CSE 208L – Object Oriented Programming Lab
Fall 2019

Chapter 5: Object Programming Essentials [Part 1]

Notes By
Engr. Sumayyea Salahuddin
DCSE, UET Peshawar

Networking Academy

Chapter 5 Objectives

After completing this module, the student will be able to:

1) Understand the concept of C++ class and object syntax
2) Create objects
3) Access object members
4) Create setter/getter access methods
5) Limit the range of accepted values in access methods
6) Understand and implement different strategies for obtaining derived data
7) Define a C++ class from scratch
8) Model real-world entities with classes and objects
9) Limit acceptable input range
10) Manage multiple objects
11) Provide meaningful and helpful representation of objects

Chapter 5 Objectives [Cont.]

12) Understand the interactions between objects of the same type
13) Create objects based on objects of other objects of custom classes (don't worry - this sounds more difficult than actually is)
14) Understand the concept of inheritance syntax and operation
15) Share functionality between objects using inheritance
16) Implement data structures in C++
17) Understand the concept of dynamic allocation of C++ objects
18) Prevent memory leaks and de-allocate acquired resources
19) Provide derived data about the implemented data structure
20) Keep the data structure consistent at all times
21) Traverse data structures
22) Access data stored in data structures
23) Create copies of data structures
24) Implement and use copy constructors

5 Object Programming Essentials

5.1 Basic concepts of object programming

5.2 A stack: a view from two different perspectives

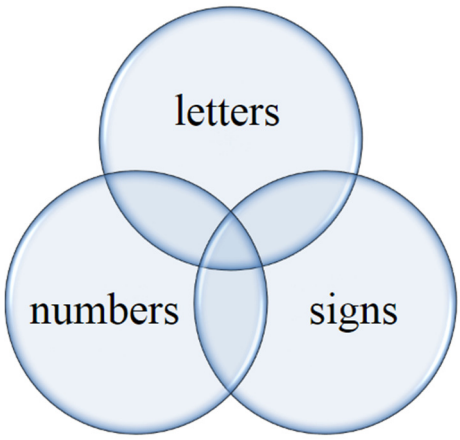
5.3 Anatomy of the class

5.4 Static components

5.5 Objects vs. pointers and objects inside the objects

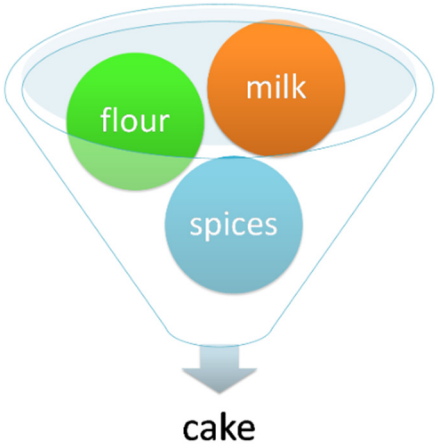
Networking Academy

5.1.1 Classes and objects in real life (1)

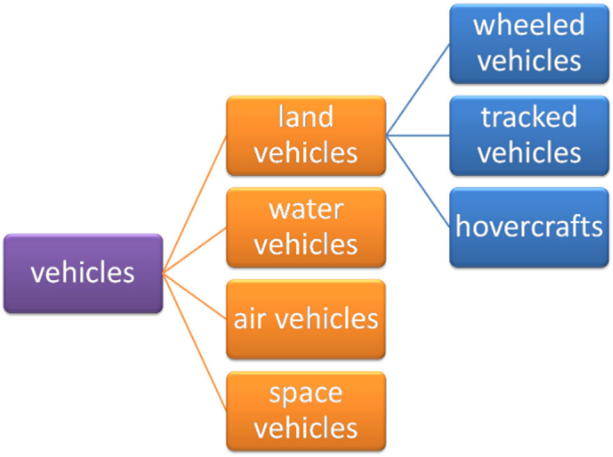


Networking Academy

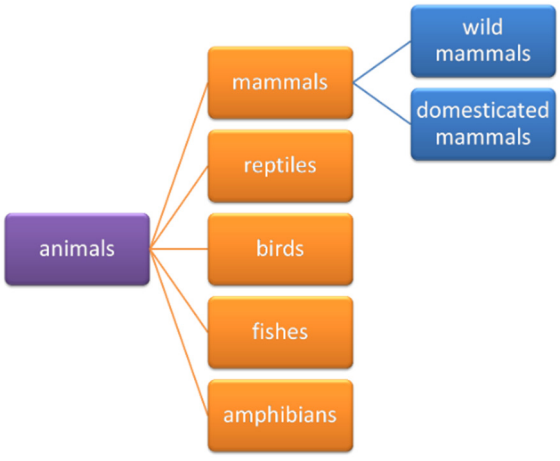
5.1.2 Classes and objects in real life (2)



5.1.3 Class – what is it? (1)

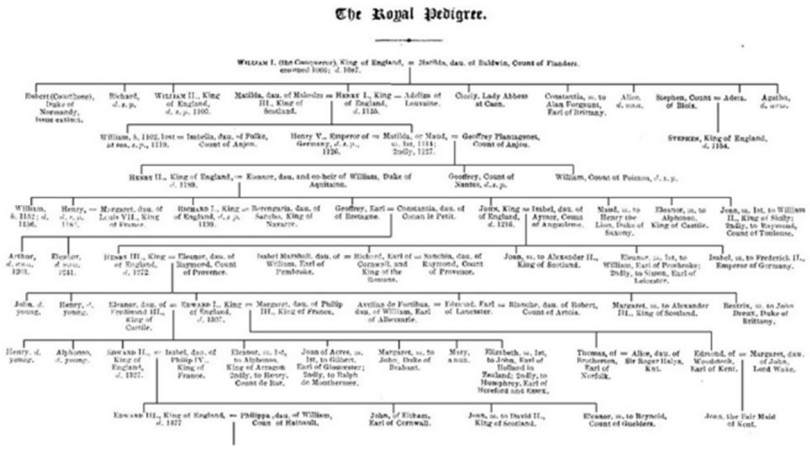


5.1.4 Class – what is it? (2)



5.1.5 Object – what is it?

5.1.6 Inheritance



Networking Academy

5.1.7 What does any object have?

1. an object has a name that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
2. an object has a set of individual properties that make it original, unique or outstanding (although there is the possibility that some objects may have no properties at all)
3. an object has a set of abilities to perform specific activities that can change the object itself or some of the other objects

- a noun, you probably define the object's name
- adjective, you probably define the object's property
- a verb, you probably define the object's activity

"A pink Cadillac went quickly"
Object name = Cadillac
Home class = Wheeled vehicles
Property = Colour (pink)
Activity = Drive (quickly)

Networking Academy

5.1.8 Why all this?

- Class is a model of a very specific part of reality reflecting properties and activities found in the real world.
- The new class may add new properties and new activities and therefore may be more useful in specific applications.
- The existence of a class doesn't mean that any of the compatible objects will be automatically created. The class itself isn't able to create an object – you have to create it yourself.

```
class OurClass { };
```

5.1.9 The very first object

```
OurClass our_object;
```


5 Object Programming Essentials

5.1 Basic concepts of object programming

5.2 A stack: a view from two different perspectives

5.3 Anatomy of the class

5.4 Static components

5.5 Objects vs. pointers and objects inside the objects

5.2.1 Stack aka LIFO (1)

- A **stack** is a structure developed **to store data** in a very specific way
- Example: A stack of coins
- You can't put a coin anywhere else but on the top of the stack.
- You can't get a coin off the stack from anywhere other than the stack's top. If you want to get the coin on the bottom, you have to remove all the coins that are sitting on top of it.
- Alternate Name: "**Last In – First Out**" (LIFO).
- Two elementary operations conventionally named "**push**" (when a new element is placed on the top) & "**pop**" (when an existing element is taken away from the top).

5.2.2 Stack aka LIFO (2)

```
int stack[100];
```

Networking Academy

5.2.3 Stack pointer

```
int SP =0;
```

A variable that can be responsible for storing **a number of elements currently stored** on the stack. This variable is generally called a "**stack pointer**", or **SP** for short.

Networking Academy

5.2.4 Push

We're now ready to define a function that **places a value** onto the stack. Here is what we've supposed:

- the **name** for the function is "push"
- the function gets **one parameter** of type *int* (this is the value to be placed onto the stack)
- the function **returns nothing** (self-explanatory, right?)
- the function **places the parameter** value into the first free element in the vector and **increments the SP**

```
void push(int value) {
    stack[SP++] = value;
}
```

Networking Academy

5.2.5 Pop

Now it's time for the function to **take a value** off the stack. This is how we imagine it:

- the **name** of the function is "pop" (we don't want to discover America again)
- the function **doesn't have any parameters**
- the function **returns the value** taken from the stack
- the function **reads the value** from the stack's top and **decrements SP**

```
int pop(void) {
    return stack[--SP];
}
```


Networking Academy

5.2.6 The stack in action

```
#include <iostream>

using namespace std;

int stack[100];
int SP = 0;

void push(int value) {
    stack[SP++] = value;
}

int pop(void) {
    return stack[--SP];
}
```

```
int main(void) {
    push(3);
    push(2);
    push(1);
    cout << pop() << endl;
    cout << pop() << endl;
    cout << pop() << endl;
    return 0;
}
```

The program outputs the following text to the screen:

```
1
2
3
```

Networking Academy

5.2.7 Pros and cons [Part 1]

- two essential variables (stack and SP) are **completely vulnerable; anyone can modify them in an uncontrollable way**, destroying the stack;
- need more than one stack;
- it may also happen that you need not only *push* and *pop* functions, but also some other things; you can implement them but try to imagine what'll happen when you have dozens of separately implemented stacks
- use the stacks defined for other types: floats, strings or even arrays and structures; what'll happen then?

Networking Academy

5.2.7 Pros and cons [Part 2]

- the ability to hide (protect) selected values against unauthorized access is called **encapsulation**; the encapsulated values can be neither accessed nor modified if you want to use them exclusively
- when you have a class implementing all the needed stack behaviours, you can produce as many stacks as you want; you don't need to copy or replicate any part of the code
- the ability to enrich the stack with new functions comes from the **inheritance**; you can create a new class (or more precisely a **subclass**) which inherits all the existing traits from the superclass and adds some new ones
- you can create a **template** which is a **generalized, parameterized class**, ready to materialize itself in many different incarnations; its code can adapt to varying requirements and, for example, create stacks ready to work with other types of data

5.2.8 Stack from scratch (1)

```
class Stack {
   int stackstore[100];
   int SP;
  };
```


5.2.9 Stack from scratch (2)

```
class Stack {
private:
   int stackstore[100];
   int SP;
};
```

Networking Academy

5.2.10 Stack from scratch (3)

```
class Stack {
private:
    int stackstore[100];
    int SP;
public:
    void push(int value);
    int pop(void) {
        return stackstore[--SP];
    }
};
```


Networking Academy

5.2.11 Stack from scratch (4)

```
class Stack {
  private:
    int stackstore[100];
    int SP;
  public:
    void push(int value);
    int pop(void) {
      return stackstore[--SP];
    }
};

void Stack::push(int value) {
    stackstore[SP++] = value;
}
```

Networking Academy

5.2.12 Stack from scratch (5)

```
class Stack {
  private:
    int stackstore[100];
    int SP;
  public:
    Stack(void) { SP = 0; }
    void push(int value);
    int pop(void) {
        return stackstore[--SP];
    }
};

void Stack::push(int value) {
    stackstore[SP++] = value;
}
```


Networking Academy

5.2.13 Stack from scratch (6)

```
#include <iostream>

using namespace std;

int main(void) {
    Stack little_stack, another_stack, funny_stack;

    little_stack.push(1);
    another_stack.push(little_stack.pop() + 1);
    funny_stack.push(another_stack.pop() + 2);
    cout << funny_stack.pop() << endl;
    return 0;
}
```

5.2.14 Stack from scratch (7)

```
class AddingStack: Stack {
};
```

- **a new stack with new capabilities**.
- construct a **subclass** of the *Stack* class.
- **It derives all the components defined by its superclass** .
- Any object of the *AddingStack* class can do everything that each *Stack* class' object does.
- We want the push function not only to push the value onto the stack, but also to add the value to the *sum* variable.
- We want the pop function not only to pop the value off the stack, but also to subtract the value from the *sum* variable

5.2.15 Stack from scratch (8)

```
class AddingStack : Stack {
  private:
    int sum;
  public:
    void push(int value);
    int pop(void);
};
```

5.2.16 Stack from scratch (9)

```
void AddingStack::push(int value) {
    sum += value;
    Stack::push(value);
}
```


5.2.17 Stack from scratch (10)

```
int AddingStack::pop(void) {
    int val = Stack::pop():
    sum -= val;
    return val;
}
```

5.2.18 Stack from scratch (11)

```
int AddingStack::getSum(void) {
    return sum;
}
```


5.2.19 Stack from scratch (12)

```
AddingStack::AddingStack(void) : Stack() {
    sum = 0;
}
```

The initial value of the *sum* variable should be **zeroed** when the object is created.

Networking Academy

5.2.20 Stack from scratch (13)

```
class AddingStack : Stack {
  private:
    int sum;
  public:
    AddingStack(void);
    void push(int value);
    int pop(void);
    int getSum(void);
};
AddingStack::AddingStack(void) : Stack() {
    sum = 0;
}
void AddingStack::push(int value) {
    sum += value;
    Stack::push(value);
}
int AddingStack::pop(void) {
    int val = Stack::pop();
    sum -= val;
    return val;
}
int AddingStack::getSum(void) {
    return sum;
}
```


Networking Academy

5.2.21 Stack from scratch (14)

```
#include <iostream>

using namespace std;

int main(void) {
    AddingStack super_stack;

    for(int i = 1; i < 10; i++)
        super_stack.push(i);
    cout << super_stack.getSum() << endl;
    for(int i = 1; i < 10; i++)
        super_stack.pop();
    cout << super_stack.getSum() << endl;
    return 0;
}
```

5 Object Programming Essentials

5.1 Basic concepts of object programming

5.2 A stack: a view from two different perspectives

5.3 Anatomy of the class

5.4 Static components

5.5 Objects vs. pointers and objects inside the objects

5.3.1 Class Components

A class is an **aggregate** consisting of **variables** (also called fields or properties) and functions (sometimes called methods). Both variables and functions are class **components**.

The class on the right → has three components: one variable of type *int* called *value* and two functions called *setVal* and *getVal* respectively. The class is named *Class*.

Since all the components are declared without the use of an **access specifier** (neither a *public* nor *private* keyword was added among the declarations) all three components are **private**.

```
class Class {
    int value;
    void setVal(int value);
    int getVal(void);
};
```

5.3.2 Access Specifiers

- The *setVal* and *getVal* components are **public** – they're accessible to all users of the class. The *value* component is **private** – it's accessible only within the class.

```
class Class {
public:
        void setVal(int value);
        int getVal(void);
private:
        int value;
};
```


5.3.3 Creating an Object

```
Class the_object;
```

Networking Academy

5.3.4 Overriding Component Names

- If any function introduces an entity of the name identical to any of the class components, the name of the class component is overridden.
- It can be accessed only by the qualification with the home class name.
- The *setVal* function uses a parameter called *value*. The parameter overrides the class component called *value.*

```
class Class {
public:
   void setVal(int value) {
        Class::value = value;
   }
   int getVal(void);
private:
   int value;
};
```


Networking Academy

5.3.5 "this" Pointer

- its name is **this**
- it mustn't be declared explicitly (it's a keyword) so **it may not be overridden**
- it's **a pointer to the current object** – each object has its own copy of the *this* pointer

The general rule says that:

- if *S* is a **structure** or class and *S* has a **component** named *C* and
- if *p* is a pointer to a structure of type *S*
- then the *C* component may be accessed in the two following ways:

(*p).C // *p* is explicitly dereferenced in order to access the *C* component

p->C // *p* is implicitly dereferenced in order to access the *C* component

```
class Class {
public:
   void setVal(int value) {
       this -> value = value;
   }
   int getVal(void);
private:
   int value;
};
```

5.3.6 Qualifying Component Names (1)

- If any class function body is given outside the class body, its name must be qualified with the home class name and the "**::**" operator.
- A function defined in this way has the same full access to all class components as any function defined inside the class.
- All rules for overriding names are valid too.

```
class Class {
public:
  void setVal(int value) {
       this -> value = value;
  }
  int getVal(void);
private:
  int value;
};

int Class::getVal(void) {
    return value;
}
```

Material taken from CPA: Programming Fundamentals in C++ 43

5.3.7 Qualifying Component Names (2)

```
class Class {
public:
      void setVal(int value) { this -> value = value;  }
      void setVal(void) { value = -2; }
      int getVal(void) { return value; }
private:
      int value;
};
```

Material taken from CPA: Programming Fundamentals in C++ 44

5.3.8 Constructors

```
class Class {
public:
    Class(void) { this -> value = -1; }
    void setVal(int value) { this -> value = value; }
    int getVal(void) { return value; }
private:
    int value;
};
```


5.3.9 Overloading Constructor Names (1)

The actual constructor is chosen during the object's creation. Look at the following snippet, please:

```
Class object1, object2(100);
cout << object1.getVal() << endl;
cout << object2.getVal() << endl;
```

The *object1* object will be created using a parameter-less constructor, while the *object2* will be created with a one-parameter constructor.

The snippet will output the following values:

```
-1
100
```

```
class Class {
public:
    Class(void) { this -> value = -1; }
    Class(int val) { this -> value = val; }
    void setVal(int value) { this -> value = value; }
    int getVal(void) { return value; }
private:
    int value;
};
```

5.3.10 Overloading constructor names (2)

- If a class has a constructor (or more precisely, at least one constructor), **one of them must be chosen** during object creation i.e. you're not allowed to write a declaration which doesn't specify a target constructor.

```
class Class {
public:
    Class(int val) { this -> value = val; }
    void setVal(int value) { this -> value = value; }
    int getVal(void) { return value; }
private:
    int value;
};
```


5.3.11 Copying Constructors

```
#include <iostream>
using namespace std;
class Class1 {
public:
    Class1(int val) { this -> value = val; }
    Class1(Class1 const &source) { value = source.value + 100; }
    int value;
};
class Class2 {
public:
    Class2(int val) { this -> value = val; }
    int value;
};
int main(void) {
    Class1 object11(100), object12 = object11;
    Class2 object21(200), object22 = object21;
    cout << object12.value << endl;
    cout << object22.value << endl;
    return 0;
}
```

5.3.12 Memory Leaks

Many of the objects are allocated memory that they need for their operation. This memory should be released when the object finishes its activity and the best way to do this is to do the cleaning automatically. Failure to clean the memory will cause a phenomenon named "**memory leaking**", where the unused (but still allocated!) memory grows in size, affecting system performance.

```
#include <iostream>
using namespace std;
class Class {
public:
    Class(int val) {
        value = new int[val];
        cout << "Allocation (" << val << ") done." << endl;
    }
    int *value;
};
void MakeALeak(void) {
    Class object(1000);
}
int main(void) {
    MakeALeak();
    return 0;
}
```


5.3.13 Destructors

We can safeguard ourselves against this danger by defining a special function called **destructor**. Destructors have the following restrictions:

- if a class is named X, its destructor is named ~**X**
- a class can have **no more than one destructor**
- a destructor **must be a parameter-less function** (note that the two last restrictions are the same – can you explain why?)
- a destructor shouldn't be invoked explicitly

```
#include <iostream>
using namespace std;
class Class {
public:
    Class(int val) {
        value = new int[val];
        cout << "Allocation (" << val << ") done." << endl;
    }
    ~Class(void) {
        delete [] value;
        cout << "Deletion done." << endl;
    }
    int *value;
};
void MakeALeak(void) {
    Class object(1000);
}
int main(void) {
    MakeALeak();
    return 0;
}
```

Task 1

1. Write the CP Stack Code given on Slide#20 in Code::Blocks and show results in next week lab.
2. Write the OOP Stack Code given on Slide#27-28 in Code::Blocks and show results in next week lab.
3. Write the updated OOP Stack Code given on Slide#35-36 in Code::Blocks and show results in next week lab.

51

Conclusion

- Discussed object programming concepts
- Discussed an example of stack using both structural/procedural way and object-oriented way
- Discussed the anatomy of class

52

Next Week

- Static Components and pointers will be discussed next time