Lecture 7

An extensible array, amortised analysis, common pitfalls

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
September 25th, 2024

IDA, Linköping University

7.1

Content

Contents

7.2

1 An extensible array

1.1 Dynamic memory

Fields/array

- `type name[length];`
  - a fix field; can not be resized
- `type* name = new type[length];`
  - a *dynamically allocated* array;
  - assignment can be done later, to change the array size
  - memory allocated dynamically must be freed manually otherwise there will be *memory leaks* in the program
- there are other differences between the two syntax
  - the objects are stored in different part of the memory; the first syntax uses the *stack* while the other use the *heap*

7.3

Free memory

- `delete[] name;`
  - Free the memory associated with the pointer
  - Must be called for all fields created with `new type[]`
    - Otherwise, the program has a memory leak (No garbage collector unlike in Java)
    - Leaked memory is freed when the program exit, but for applications with long running time a memory leak can lead to exhausting the computer memory

```cpp
int* a = new int[3];
a[0] = 42;
a[1] = -5;
a[2] = 17;
for (int i = 0; i < 3; i++) {
    cout << i << ": " << a[i] << endl;
}
...
delete[] a;
```

7.4

1.2 `ArrayList`

Example

- Write a class that implements an array of integers
  - We call it `ArrayList`
  - Behavior:

```
add(value)        insert(index, value)
get(index)        set(index, value)
size()            isEmpty()
remove(index)
indexOf(value)    contains(value)
toString()
...
```

- The size of the list will be the number of elements inserted so far
  - The actual length of the array (capacity) can be larger. Start with a size of 10 by default.

7.5

1.3 Destructor

Destructor

- ```cpp
  // ClassName.h        // ClassName.cpp
  ~ClassName();         ClassName::~ClassName() { ...
  ```
  - Called when the object is destroyed by the program (when the object goes out of scope or delete is used)
  - Can be useful to:
    * free temporary resources
    * free dynamically allocated memory used by the members
- Does `ArrayList` need a destructor? What should it do?
  - Yes; to free the memory associated with storing elements

7.6

1.4 Increase capacity

Increase capacity

- What if the users wants to add more than ten elements?

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	4	8	1	6
size	10	*capacity*	10							

```
list.add(75) //add a 11th element
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	3	8	9	7	5	12	4	8	1	6	**75**	0	0	0	0	0	0	0	0	0
size	**11**	*capacity*	**20**																	

- Answer: double the size of the field
```

- Do not forget to release the memory used by the old array!

- ```
  int* a = new int[10];
  int* b = new int[20];
  std::copy(a, a+10, b);
  delete[] a;
  a = b;
  ```

7.7

2 Amortised analysis

An extensible array

We want a new type of array that automatically increase available size when full (when the number of ellementsis n is same as the capacity N). Suppose the array always insert new element in the first free position:

- Allocate a new array B with capacity $2N$
- Copy $A[i]$ to $B[i]$, for $i = 0, \ldots, N-1$
- Lets $A = B$, we let B take over the role A had.

In term of effectiveness, expanding the array is slow. But the algorithmic complexity is:

- $O(1)$ most of the time
- $O(n)$ for copying n element and $O(1)$ for inserting after reallocation.

7.8

Amortised analysis

Using *amortisation* we can show that a sequence of insertion of element to our expandable array is effective:

Proposition 1. Let S be a table implemented using an extensible array A, as previous. The total time to insert n element in S, starting with an empty table S (which means that A has capacity $N = 1$) is $O(n)$.

7.9

3 Common C++ mistakes and pitfalls

Common mistakes: delete vs delete[]

- Memory allocated with **new** must be freed with **delete**. Memory allocated with **new**[] must be freed with **delete**[]
- Using **delete** for memory allocated with **new**[] means only one destructor is called and it leads to a crash
- can be tested with memory tracking tools, e.g., `valgrind`

```
int* q = new int;
delete q;

int* p = new int[20];
delete[] p;
```

7.10

Common mistakes: Returning a reference to a temporary

```
int& f()
{
    int a;
    return a;
}
```

7.11

Common mistakes: Throwing exception from destructor

```
class A
{
public:
     ~A() { throw 0; }
};
void f()
{
    A a;
```
```

```
    throw 0;
}
int main()
{
    try { f(); }
    catch(int ) { }
    return 0;
}
```

- C++ does not know what to do when two exceptions are thrown in parallel!

7.12

Common mistakes: Using Invalidated Iterators and Pointers

- When modifing a container, assume the old iterator is not valid anymore!
- For instance when removing elements:

```
std::vector<int> v{3,4,12,-1,4,5};
for(auto it = v.begin(); it != v.end(); ++it)
{
  if(*it == 4) { v.erase(it); } // it invalid after the erase!
}
```

Instead:

```
std::vector<int> v{3,4,12,-1,4,5};
for(auto it = v.begin(); it != v.end(); )
{
  if(*it == 4) { it = v.erase(it); } // new it is valid after the erase!
  else { ++it; }
}
```

- Or adding elements:

```
std::vector<int> v{3,4,12,-1,4,5};
auto it = v.begin();
int* first = &v[0];
v.push_back(2);
//it and first are not valid because of the push_back
std::cout << *it << "␣" << *first << std::endl; //bad
```

7.13

Use C++ library as much as possible instead of the the C standard library

- Most C functions have C++ equivalents and are safer to use:
- For instance use `std::copy` and not `memcpy`:

```
memcpy(dst, src, length * sizeof(int));
std::copy(src,  src + length, dst);
```

- Use `std::string` and not C-strings:

```
  const char* s1 = "hello";
  const char* s2 = "hello";
  if(s1 == s2)
  {
    std::cout << "Never␣shown!" << std::endl;
  }
```

- Use `ifstream` or `ofstream` and not `fopen`, `printf`, `fclose`

7.14

Conversion

- C++ automatically convert most numbers without warning
- Integer division even though saving into float:

```
  int nX = 7;
  int nY = 2;
  float fValue = nX / nY;   // fValue = 3 (not 3.5!)
```

Fixed with:

```
float fValue = static_cast<float>(nX) / nY; // fValue = 3.5
```

- mixing signed and unsigned integers

```
unsigned u = 10;
int i = -42;
cout << i + i << endl; // -84
cout << u + i << endl; // 4294967265
```

7.15

Side effects

- Should the following print 25, 30 or 36?

```
void multiply(int x, int y)
{
    using namespace std;
    cout << x * y << endl;
}

int main()
{
    int x = 5;
    multiply(x, ++x);
}
```

- order of evaluation of arguments is **undefined**!

7.16

Switch statements without break

```
switch(v)
{
  case 1:
    str = "one";
  case 2:
    str = "two";
  case 1:
    str = "three";
  ...
};
```

- The correct way is:

```
switch(v)
{
  case 1:
    str = "one";
    break;
  case 2:
    str = "two";
    break;
  case 1:
    str = "three";
    break;
  ...
};
```

7.17

4 vector vs dequeue

push_back: vector and dequeue

```
// Vector test code
vector<int> v;
// Insert at the start of the vector
for (int i = 0; i < N; i++)
    v.push_back(i);
// Clear by using pop_front (erase)
for (int i = 0; i < N; i++)
    v.pop_back();
```

```
// Deque test code
deque<int> d;
// Insert elements using push_front
for (int i = 0; i < N; i++)
    d.push_back(i);
// Clear by using pop_front
for (int i = 0; i < N; i++)
    d.pop_back();
```

	<vector>	<deque>
N = 10 000 000	40ms	250ms
N = 1 000 000 000	2200ms	2200 ms

7.18

push_front: vector and dequeue

```
// Vector test code
vector<int> v;
// Insert at the start of the vector
for (int i = 0; i < N; i++)
    v.insert(v.begin(), i);
// Clear by using pop_front (erase)
for (int i = 0; i < N; i++)
    v.erase(v.begin());

// Deque test code
deque<int> d;
// Insert elements using push_front
for (int i = 0; i < N; i++)
    d.push_front(i);
// Clear by using pop_front
for (int i = 0; i < N; i++)
    d.pop_front();
```

	<vector>	<deque>
N = 10000	10ms	0ms
N = 100000	660ms	0ms
N = 1000000	TO	5ms
N = 10000000	TO	40ms

Timeout TO = 30 seconds. 7.19