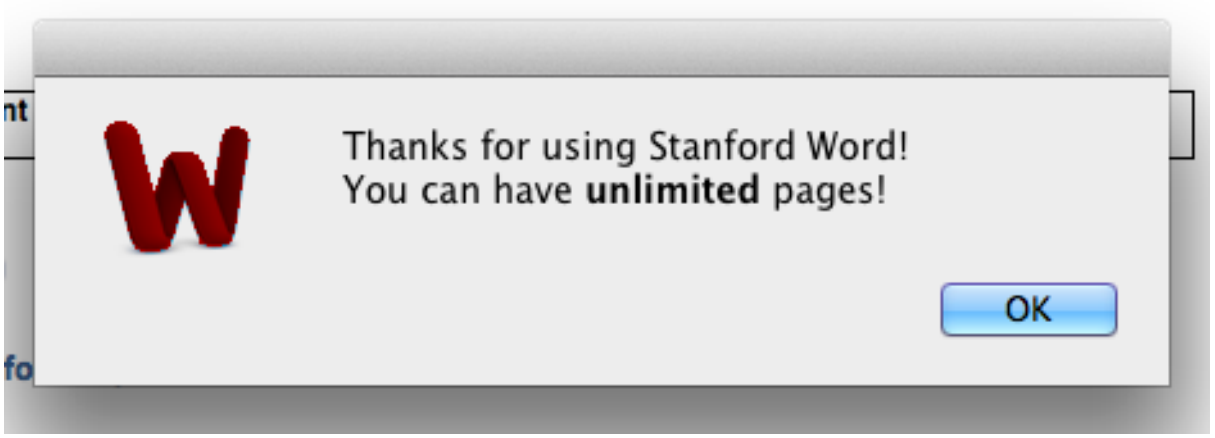
Today's Topics

- Logistics

- More information on **`delete`**
- Destructors
- Implementing the Vector class
  - Header File
  - Implementation
  - focus on expand()

Next Time: Building a Vector class with arrays

- In the next lecture, we will discuss how the Vector is built, using dynamic memory.
- We will need to keep track of all the details ourselves:
  - How much space we have allocated for the Vector
  - How many items are in the Vector
  - How to add / remove / insert into the Vector
  - How to *expand* the Vector



Dynamic Memory Allocation: delete

- **`delete`** is sometimes confusing. Take a look at the following function:

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

Dynamic Memory Allocation: delete

- **`delete`** is sometimes confusing. Take a look at the following function:

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

- First: notice that we delete multiple, and then use it again!
- Is that allowed??

Dynamic Memory Allocation: delete

- **`delete`** is sometimes confusing. Take a look at the following function:

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

- First: notice that we delete multiple, and then use it again!
- Is that allowed??
  - It is! **`delete`** *does not delete any variables! Instead, it follows the pointer and returns the memory to the OS!*
  - However, you are not allowed to use the memory after you have **`delete`**d it.
  - This does not preclude you from re-using the pointer itself.

Dynamic Memory Allocation: delete

- What does this print out, by the way for an **`origArray = {1, 5, 7}`**?

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

```
void printArray(int *array,
                int length) {
    cout << "[";
    for (int i=0; i < length; i++) {
        cout << array[i];
        if (i < length-1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}
```

Output:

```
[2, 10, 14]
[2, 3, 10, 15, 7, 21]
```

Dynamic Memory Allocation: your responsibilities

- With great power comes great responsibility
- You have a responsibility when using dynamic memory allocation to **`delete`** anything you have requested via **`new`**.
- This is the contract you make with the operating system: if you're done with the memory, you should return it. The OS will take it back when your program ends, but this wastes memory, and this is called a "memory leak."

```
const int INIT_CAPACITY = 10000000;

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;
}
```

This program crashed my entire computer when I ran it. Why?

Dynamic Memory Allocation: your responsibilities

- This program crashed my entire computer when I ran it. Why?
- We're allocating a *ton* of memory, and not deleting it!
- We can fix it by adding a "destructor" -- when the class instance goes out of scope, the destructor is called, cleaning up the memory for us.

```
const int INIT_CAPACITY = 10000000;

class Demo {
public:
    Demo(); // constructor
    ~Demo(); // destructor
    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!";
    }
}
```

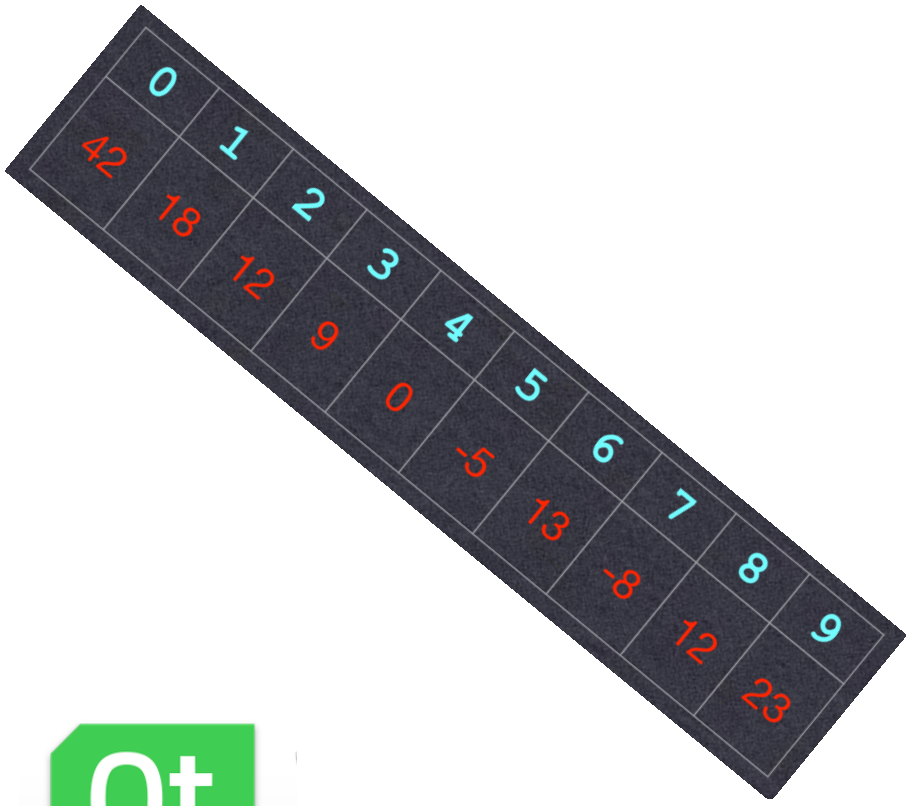
```
Demo::~Demo() {
    delete[] big_array;
}

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;
}
```

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

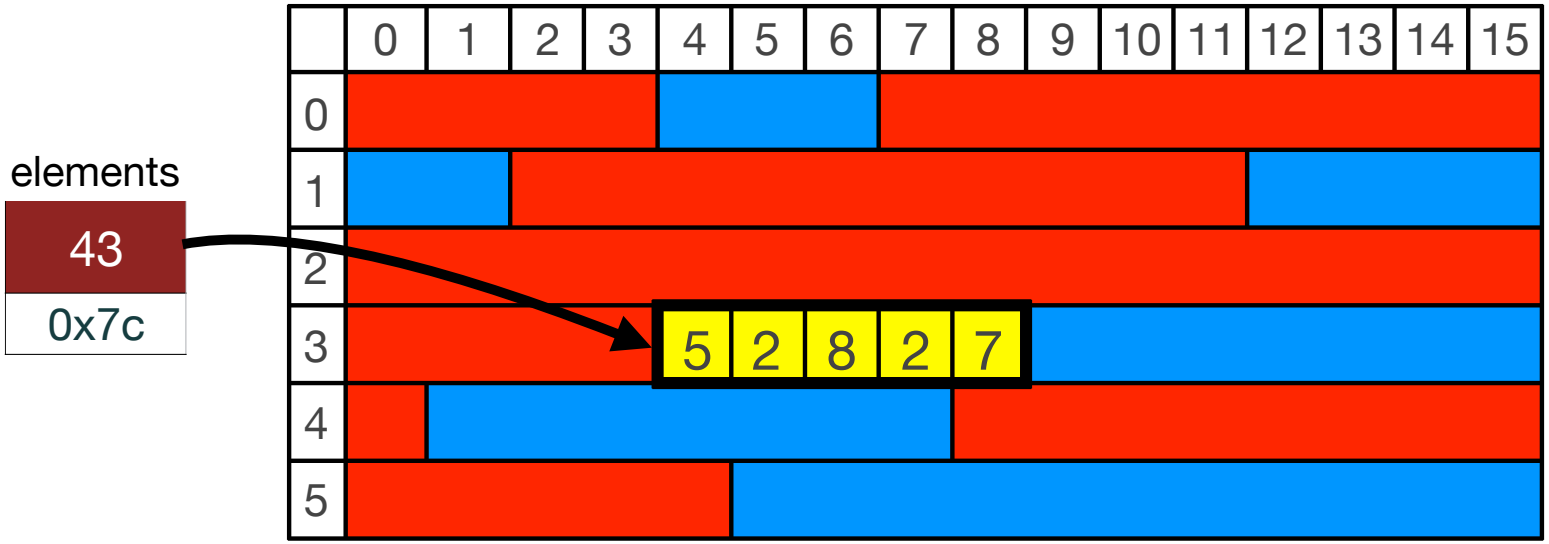
- When do we decide to expand an array?
  - When it is full. How do we know it is full? We keep track!

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?

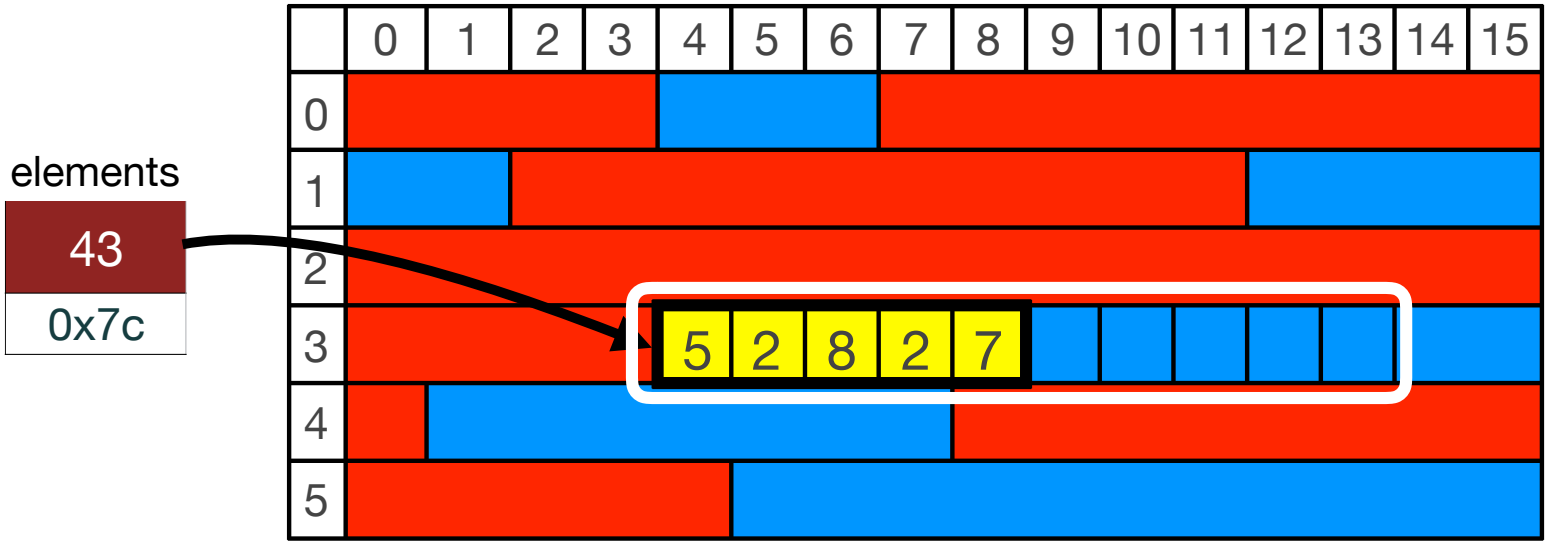


Vector Expansion: Memory

- 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?

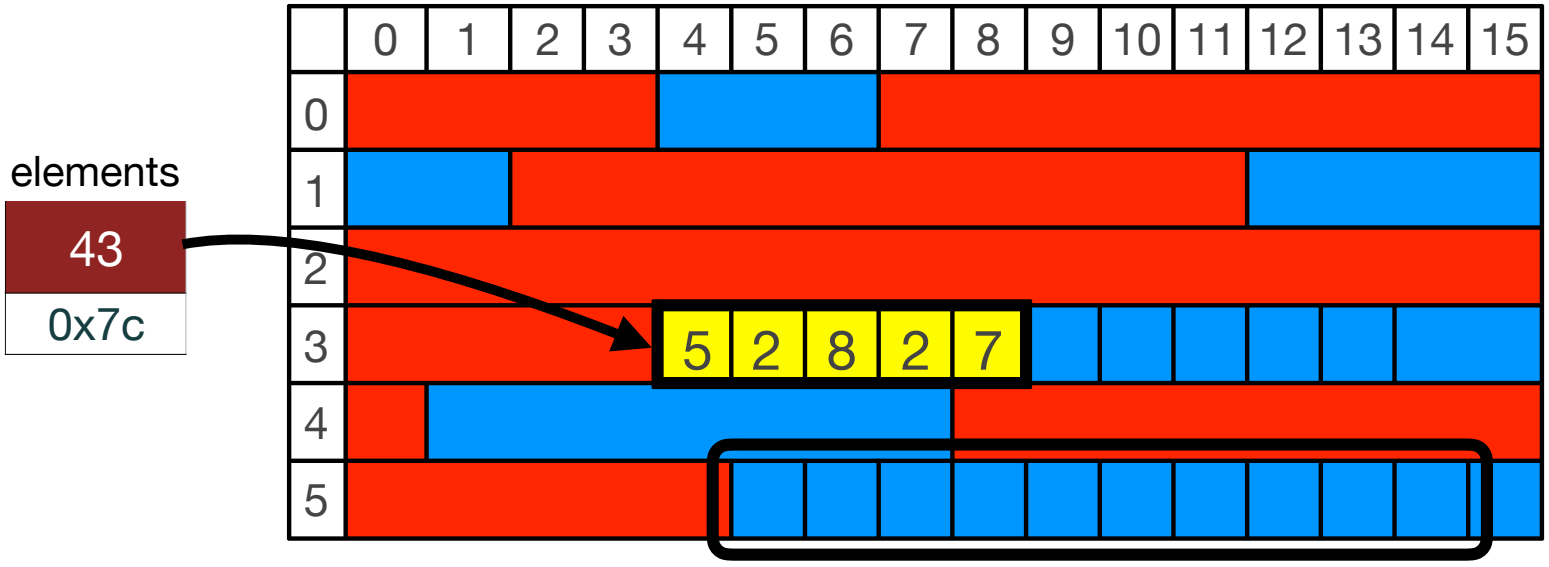
No — you're already using the first five!

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?



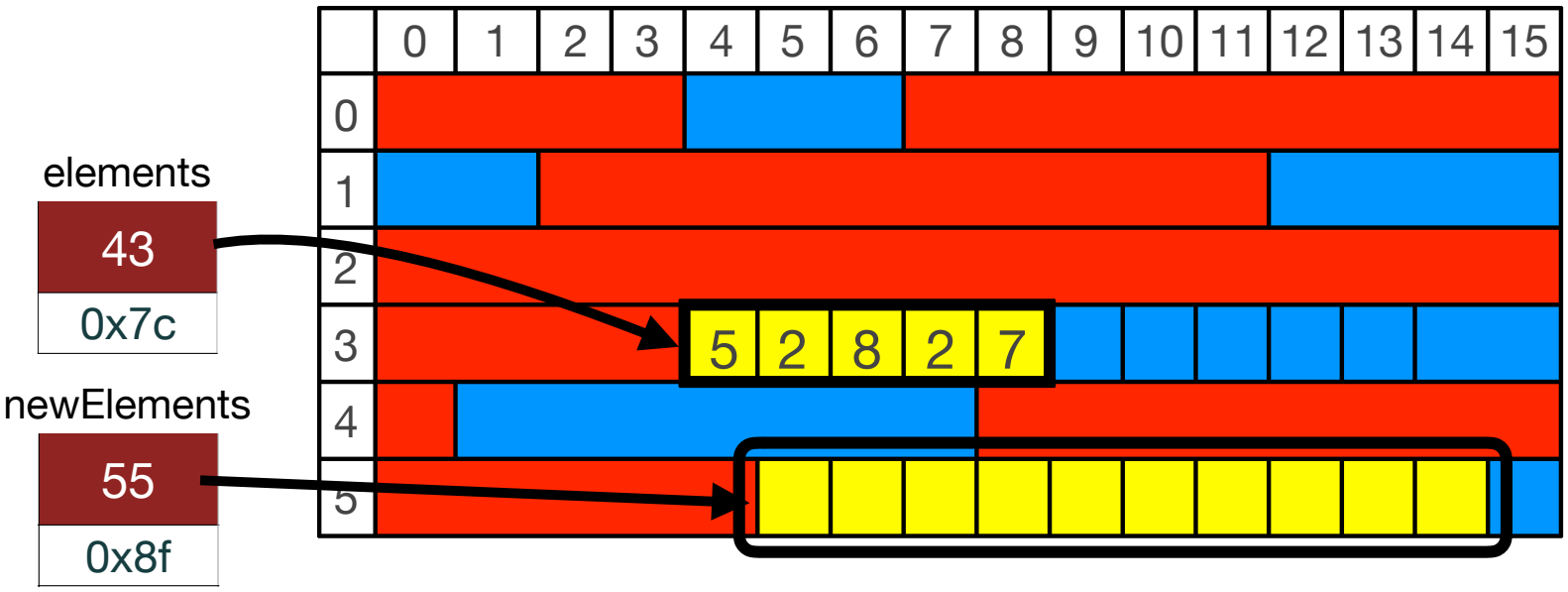
This is the option the OS has to choose.

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?



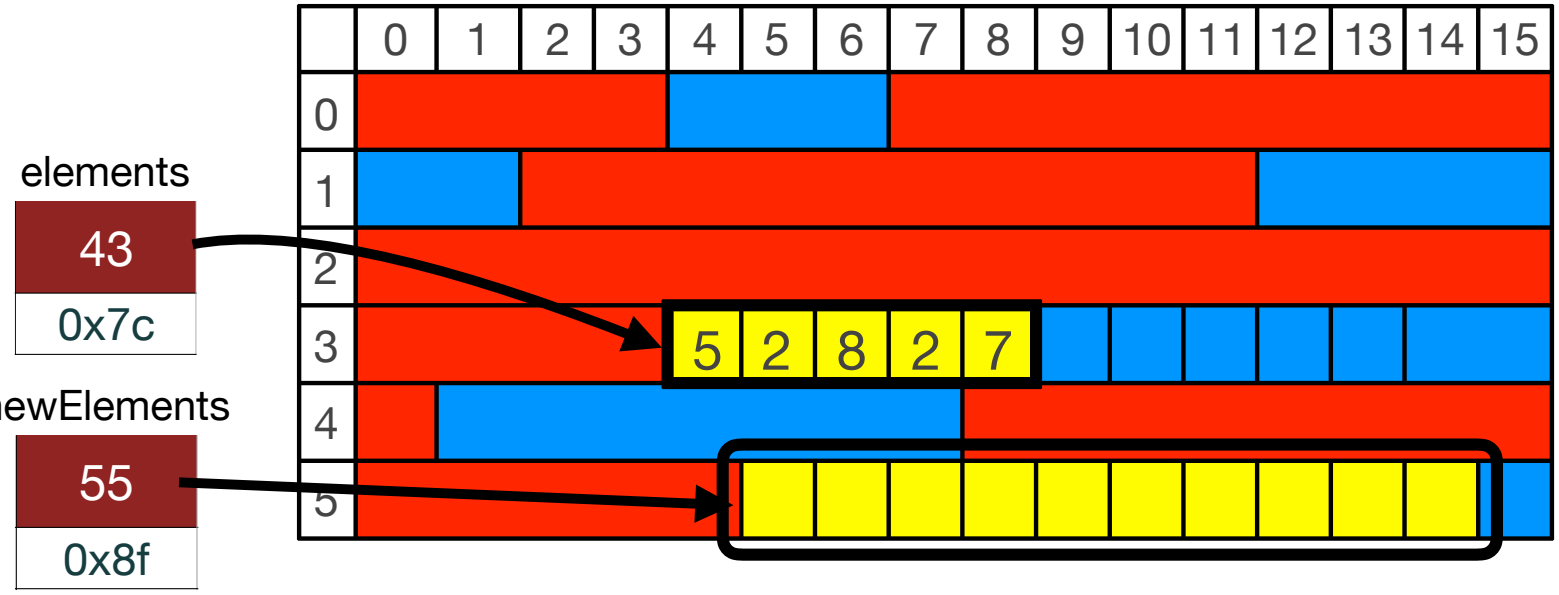
This is the option the OS has to choose.

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];
}
```



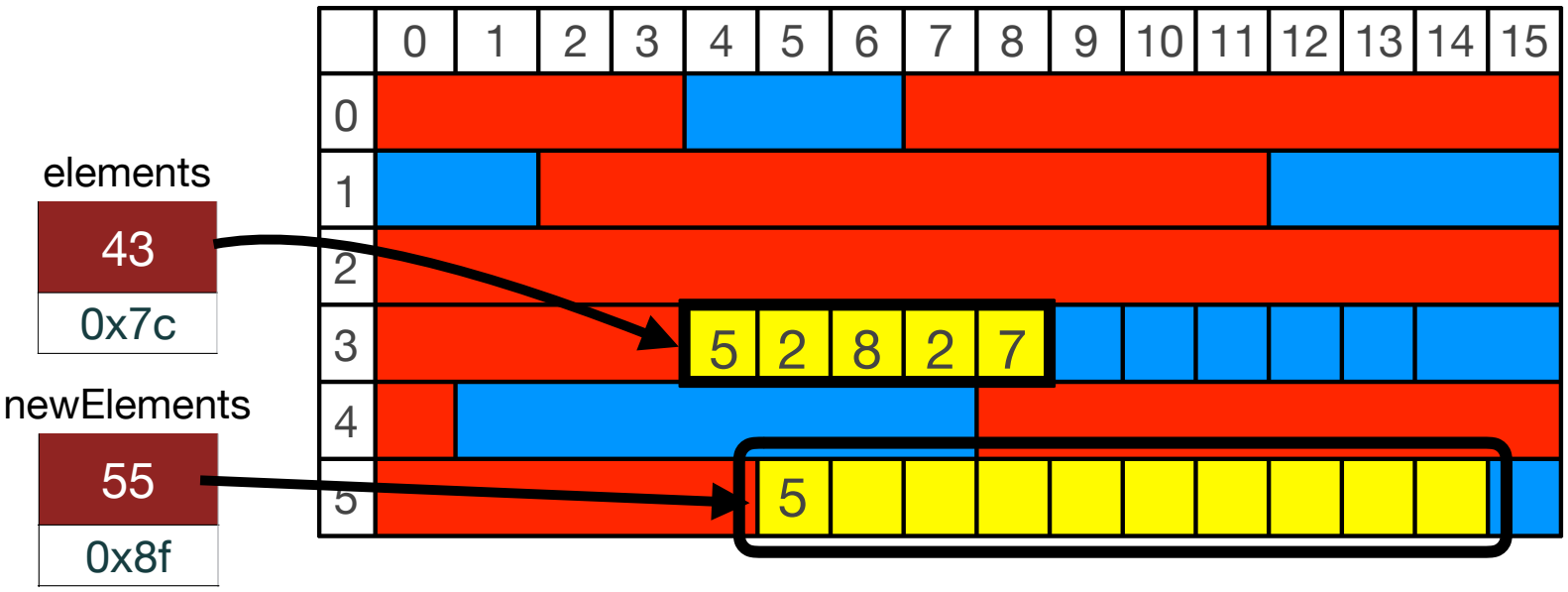
This is the option the OS has to choose.

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];
}
```



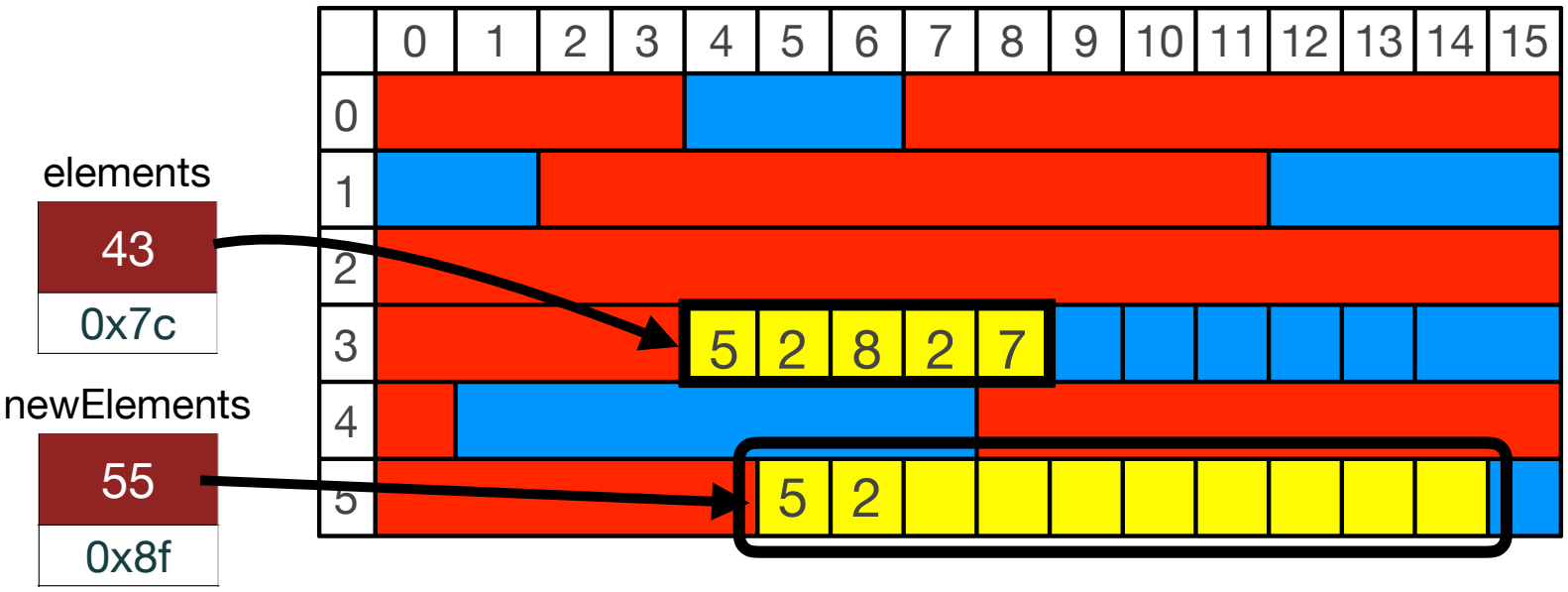
This is the option the OS has to choose.

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];
}
```



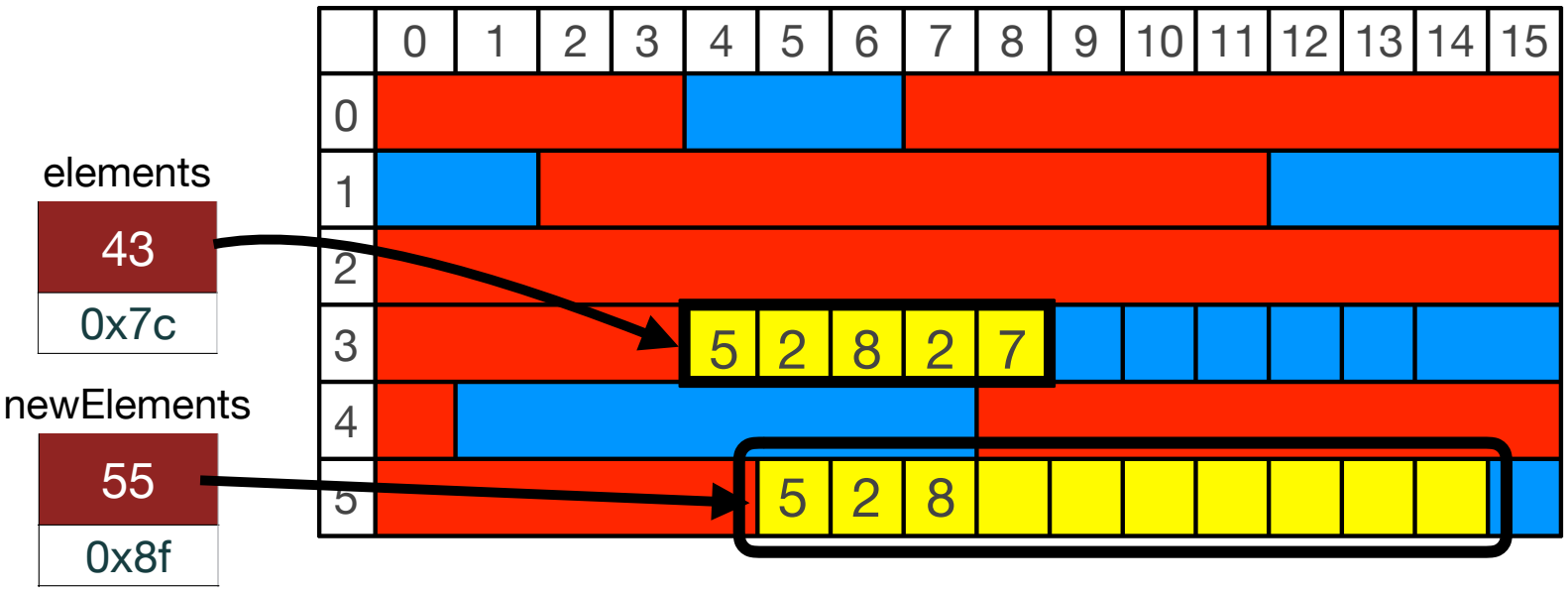
This is the option the OS has to choose.

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];
}
```



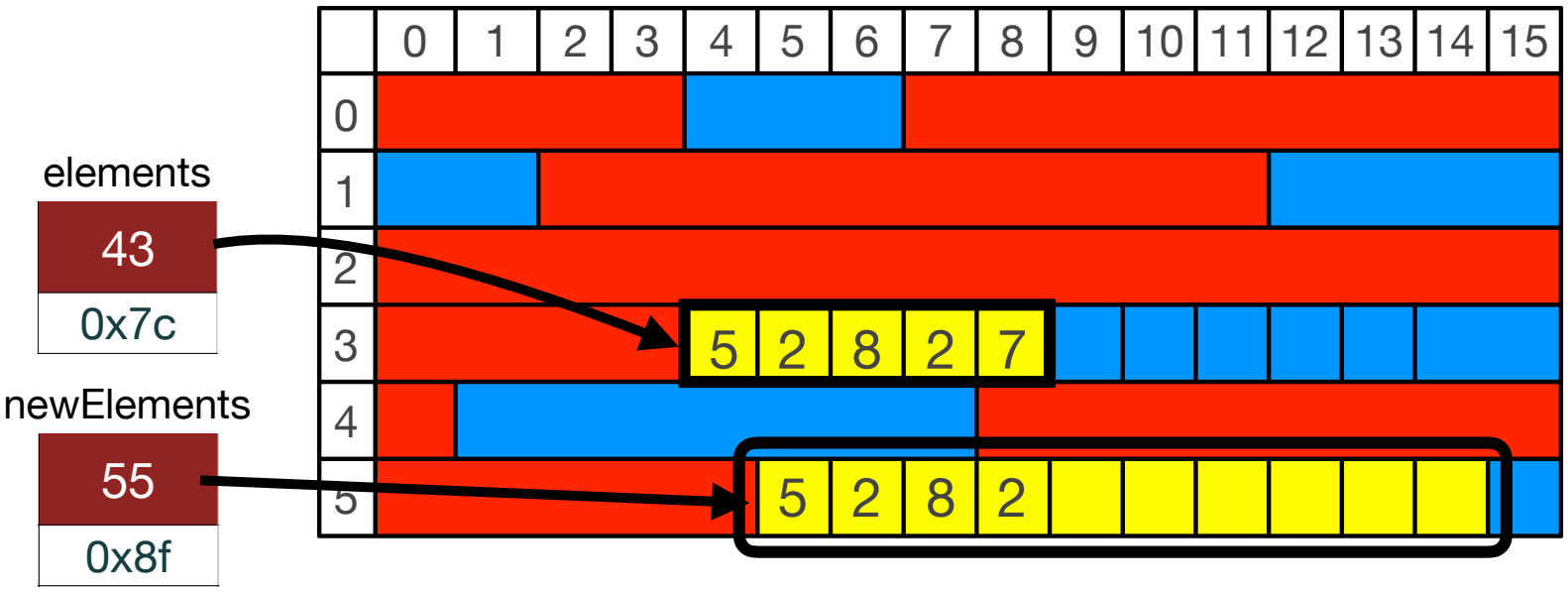
This is the option the OS has to choose.

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];
}
```



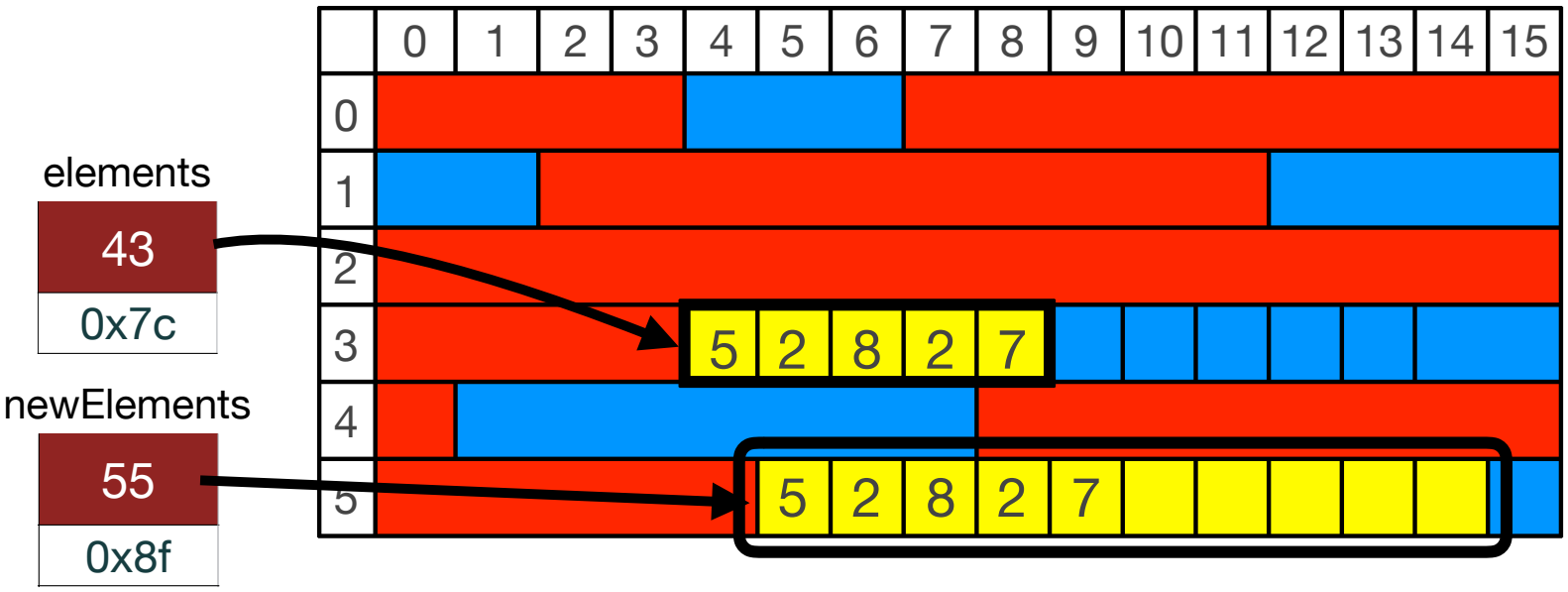
This is the option the OS has to choose.

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];
}
```



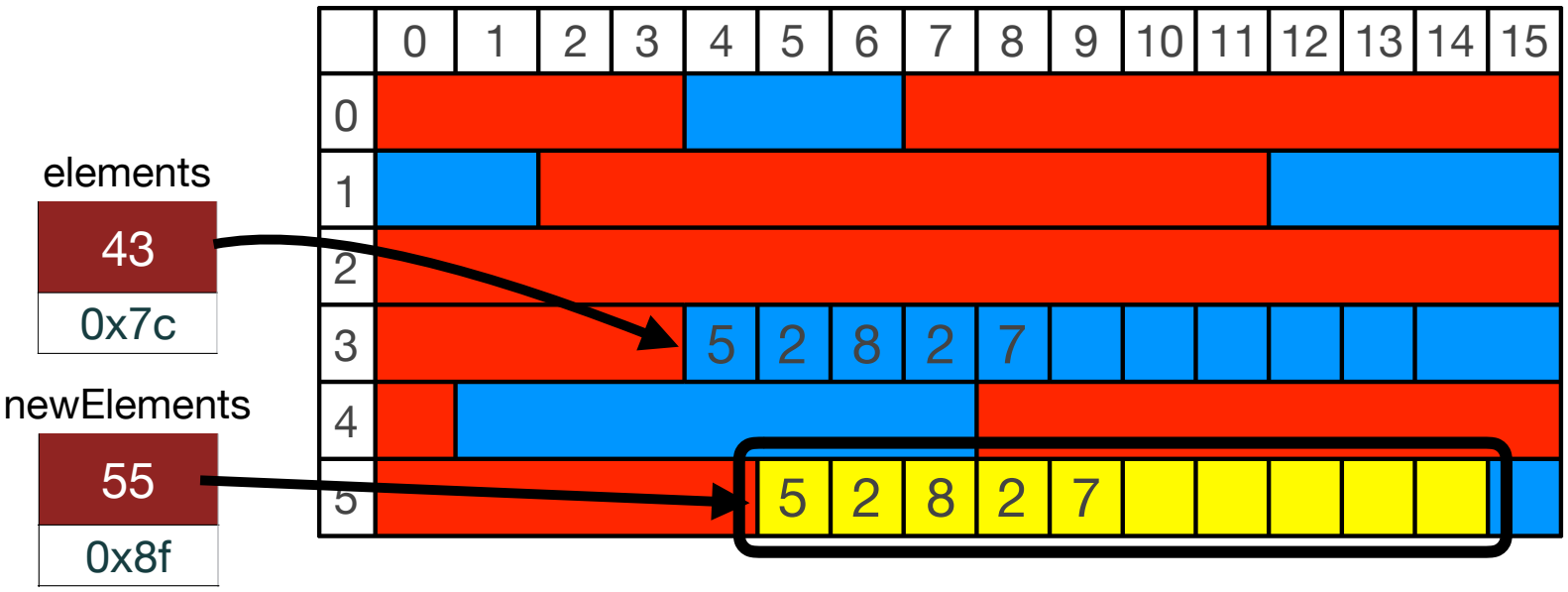
This is the option the OS has to choose.

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;
```

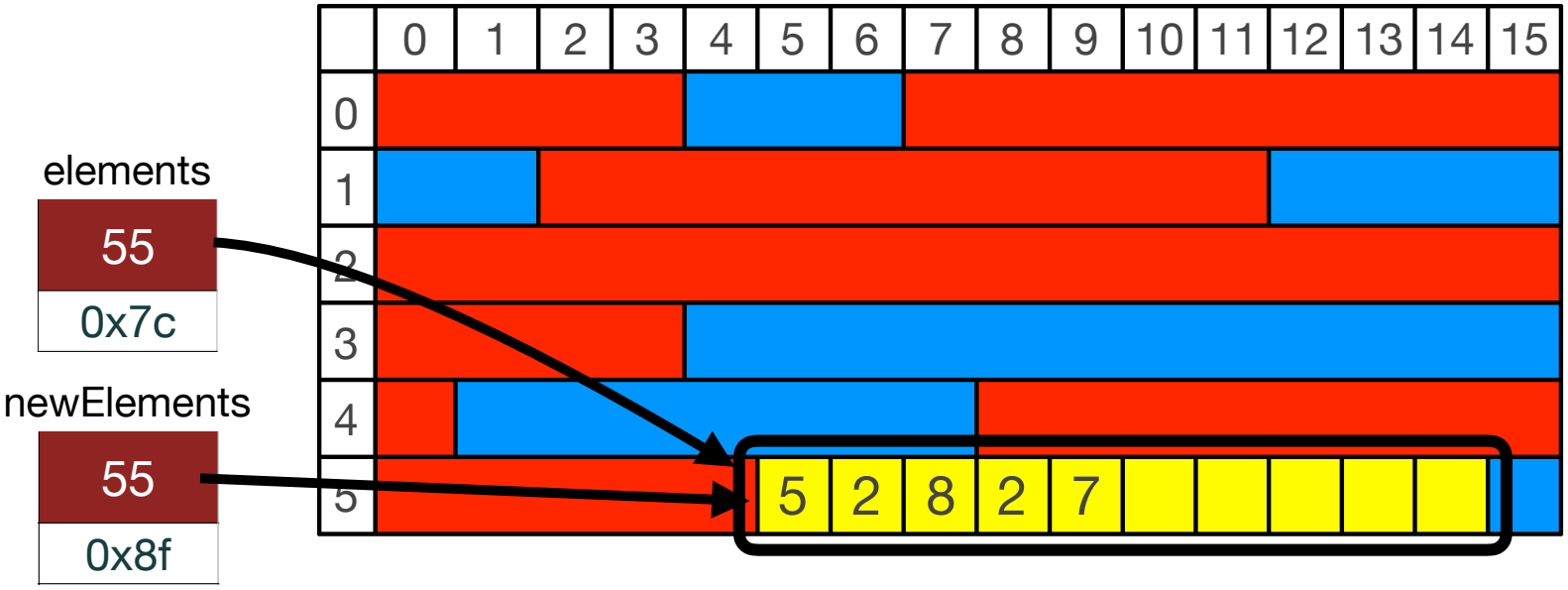


This is the option the OS has to choose.

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;
```

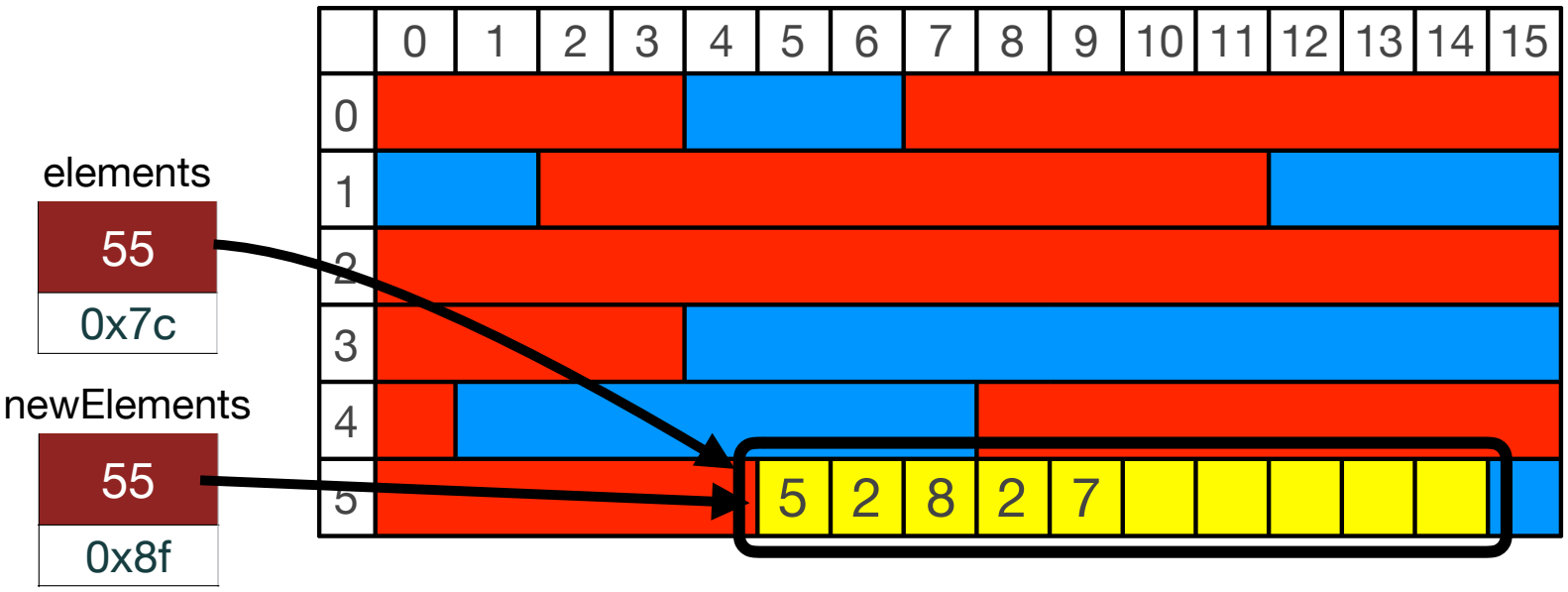


This is the option the OS has to choose.

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