How to use C++ effectively?

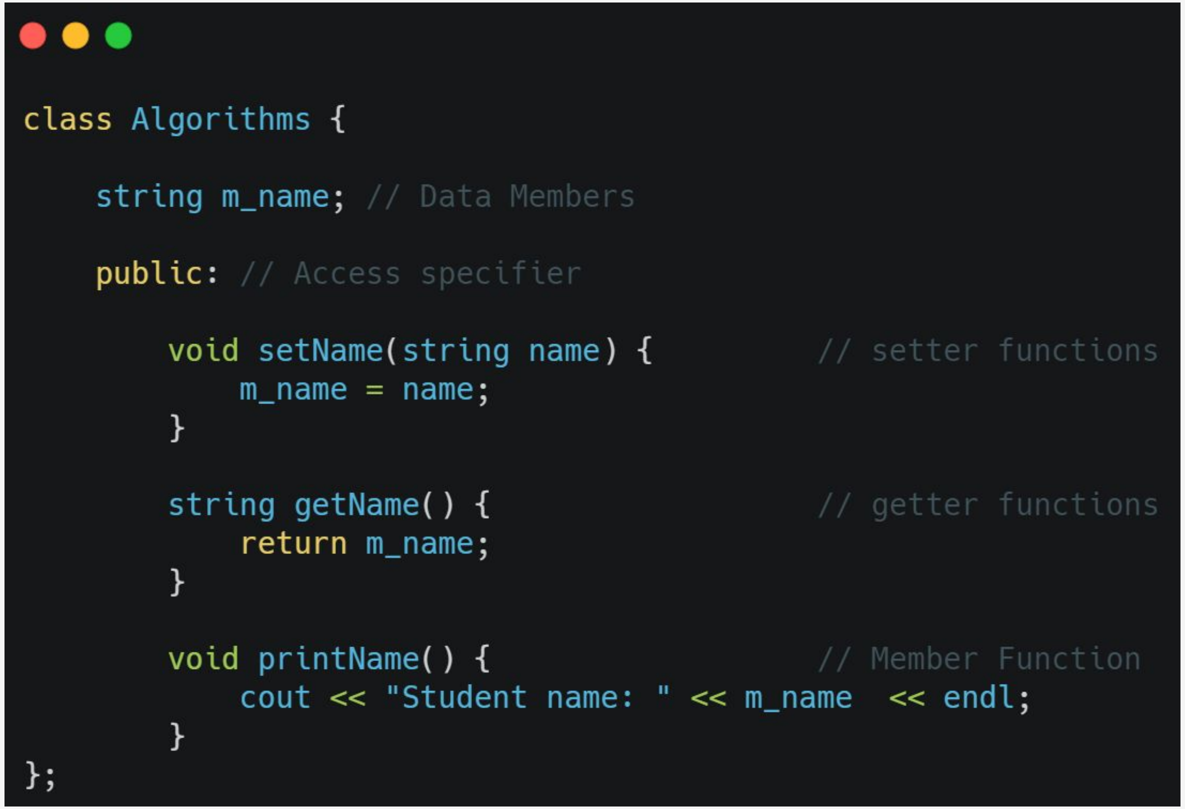
…

Contents

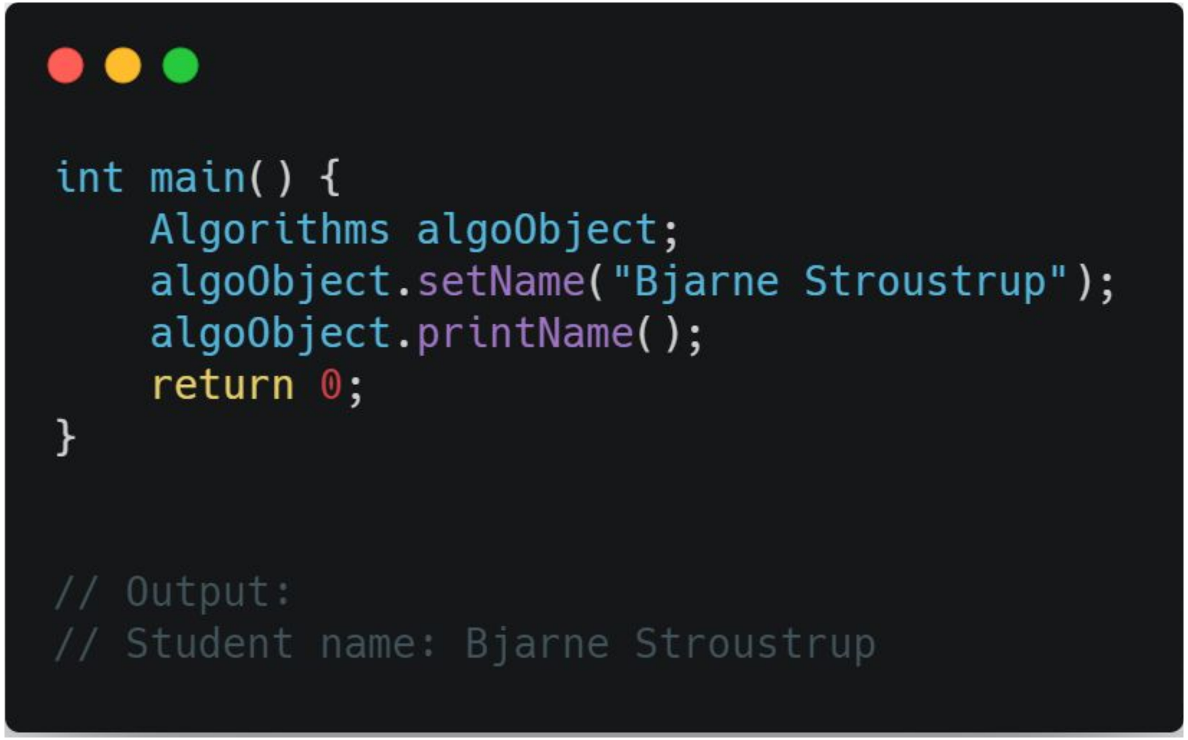
Classes & Objects

- You can think of classes as **encapsulation** of variables and methods.
- One way to write code is **data oriented,** where you have some input data, you declare associated variables and functions; you call functions on input data and you have output data.
- That is okay, for small codes but when you are writing code for big softwares i.e 10,000 files each with 10,000 lines of code you would probably want some form of organisation of code so that it facilitates easy **reusability**.
- Just like functions help organise lines of code and facilitate reusability; classes do it at a more higher level.

Sample code



```
class Algorithms {

    string m_name; // Data Members

    public: // Access specifier

        void setName(string name) {         // setter functions
            m_name = name;
        }

        string getName() {                  // getter functions
            return m_name;
        }

        void printName() {                  // Member Function
            cout << "Student name: " << m_name  << endl;
        }
};
```



```cpp
int main() {
    Algorithms algoObject;
    algoObject.setName("Bjarne Stroustrup");
    algoObject.printName();
    return 0;
}


// Output:
// Student name: Bjarne Stroustrup
```

Containers in C & C++

- The only container C provides in **array**.
- When you read more about memory organisation in a computer, array is the only container which comes with **zero** overhead. You essentially take a unit of data char or int and you set the associated bits. If you want to think in terms of pointer, you are returned an address of memory that you can use.
- This comes with **high risk!** because now it is programmer's responsibility to ensure they don’t access memory outside the allocated region (otherwise?!).
- And **high reward** because they are insanely fast! Programmers who work close to hardware, who care about 1ns of latency use C, others even code in **Assembly**.
- STL: Standard Template Library provides some mechanism through which you can safeguard yourself from errors but this error checking comes at a cost of time.

STL Containers

- Two types: sequence and associative.
- Sequence: Programmers define the sequence of elements.
  - Example: vector, list, deque.
  - These containers are classes as well which abstract the underlying data structure.
  - Public methods such as insert, erase, clear are available to make changes.
- Associative: Container controls the sequence of elements in container.
  - Example: map, set, multimap, multiset.
  - How do you access the elements? Key.
  - These containers are also implemented as classes, and provide similar public methods for inserting and removing elements.

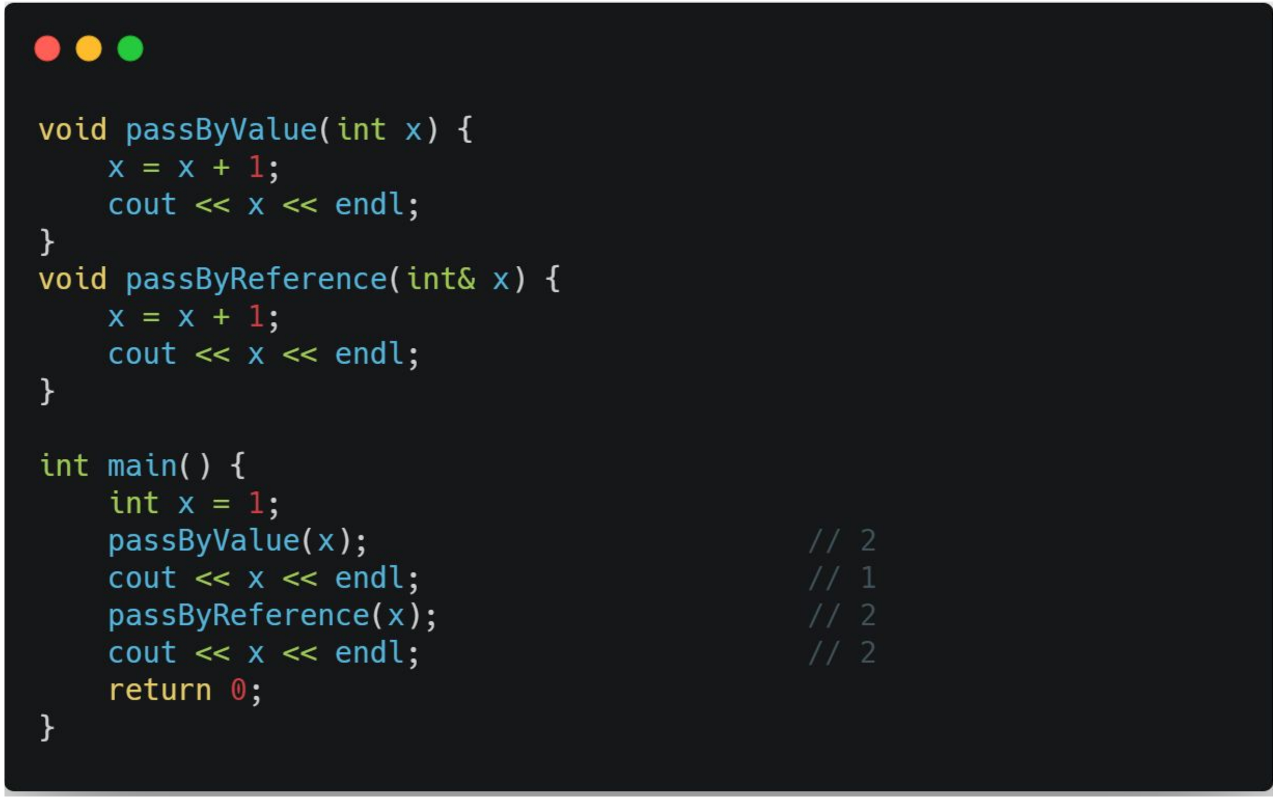
vector<T>

- The underlying data structure is an array and is implemented using a **class**.
- T is the typename. For example: vector<int> A would be an array of integers.
- It is not necessary to declare size during initialisation.
- It has member methods such as:
  - push_back(1): insert 1 at the end of the vector. (Complexity?) (size of array?)
  - insert(A.begin(), 1): insert 1 at the beginning of the vector. (Complexity?)
  - pop_back(): remove element from the end of vector.
  - erase(A.begin()): remove first element of the vector.
  - size(): return the size of the vector.
- Useful for STL algorithm:
  - sort using sort(A.begin(), A.end())
  - perform binary search using lower_bound(A.begin(), A.end(), value)

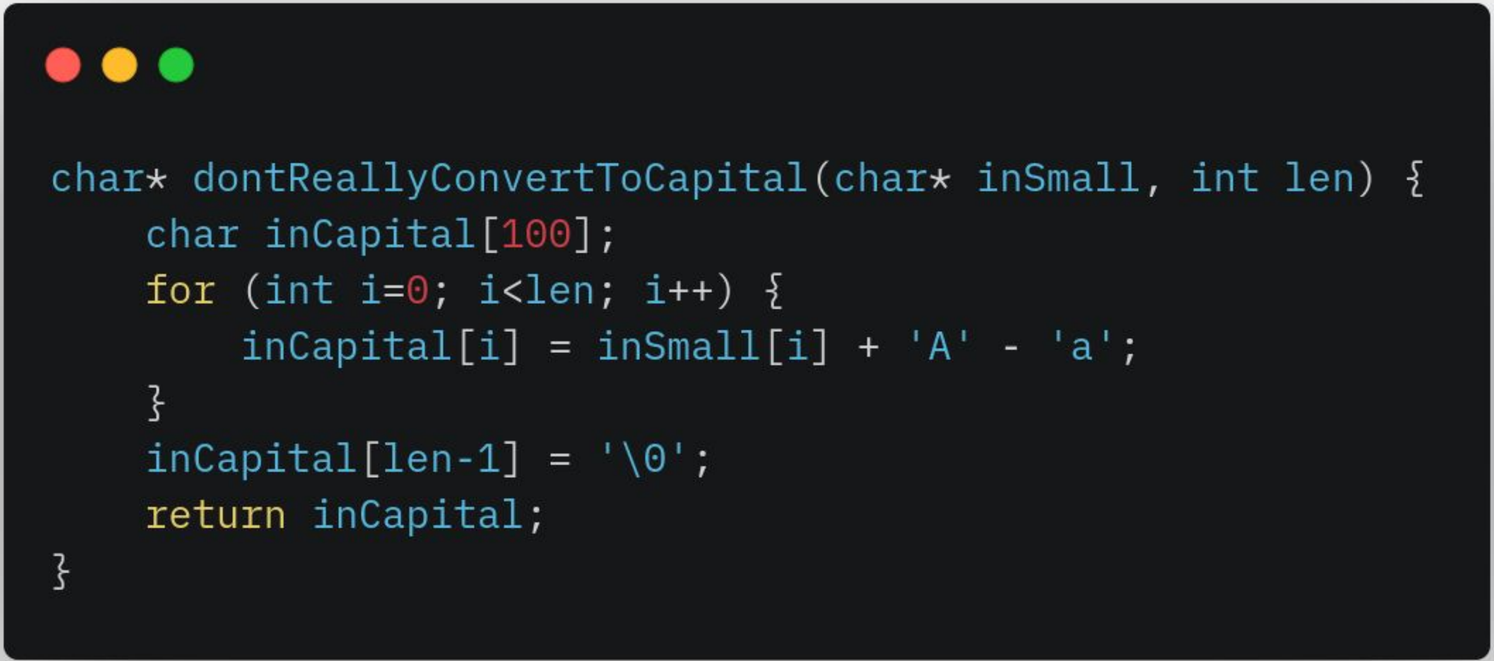
map<T1, T2>

- The underlying data structure is a BBST over keys.
- T1: Key, T2: Value . For example: map<int, string> to store a map of roll no. to name.
- It has member methods such as:
  - how to get all the keys in sorted order? **iterator.**
  - insert({key, value}): insert the key value pair into the container. (Complexity?)
  - erase(key): find the key, erase **all** nodes with this key. (Complexity?)
- set: key and value are same. Example set<int> to store unique integer values.
- multiset, multimap: can have multiple elements in the container with the same key.
  - What changes to be done in BBST?

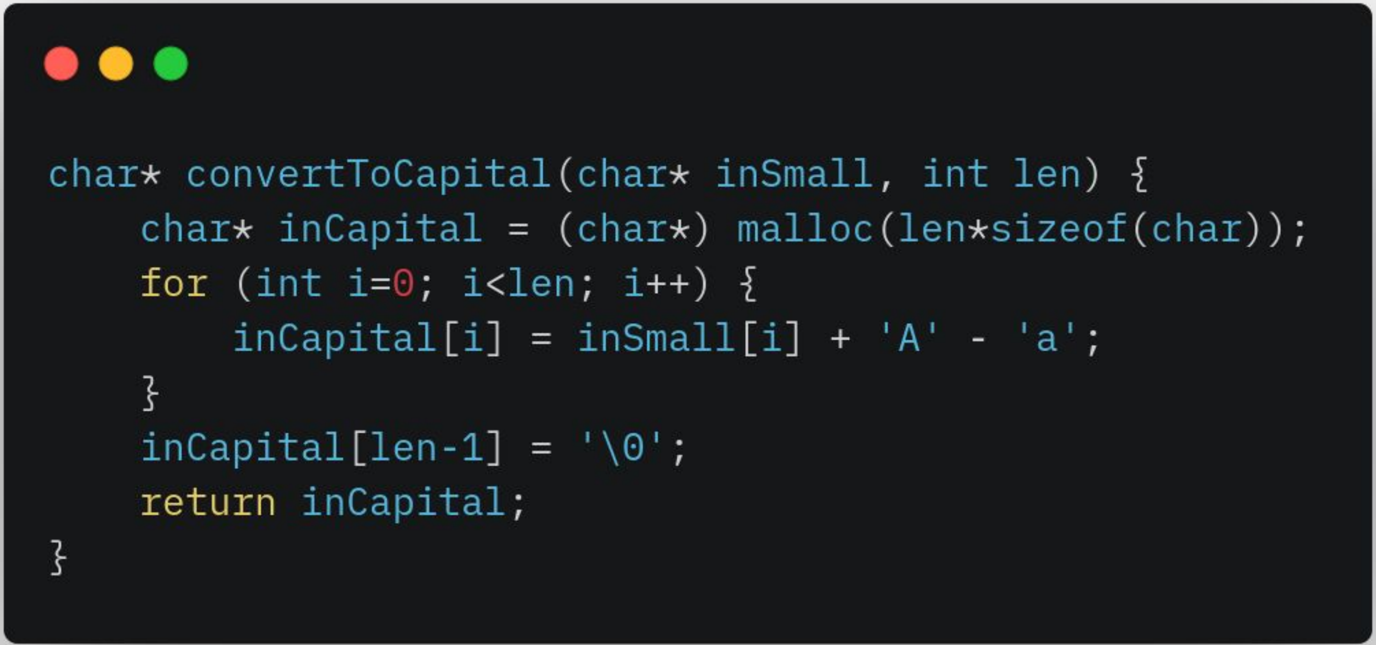
Pass by reference & value



```cpp
void passByValue(int x) {
    x = x + 1;
    cout << x << endl;
}
void passByReference(int& x) {
    x = x + 1;
    cout << x << endl;
}

int main() {
    int x = 1;
    passByValue(x);                         // 2
    cout << x << endl;                      // 1
    passByReference(x);                     // 2
    cout << x << endl;                      // 2
    return 0;
}
```



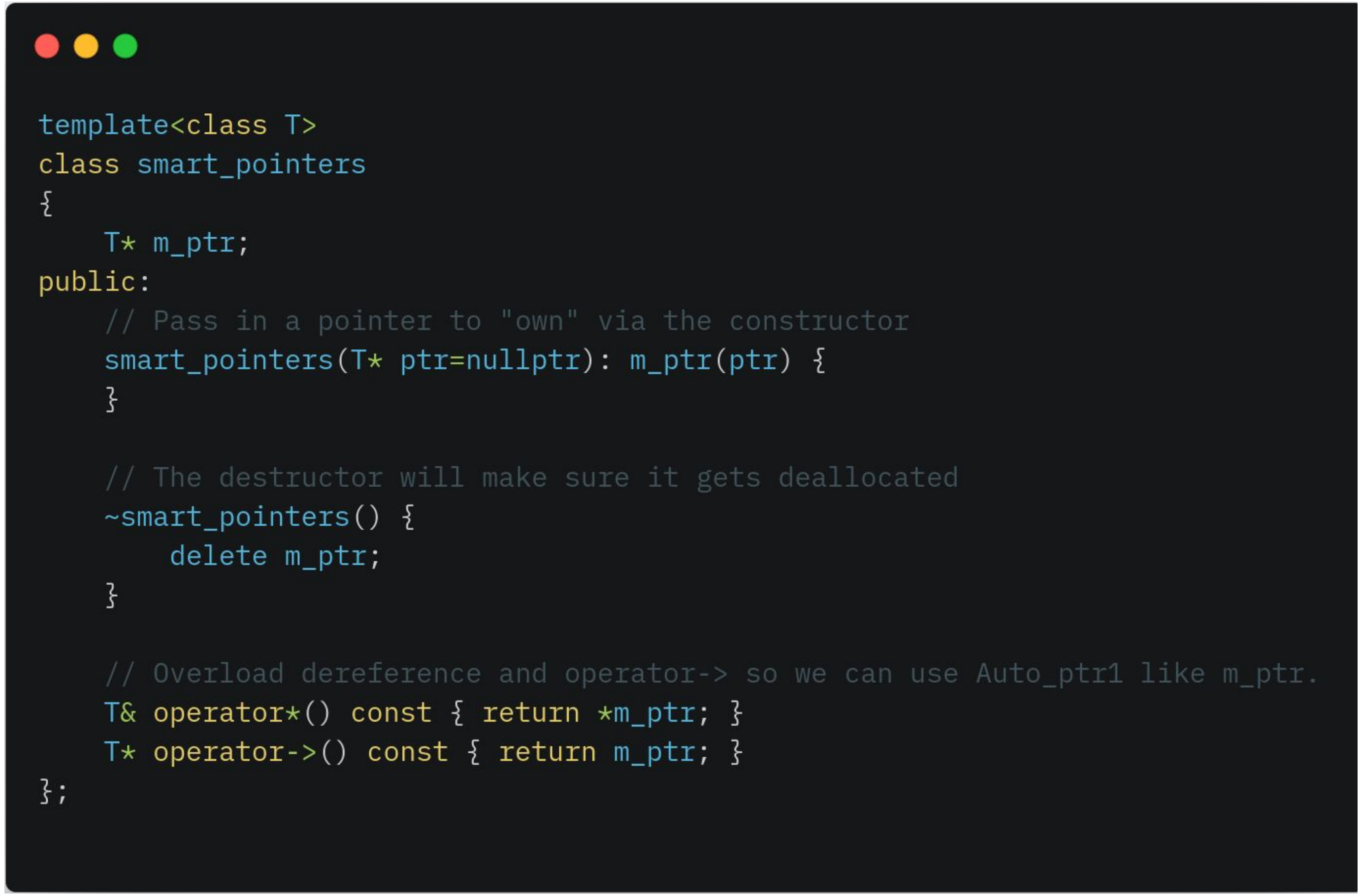
```
char* dontReallyConvertToCapital(char* inSmall, int len) {
    char inCapital[100];
    for (int i=0; i<len; i++) {
        inCapital[i] = inSmall[i] + 'A' - 'a';
    }
    inCapital[len-1] = '\0';
    return inCapital;
}
```



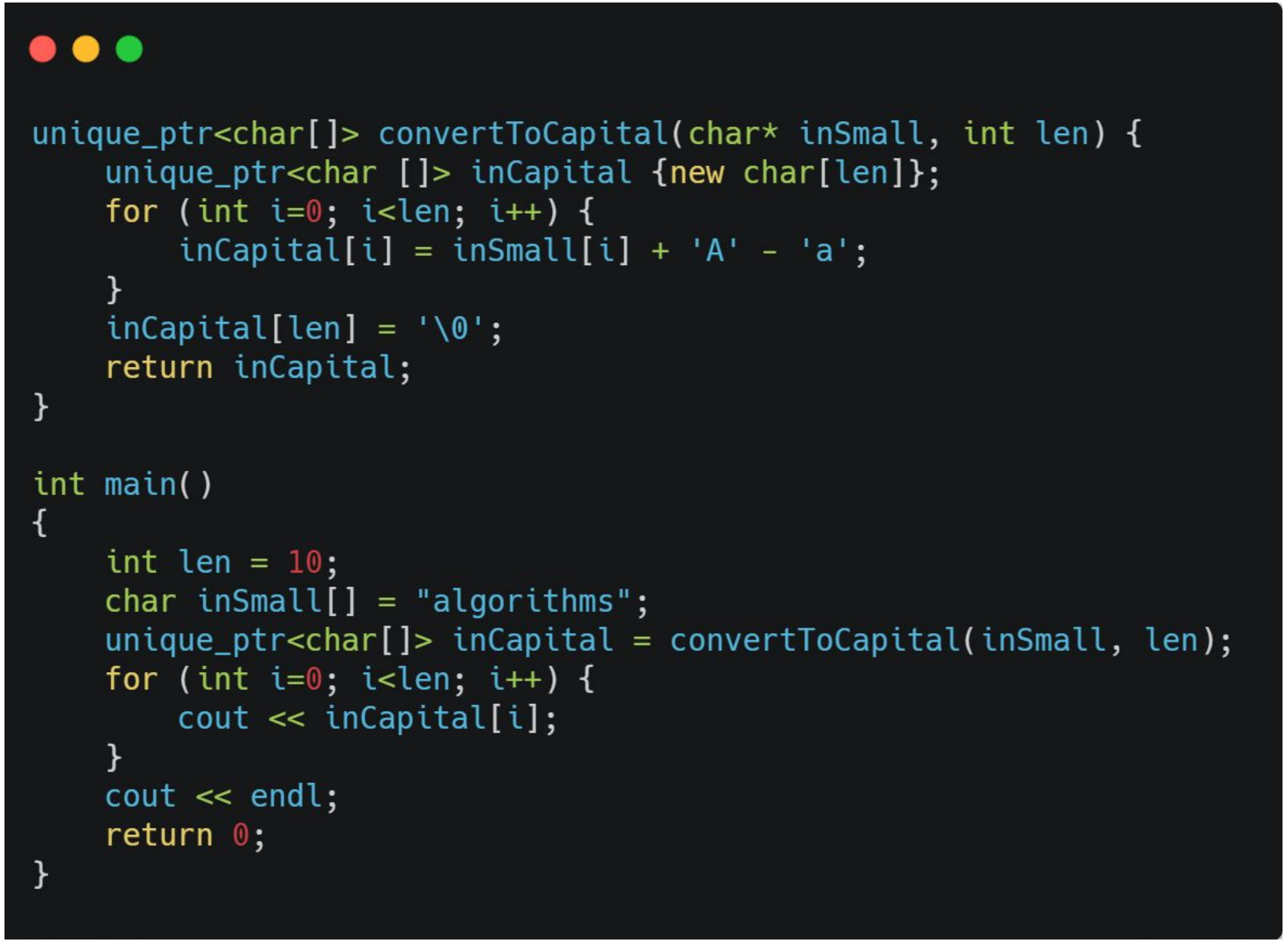
```c
char* convertToCapital(char* inSmall, int len) {
    char* inCapital = (char*) malloc(len*sizeof(char));
    for (int i=0; i<len; i++) {
        inCapital[i] = inSmall[i] + 'A' - 'a';
    }
    inCapital[len-1] = '\0';
    return inCapital;
}
```

Smart Pointers

- *char[]* is allocated dynamically on heap inside the function, and the memory will live even after the function exits.
- If we just allocate memory in start of function and delete by the end everything should be fine? well what about exceptions and early returns.
- Instead we come up with Smart Pointers follow pattern where the ownership of a heap-allocated object is with a stack-allocated object so that when its **destructor** is called, the deallocation of the dynamic memory can be done.
- RAII or Resource Acquisition Is Initialization



```
template<class T>
class smart_pointers
{
    T* m_ptr;
public:
    // Pass in a pointer to "own" via the constructor
    smart_pointers(T* ptr=nullptr): m_ptr(ptr) {
    }

    // The destructor will make sure it gets deallocated
    ~smart_pointers() {
        delete m_ptr;
    }

    // Overload dereference and operator-> so we can use Auto_ptr1 like m_ptr.
    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
};
```



```cpp
unique_ptr<char[]> convertToCapital(char* inSmall, int len) {
    unique_ptr<char []> inCapital {new char[len]};
    for (int i=0; i<len; i++) {
        inCapital[i] = inSmall[i] + 'A' - 'a';
    }
    inCapital[len] = '\0';
    return inCapital;
}

int main()
{
    int len = 10;
    char inSmall[] = "algorithms";
    unique_ptr<char[]> inCapital = convertToCapital(inSmall, len);
    for (int i=0; i<len; i++) {
        cout << inCapital[i];
    }
    cout << endl;
    return 0;
}
```

What’s next?

- Move Semantics
- https://www.learncpp.com/
- https://en.cppreference.com/w/
- CppCon, C++ Weekly With Jason Turner - YouTube