Dynamic Storage Exercise
Dynamic Storage Exercise

```cpp
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[].
```
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 \texttt{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,
Dynamic Storage Solution

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

• Idea:
  1. Allocate some range (using new [])
  2. As soon as range full, allocate larger range (using new [])
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
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
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
  5. Go back to 2. with newly generated memory

(From: Script Exercise 158.a)
Dynamic Storage Solution

• And the code...

```cpp
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
```

(From: Script Exercise 158.a)
New Range - How Much Larger?
Dynamic Storage Solution

• "Much" larger?
  • Pro: ranges less often full $\rightarrow$ copy less often
  • Con: larger memory consumption

• Important: Larger by a factor, not by a constant...
  • $\text{length}_n = \text{length}_o \times 2$
  • $\text{length}_n = \text{length}_o + 2$

(From: Script Exercise 158.a)
Dynamic Storage Solution

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

<table>
<thead>
<tr>
<th>elements</th>
<th>Case a)</th>
<th>Case b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(From: Script Exercise 158.a)
Dynamic Storage Solution

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

<table>
<thead>
<tr>
<th>elements</th>
<th>Case a)</th>
<th>Case b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>32</td>
<td>18</td>
</tr>
</tbody>
</table>

(From: Script Exercise 158.a)
Dynamic Storage Solution

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

<table>
<thead>
<tr>
<th>elements</th>
<th>Case a)</th>
<th>Case b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>32</td>
<td>18</td>
</tr>
</tbody>
</table>

Case a): Significantly fewer resizings.

(From: Script Exercise 158.a)
Dynamic Storage Solution

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

<table>
<thead>
<tr>
<th>elements</th>
<th>Case a)</th>
<th>Case b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>32</td>
<td>18</td>
</tr>
</tbody>
</table>

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) constant 2

<table>
<thead>
<tr>
<th>elements</th>
<th>Case a)</th>
<th>Case b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16384</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>65536</td>
<td></td>
</tr>
</tbody>
</table>

Each resizing means:
Copy WHOLE array.

Significantly fewer resizings.

Factor 2 is an arbitrary, but good choice.

(From: Script Exercise 158.a)
Vectors
• Vectors can grow!

```cpp
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:

```cpp
def main()
{
    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:

```cpp
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
```

![Diagram showing dynamic storage in a vector with elements 0, 0, 7, 2, 6]
Dynamic Storage in Vectors

- Example:

```cpp
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:

```cpp
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:

```cpp
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
```

```
begin
0 0 7 2
end
```
Dynamic Storage in Vectors

• Example:

```cpp
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
```

Space full
Now: copy range

```cpp
begin
```

```cpp
end 12
```

0 0 7 2
Dynamic Storage in Vectors

• Example:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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
```

```
begin

0 0 7 2

end !2

0 0 7 2 6
```
Dynamic Storage in Vectors

• Example:

```cpp
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:

```cpp
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:

```cpp
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