

Lesson 4: Use Arrays



Variable length arrays, implementing stacks

Use Arrays

- An array is a contiguous chunk of memory
- In C we can create arrays using
`int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance, although never use raw arrays!

Outline

1. Why Arrays?
2. Variable Length Arrays
3. Programming Language
4. Implementing Stacks



ADT for a List

- What do we want of a list of `ints`?
 - ★ `void push_back(int value)`
 - ★ random access `array[i]`
 - ★ `int size()`
- It would be useful if it resized
- It would be great to have some algorithms (e.g. sort) that can be run on a list

1. Why Arrays?
2. Variable Length Arrays
3. Programming Language
4. Implementing Stacks



Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
 - ★ E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are—annoying

Variable Length Arrays

- We want a variable length array
- Initially a variable length array would have length zero
- We should be able to
 - ★ Add an element to an array
 - ★ Access any element in the array
 - ★ Change an element
 - ★ Delete elements
 - ★ Know how many elements we have

Implementation

- How should we implement a list?
- Use an array, of course!
- We need to distinguish between
 - ★ the number of elements in the list `size()`
 - ★ the number of elements in the array `capacity()`
- If the number of elements grows larger than the capacity then we need to increase the capacity

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use $c n$ bits of memory where we want to keep c small

BIOM2005

Further Mathematics and Algorithms

9

Amortised Time Analysis

- How efficient is resizing?
- Most `push_back(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- Adding to a full array is slow but it is **amortised** by other quick adds
- **amortised**: effect of a single operations 'deadened' by other operations

BIOM2005

Further Mathematics and Algorithms

11

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where

$$\begin{array}{ccccccc}
 0 & n & 2n & 4n & 8n & 16n & 32n \\
 \hline
 n \times 2^{m-1} < N \leq n \times 2^m & \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil
 \end{array}$$

- The number of elements copied is

$$n + 2n + 4n + \dots + 2^{m-1}n = n(1 + 2 + \dots + 2^{m-1}) = n(2^m - 1)$$
- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n(2^m - 1) = N + n2^{\lceil \log_2(\frac{N}{n}) \rceil} - n < N + 2N - n < 3N$$

BIOM2005

Further Mathematics and Algorithms

13

Outline

1. Why Arrays?
2. Variable Length Arrays
3. **Programming Language**
4. Implementing Stacks



BIOM2005

Further Mathematics and Algorithms

15

Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
 - ★ create a new array with double the capacity of the old array
 - ★ copy the existing elements from the old array to the new array

`list.push_back(83)` →

31	35	85	23
----	----	----	----

31	35	85	23	62			
----	----	----	----	----	--	--	--

BIOM2005

Further Mathematics and Algorithms

10

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
 - ★ adds: 100
 - ★ copies: $10 + 20 + 40 + 80 = 150$
 - ★ `new int[]`: 4
- 250 adds and copies operations + 4 **new** operations

BIOM2005

Further Mathematics and Algorithms

12

Insertion and Deletion

- `vector<T>` is very useful and very fast for lots of things
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order (binary trees will do that for you much more efficiently)
- Linked lists allow you to splice in a sublist into a list in constant time although linked lists have a lot of drawbacks

BIOM2005

Further Mathematics and Algorithms

14

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C) it pays the price of allowing bugs that hard to detect
- **Java** was designed to be very safe (avoiding lots of bugs) but is not fast and a bit long winded
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code but it is not fast or safe

BIOM2005

Further Mathematics and Algorithms

16

- Amongst a number of issues that make C++ dangerous are
 - ★ Memory management
 - ★ Writing to parts of memory that you should not
 - ★ Multiple inheritance, although you seldom need to do this
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue

BIOM2005

Further Mathematics and Algorithms

17

Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapidly exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use:


```
delete[] storage;
```
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

BIOM2005

Further Mathematics and Algorithms

19

Writing over Memory

- In C++ the following will compile and run


```
int *array = new int[4];
int *a = new int[2];
double *darray = new double[4];
array[4] = 4;
```
- However `array[4]` has not been assigned (unlike `array[0]`, `array[1]`, `array[2]` and `array[3]`)
- The memory on the heap corresponding to the address of `array[4]` might have been assigned to `a[0]` in which case you may inadvertently have set `a[0]` to 4 leading to the program not doing what you want
- It might be that you have put an `int` into `darray[0]` which will then crash the system when you read `darray[0]`

BIOM2005

Further Mathematics and Algorithms

21

Follow Programming Idioms

- Using common data structures and following common idioms will prevent most errors

```
int n = 5;
vector<int> array(n);

for(int i=0; i<array.size(); ++i) {
    array[i] = i;
}

for(auto pt=array.begin(); pt != array.end(); ++pt) {
    *pt *= 2;
}

for(int& element: array) {
    element += 2;
}
```

BIOM2005

Further Mathematics and Algorithms

23

- Most programming languages have two types of memory

The Stack: is the area of memory controlled by compiler for local variables, function calls, etc.

The Heap: is area that the programmer (you) can request, which is nice

- In C++ you are given the **right** to ask for memory

```
int *storage = new int[n];
```

- You have **responsibility** to free the memory

```
delete[] storage;
```

BIOM2005

Further Mathematics and Algorithms

18

Resource Acquisition is Initialisation (RAII)

- Java and Python use garbage collectors which automatically check whether memory can be accessed and if not it is removed
- In C++ this is your responsibility
- But there is a standard **programming pattern** to elevate the problem known as **Resource Acquisition is Initialisation (RAII)**

Wrap all resources in classes. Request the resources in the constructor and release the resource in the destructor
- When the object goes out of scope (you leave a `for` loop, function call, etc.) the destructor is called and the resource is safely released

BIOM2005

Further Mathematics and Algorithms

20

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

BIOM2005

Further Mathematics and Algorithms

22

Outline

1. Why Arrays?
2. Variable Length Arrays
3. Programming Language
4. Implementing Stacks



BIOM2005

Further Mathematics and Algorithms

24

- Lets look at implementing a stack
- Remember a stack has methods
 - ★ `push(Object)`
 - ★ `pop()`
 - ★ `top()`
 - ★ `empty()`

Notes on Implementation

- I don't need to write a constructor as C++ generates a default constructor that will initialise the stack correctly
- I don't need to write a destructor because by default the destructor for `vector<T>` will be called which releases memory
- I've written the `pop` command, that I like, but if I run


```
stack<Widget> widget_stack;
Widget w;
widget_stack.push(w);
Widget w1(widget_stack.pop());
```

if the last command throws an exception then the last term on the stack is lost for ever

Using MyStack

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you can't swallow a cage can't you

you
can't
swallow
a
cage
can
you

Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays frequently

```
template <typename T>
class MyStack
{
private:
    std::vector<T> stack;

public:
    void push(const T& obj) {stack.push_back(obj);}

    T top() const {return stack.back();}

    T pop() {
        T tmp = stack.back();
        stack.pop_back();
        return tmp;
    }

    T empty() {return stack.size()==0;}
};
```

Why not use a vector

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory—I don't need to write comments as it is clear what I am doing

Reversing Strings in File

```
#include <stack>
#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>
using namespace std;

int main(int argc, char *argv[]) {
    ifstream in(argv[1]);

    stack<string> stack;

    string word;
    while (in >> word)
        stack.push(word);

    while(!stack.empty()) {
        cout << stack.top() << ' ';
        stack.pop();
    }
}
```