Further Mathematics and Algorithms

Lesson 6: *Writing an Arrays*

Common errors, memory leaks, templates

Introduction

- These are notes on the tutorial session on writing a resizeable array
- We did not get very far with them in the first lecture
- In this lecture we are going to make are code more solid
- Add some functionality
- Make the code generic

Copy Constructor

- C++ conveniently generates a copy constructor

```
    Array b(a);
```

- Unfortunately this copies the address to `data` and the `length`

- But his is a *shallow copy* which means that both arrays work on the same data array

- This would be deeply confusing. Instead we have to write our own *copy constructor* to do a deep copy

```
Array::Array(Array& other) {
  data = new int[other.size()];
  length = other.size();
  for(int i=0; i<size(); ++i) {
    data[i] = other[i];
    }
}
```

Assignment Constructor

- We can also generate a new array through assignment

```
Array a = b;
```

- As with the copy constructor this is generated by default
- However, it calls the copy constructor
- If we fix the copy constructor this now works as expected
- Almost . . .

Being Explicit

- One oddity of C++ is that the following code compiles

```
Array a = 4;
```

- We have not defined what happens when we set and array to an integer

- However, the compiler tries to make sense of this and sees that it can create an array on the right-hand side using the constructor

```
Array(int n);
```

- It sees this as a way of promoting an integer to an array

- This isn't what most people would expect. I expect a compile error

- To achieve this I can redefine the constructor

```
explicit Array(int n);
```

Compilers are our Friends

- Compile errors are our friends: they are quick to fix and prevent serious errors
- One little understood strength of C++ is the compiler allows us to determine what changes
- Defining the function

  ```
  void print(const Array&, string name);
  ```

  passes the array by const reference. This is efficient. Making it const means we know print won't change the reference
- But this triggers a whole lot of consequence because print is only allowed to use const member functions

Constant consistency

- At first it appears we have opened a can of works
- We have declare lot of member functions as constant

```
int size() const;

int& operator[](int index);
int operator[](int index) const;
```

- We have to declare a constant version of the access operator
- When you first do this it seems like a lot of unnecessary work
- But the is some satisfaction in specifying all the functions consistently
- And in the long run it will prevent many bugs

Memory Leaks

- Another “bug” in our code is that we are grabbing memory, but not giving it up

- This can become very expensive

```
for(int i=0; i<500000; i++) {
  Array a(10000000);
  if (i % 10000==0) {
    cout << i << endl;
    sleep(1);
  }
  cout << "Finished\n";
}
```

- In linux I can look at memory usage using
  `top -c $(pgrep -d’,’main)$`

RAII

- The method for preventing memory leaks is known as “Resource Allocation is Initialisation”

- This means we take the resource (in this case memory) in the constructor of a class

- And give it back in the destruction

```
Array::~Array() {
  delete[] data;
}
```

Make it Generic

- To make an array for doubles or strings we only have to change the type of the data from `int` to `double` or `string`
- We can write a template with `T` (or any other name we want to use) as representing some generic type
- We can't compile the code as the compiler needs to know the type we are using
- We would therefore have to do a global replace of `T` by the type we want to use and create new `array.h` and `array.cc` for each type of array we use
- Fortunately the C++ compiler will do this for us

Template Programming

- To do this all we need to do is write `template <typename T>` in front of any class or function that uses a generic type

- These need to be included in the header file

- In the main code to ask for an array of type `string`, for example, we write

```
      Array<string> string_array;
```

- The compiler invisibly creates the code for this class, but replacing the template variable by `string`

Template Code Example

```
template <typename T>
class Array {
private:
  T *data;
  unsigned length;
public:
  explicit Array(int n);
  Array(const Array& other);
  ~Array();
  T& operator[](unsigned index);
  T operator[](unsigned index) const;
  unsigned size() const;
};

template <typename T>
Array<T>::Array(int n) {
  data = new T[n];
  length = n;
}
```