Object Oriented Basics

Topics:

- Classes and objects
- Constructors and destructors
- Inheritance

Classes and Objects

Fundamentally, an object is a collect of data and methods that you can invoke on it. Let's create an object in C, and define APIs on it:

```
typedef struct {
  int age;
} person_s;
void person__set_age(person_s* pointer, int age) {
    pointer->age = age;
}
void c_usage() {
    person_s person;
    person__set_age(&person, 123);
    person.age = 456; // C has no concept of private/public}
```

The C API has the following limitations:

- `struct` members (ie: age), are "public"
- This means that anyone can access the members of `person_s`
- You manually have to create a function whose first parameter is `person_s *pointer` such that this function can modify the object in a mutable way

C++

In C++, the limitations are overcome, and the syntax becomes:

```
class person_class {
  int m_age;

  public:
    void set_age(int age) {
      m_age = age;
    }
} ;
void usage() {
  person_class person;
  person.set_age(123);

  // p.set_age(123) really resolves to:
  //person::set_age(&person, 123);
  // cannot access private
  //person.m_age = 123;}
```

In a way, C++ just adds a little syntax sugar to achieve the following:

- Class has default visibility of private, hence m_age (as in member variable age) is private
- Public API doesn't need the mutable pointer passed in, it is automatic
  - There is a hidden "this" pointer as a first parameter

Constructors

One of the severe limitations of C is that constructors are not automatic. Let's find out what that means:

```
class person_class {
  int m_age;

  public:
    // "default" constructor
    person_class() {
      std::cout << "Constructor of person_class has been called" << std::endl;
      m_age = 0;
    }
    void set_age(int age) {
```

```
      m_age = age;
    }};
```

Default constructor is one you get for "free", and you may not always need to define it, especially if the default constructor has empty code. However, if you specify another constructor with different parameters, default constructor is deleted.

```
class person_class {
  int m_age;

  public:
    // constructor
    person_class(int age) {
      std::cout << "Constructor of person_class has been called" << std::endl;
      m_age = age;
    }
    void set_age(int age) {
      m_age = age;
    }};
```

In the code above, this effectively yields this syntax:

```
class person_class {
  int m_age;

  public:
    person_class() = delete;

    // constructor
    person_class(int age) {
      std::cout << "Constructor of person_class has been called" << std::endl;
      m_age = age;
    }};
```

Destructors

Destructors are intuitively opposite of the constructors. Unlike a constructor when the function is invoked when the object is built, the destructor is called when the object goes out of scope and is thus destroyed.

```
class person_class {
  int m_age;

  public:
    ~person_class() {
      std::cout << "Destructor of person_class has been called" << std::endl;
    }};
```

Practical Example

```
class Vector {
private:
    int* m_array;  // Pointer to dynamically allocated array
    int m_size;    // Size of the vector
public:
    // Constructor with size and default value
    Vector(int size) {
        m_size = size;
        m_array = new int[size];
        std::cout << "Vector constructor called. Size: " << size << std::endl;
    }
    // Destructor
    ~Vector() {
        delete[] m_array;
        std::cout << "Vector destructor called. Size: " << size << std::endl;
    }};
```

Exercise 1

Let's put all the knowledge acquired so far towards an exercise. We will build a simpler version of the std::vector, or simply an integer array.

```
// file: vector.hh
class Vector {
private:
    int* m_array;   // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
    int m_size;     // Current size of the vector
public:
    // Constructor with max size
    Vector(int max_size);
    ~Vector();

    bool push_back(int value);
    int pop_back();

    int back();
    int front();

    int get_size();
    int get_max_size();

    void clear();};
```

Self-test framework

Ideally, we would create a unit-test framework, but to keep things simple, we can use the assert() API to provide a rudimentary unit-test framework.

```
#include <iostream>
#include <assert>
class Vector;
int main() {
  Vector v(5);

  assert(0 == v.get_size());
  assert(3 == v.get_max_size());
  assert(0 == v.pop_back());
```

```
  assert(true == v.push_back(123));
  assert(1 == v.get_size());
  assert(123 == v.pop_back());
  // ...
  return 0;}
```

Copy Problem

There is a problem with our current design of the vector. The code below has an issue; please compile and run the code and see what happens!

```
#include <iostream>
#include <assert>
class Vector;
int main() {
  Vector v1(10);
  // We want another vector with same properties as v1
  // Problem: We did not allocate new memory but are now referring to v1's memory
  Vector v2 = v1;
  return 0;}
```

std::unique Pointer to the Rescue

After gaining advanced C++ experience, you will align to the fact that there should never be "naked pointers" in C++. Pointers should always use more advanced pointers provided by the C++ 11 standard. If we had built our vector like this, we would have caught the problem at compile-time rather than run-time. Note that the code below is just a preview of what we will learn in the future, and this is not required for the exercises as part of this article.

```
// file: vector.hh
#include <memory>

class Vector {
private:
    std::unique_ptr<int> m_array;   // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
```

```
    int m_size;     // Current size of the vector
public:
    // Constructor with max size
    Vector(int max_size);
    ~Vector();

    // ...};
```

Copy Constructor

The solution is that we need to "deep copy" the object which is called the copy constructor. Let's implement the copy constructor and see how it will work.

```
class Vector {
    // RULE: Whenever there is dynamic memory allocation (new operator)
    // There shall always be a copy constructor to perform "deep copy"
    Vector(const Vector& copy) {
        m_max_size = copy.m_max_size;
        // do not :
       //m_array_pointer = copy.m_array_pointer;
        m_array_pointer = new int[m_max_size]; // allocate your own memory, do not reference same memo
         std::cout << "Vector COPY constructor is called for size " << m_max_size << std::endl;
        // Deep copy
        for(int i = 0; i < m_max_size; i++) {
            m_array_pointer[i] = copy.m_array_pointer[i];
        }
    }

  // ...};
```

Rule of 3

The "Rule of Three" in C++ refers to a guideline for defining three specific member functions when a class manages resources like **dynamic memory** (e.g., through pointers) to ensure proper behavior regarding copying and destruction. The three key member functions are:

1.

Destructor (`~ClassName()`):

- The destructor is responsible for releasing resources (like dynamic memory) held by an object when it is destroyed.
- This is crucial to prevent memory leaks and properly clean up allocated resources.

2. **Copy Constructor (`ClassName(const ClassName& other)`):**
   - The copy constructor creates a new object as a copy of an existing object.
   - It is used when an object is initialized from another object of the same type (e.g., during object initialization, function parameter passing by value).

3. **Copy Assignment Operator (`ClassName& operator=(const ClassName& other)`):**
   - The copy assignment operator defines how an existing object can be assigned the value of another object of the same type.
   - It is invoked when you assign one object to another using the assignment operator `=`.

Sample Code for Rule of 3

```
class Vector {
  int *m_memory_for_integers;
  int m_max_size;
  int m_current_size;
  void deep_copy(const Vector& source) {
    // Deep copy: Copy each member from one vector to another
    for (int i=0; i < m_current_size; i++) {
      m_memory_for_integers[i] = source.m_memory_for_integers[i];
    }
  }
public:
  // Fixed size vector that allocates memory once but cannot grow (by design)
  Vector(int max_size) {
    printf("Constructor is called to allocate %d integers\n", max_size);
    m_max_size = max_size;
    m_memory_for_integers = new int[m_max_size]; // Allocate memory dynamically
  }
  // Rule of 3: If dynamic memory, then:
  // - We must destructor
  // - Copy consrructor
  // - Assignment operator
```

```
  // 1: Destructor
  ~Vector() {
    std::cout << ("Destructor is called") << std::endl;
    delete [] m_memory_for_integers;
  }
#if 0 /* BUGGY CODE: */
  // You get trivial copy constructor for free:
  // But if you allocate dynamic memory, is this what you want? No!
  Vector(const Vector& source) {
    m_max_size = source.m_max_size;
    m_current_size = source.m_current_size;
    // BUGGY CODE:
    m_memory_for_integers = source.m_memory_for_integers; // THIS IS THE PROBLEM
  }
#endif
  // 2: Copy constructor
  Vector(const Vector& source) {
    printf("Copy constructor called\n");
    m_max_size = source.m_max_size;
    m_current_size = source.m_current_size;
    // THIS IS THE KEY: Allocate our own memory
    // We do not wish to copy memory reference of another object literally
    m_memory_for_integers = new int[m_max_size];
    deep_copy(source);
  }
  // 3: Assignment operator
  Vector& operator=(const Vector& source) {
    printf("Assignment Operator called\n");

    m_max_size = source.m_max_size;
    m_current_size = source.m_current_size;
    m_memory_for_integers = new int[m_max_size]; // THIS IS THE KEY
    deep_copy(source);
    return *this;
  }
  // Const APIs that do not modify the Vector instance
```

```
  int get_size() const { return m_current_size; }
  int get_max_size() const { return m_max_size; }
  void print_memory_location_of_integers() const {
    printf("Memory allocated at\n");
    for(int i = 0; i < m_max_size; i++) {
      printf(" [%d] = %p\n", i, &m_memory_for_integers[i]);
    }
  }
};
```

Exercise 2

Based on our learning so far, let us perform another exercise to create a string library.

```
#include <string.h>
#include <iostream>
// String library
class string {
  char *m_string;
  int m_max_length;

  // Bonus points if you use unique_pointer
  // std::unique_ptr<char> m_string;
  public:
    string(int max_length) {
        m_max_length = max_length + 1; // +1 for NULL termination
        m_string = new char[m_max_length];
    }
    // Rule of 3: Because we will allocate memory dynamically
    ~string() {
        delete [] m_string;
    };
    // Implement constructor to allocate as much memory as the c_string
    string(const char *c_string);
    // Rule of 3: 2) Implement the copy constructor
```

```
    string(const string& copy);
    // Rule of 3: 3) Implmement the assignment operator
    string& operator=(const string& source);

    // Mutable API to make all characters lowercase or uppercase
    void to_upper();
    void to_lower();

    // Adding more data to the string
    void append_char(char c); // Append a char but only if string has memory available
    void append_string(const char *c_string); // Append another c_string only if string has memory avai

    // Non-mutable APIs to check certain properties of the string
    bool equals_to(const char *c_string);
    bool contains(const char *c_string);
    bool begins_with(const char *c_string);
    int get_length();
    void print() { std::cout << "String is: '" << m_string << "'" << std::endl; }

    // Other mutable APIs
    void clear();
    void set(const char *string) {
        strncpy(m_string, string, m_max_length);
        // All 3 lines do the same thing
        m_string[m_max_length - 1] = '\0';
        m_string[m_max_length - 1] = 0;
        //m_string[m_max_length - 1] = NULL;
        // We need the line(s) above because in case strncpy() ran out of space, it won't null termina
        // "hello" -> 6 spaces
        // [0] = h, [1] = e, [2] = l, [3] = l, [4] = o, [5] = '\0';
    }
};
```

Copy Constructor

The string class you built above has the same problem for the deep copy. Therefore, you will need to create a copy constructor to be able to deep copy the string.

Pay close attention to this code:

```
class string {
    // Rule of 3: 1) Functional destructor to deallocate dynamically allocated memory
    ~string() {
        delete [] m_string;
    };
    // Rule of 3: 2) Implement the copy constructor
    string(const string& copy);
    // Rule of 3: 3) Implmement the assignment operator
    string& operator=(const string& source);}
```

Header and Source File

<TODO: Template of header and source file>

Revision #15
Created 2 months ago by Preet Kang
Updated 1 month ago by Preet Kang