Notes on building an array class in C++

Adam Prugel Bennett

October 8, 2024

1 Files

- main.cpp
- array.h
- array.cpp
- Makefile

```
main: array.h main.cpp array.cpp
     g++ main.cpp array.cpp -o main
     ./main
```

2 Getting started

- Header

```
class Array {
public:
  Array(int n);
private:
  int* data;
  int my_size;
};
```

- array.cpp

```
#include "array.h"

Array::Array(int n) {
  data = new int[n];
  my_size = n;
}

int Array::size() {
  return my_size;
}
```

3 Getter

- get command

```
int& Array::get(int i) {
  return data[i];
}
```

- see what we have done

```
#include <iostream>
#include "array.h"
using namespace std;

int main(int, char**) {
  Array a(3);
  a.get(0) = 22;
  a.get(1) = 111;
  a.get(2) = 33;

  cout << a.get(0) << ", " << a.get(1) << ", " << a.get(2) << endl;
  return 0;
}
```

- operator overloading, replace get with []

```
int& Array::operator[](int i) {
  return data[i];
}
```

4 Useful default behaviour

4.1 Copy constructor

- default constructure

```
void print(Array& a, string name) {
  cout << name << ": " << a[0];
  for(int i=1; i<a.size(); ++i) {
    cout << ", "<< a[i];
  }
  cout << endl;
}

int main(int, char**) {

  Array a(3);

  a[0] = 0;
  a[1] = 11;
  a[2] = 222;

  print(a, "a");

  Array b(a);

  print(b, "b");

  return 0;
}
```

– So far so good but

```
int main(int, char**) {

  Array a(3);

  a[0] = 0;
  a[1] = 11;
  a[2] = 222;

  print(a, "a");

  Array b(a);

  print(b, "b");

  a[0] = 777;
  cout << "-----------------\n";
  print(a, "a");
  print(b, "b");

  return 0;
}
```

– Adding an explicit copy constructor

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

4.2 Assignment constuctor

- Replace `Array a(b);` with `Array a = b;`
- Add assignment constructor

```
Array& Array::operator=(const Array& rhs) {
  data = new int[rhs.my_size];
  my_size = rhs.my_size;
  for(int i=0; i< my_size; ++i) {
    data[i] = rhs.data[i];
  }
  return *this;
}
```

5 Memory leeks and hanging points

- Looking at memory

```
#include <unistd.h>
```

```
int main(int, char**){

  for(int i=0; i<500000; ++i) {
    Array a(100000000);
    if (i%10000==0) {
      cout << i << endl;
      sleep(1);
    }
  }
  cout << "Finished\n";

  return 0;
}
```

- `top -c -p $(pgrep -d',' main)`
- Add a destructor

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

- How does it work?
  - Whenever you create an Array object and it goes out of scope the destructor is called and frees the memory
- Design pattern **Source allocation is initialisation**
- Used throughout C++
- If you do this properly you don't have to worry about memory leaks
- Used for other resources (open-close files, database tokens, etc.)

6 Const consistency

- The compiler is your friend
  - Compiler errors takes seconds or minutes to fix
  - Bugs in your code can take minutes or hours
- Let's modify `print`

```
void print(Array& a, string name) {
  cout << name << ": " << a[0];
  for(int i=1; i<a.size(); ++i) {
    cout << ", "<< a[i];
  }
  cout << endl;
  a[0] = 999;
}
```

- We could pass by value

```
void print(Array a, string name) {
  cout << name << ": " << a[0];
  for(int i=1; i<a.size(); ++i) {
    cout << ", "<< a[i];
  }
  cout << endl;
  a[0] = 999;
}
```

- This is expensive
- I have to copy the whole array, but I'm not changing it
- Let we declare that the array is not to be modified

```
void print(const Array& a, string name) {
  cout << name << ": " << a[0];
  for(int i=1; i<a.size(); ++i) {
    cout << ", "<< a[i];
  }
  cout << endl;
  a[0] = 999;
}
```

- Need to declare a new access operators

```
int Array::operator[](int i) const {
    return data[i];
}
```

- or

  ```
  const int& Array::operator[](int i) const {
      return data[i];
  }
  ```

- For integers there is no advantage, but if I modify the array to be an array of memory intensive objects then the latter is preferred.
- Note that the final `const` declares that the member function does not change the underlying data
- Need to declare that `size` is a const function
- It seems expensive but notice that you can't modify the array within `print`
- When you get used to it there is a satisfying feeling of making your classes const consistent
- The compiler will usually tell you when you have violated const consistency

6.1 unsigned

- While we are at refactoring our code lets make `my_size` be `unsigned`
- We can't have negative size arrays

7 Generic programming

- `Array` is going to be useful, but what if we want to store double or floats
- It's going to be annoying to write a data structure for every possible type
- **Templates** to the rescue

```
#include <memory>

template<typename T>
class Array {
public:
  Array(unsigned n);
  Array(const Array<T>& other);
  ~Array();
  Array& operator=(const Array&);
  T& operator[](unsigned i);
  const T& operator[](unsigned i) const;
  unsigned size() const;
private:
  T* data;
  unsigned my_size;
};


template<typename T>
Array<T>::Array(unsigned n) {
    data = new T[n];
    my_size = n;
}

template<typename T>
Array<T>::Array(const Array<T>& other) {
    data = new T[other.my_size];
    my_size = other.my_size;
    for(unsigned i=0; i< my_size; ++i) {
      data[i] = other.data[i];
    }
}

template<typename T>
Array<T>::~Array() {
  delete data;
}

template<typename T>
Array<T>& Array<T>::operator=(const Array<T>& rhs) {
    data = new unsigned[rhs.my_size];
    my_size = rhs.my_size;
    for(unsigned i=0; i< my_size; ++i) {
      data[i] = rhs.data[i];
    }
    return *this;
}
```

```
template<typename T>
unsigned Array<T>::size() const {
  return my_size;
}

template<typename T>
T& Array<T>::operator[](unsigned i) {
    return data[i];
}

template<typename T>
const T& Array<T>::operator[](unsigned i) const {
    return data[i];
}
```

- We don’t write template code in a `.cpp` file as it is not compiled
- We need to change `main.cpp`

```
#include <iostream>
#include "array.h"

using namespace std;

template<typename T>
void print(const Array<T>& a, string name) {
  cout << name << ": " << a[0];
  for(int i=1; i<a.size(); ++i) {
    cout << ", "<< a[i];
  }
  cout << endl;
}

int main(int, char**) {

  Array<int> a(3);

  a[0] = 0;
  a[1] = 11;
  a[2] = 222;

  print(a, "a");

  Array<int> b = a;

  print(b, "b");

  a[0] = 777;
  cout << "-----------------\n";
  print(a, "a");
  print(b, "b");

  return 0;
}
```

- When the compiler finds `Array<int>` it compiles the code with `T` replaced by `int`
- Templates take a bit of getting used to but for data-structures they ace

8 Variable Length Arrays

- Back to data-structures 101
- How do we make a variable length array?
- Firstly, why do we need a variable length array?
  - Reading from a file

```
#include <fstream>
using namespace std;

int main() {

  ofstream file;
  file.open("some_numbers.txt");
  Array<int> a
  while (!file.eof()) {
    a.push_back(file.get());
  }

  cout << a.size() << ", " << a[0] << endl;

}
```