COMP 322: Introduction to C++

Chad Zammar, PhD
Feb 10, 2023

Lecture 6
(Classes in C++)

- Review of OO concept
- C structs
- C++ Classes
- Constructors
- Destructors

OO programming: Quick review

- Approach to design modular and reusable systems
- Programming is about manipulating data through code (methods or algorithms)
  - Object: is a coupling of both data and methods
- Extension to the concept of structures
  - Not only we group multiple data elements but we also attach the intelligence needed to manipulate the data (also called encapsulation)

C structs: compound data type

```
15 struct person
16 {
17     int age;
18     char sex;
19 };
20
21 // main function
22 int main()
23 {
24     person Mike;
25     Mike.age = 24;
26     Mike.sex = 'M';
27
28     cout << Mike.age << endl;
29     return 0;
30 }
```

C structs: data and code are separated

```
15 struct person
16 {
17     int age;
18     char sex;
19 };
20
21 bool canVote(int age)
22 {
23     if (age >= 18)
24         return true;
25     return false;
26 }
27
28 // main function
29 int main()
30 {
31     person Mike;
32     Mike.age = 24;
33     Mike.sex = 'M';
34
35     if (canVote(Mike.age))
36         cout << "Mike is eligible to vote" << endl;
37     else
38         cout << "too bad for Mike" << endl;
39     return 0;
40 }
```

C++ Classes: introduction

```
17 class Person
18 {
19 public:
20     bool canVote()
21     {
22         if (age >= 18)
23             return true;
24         return false;
25     }
26
27     int age;
28     char sex;
29 };
30
31 // main function
32 int main()
33 {
34     Person Mike;
35     Mike.age = 24;
36     Mike.sex = 'M';
37
38     if (Mike.canVote())
39         cout << "Mike is eligible to vote" << endl;
40     else
41         cout << "too bad for Mike" << endl;
42     return 0;
43 }
```

- Class is a user defined type
- It has data and methods
  - Referred to as members of the class
  - Methods are functions which are part of the class
- Members may have different access levels
  - public , private or protected
  - private by default if not specified
  - Public members are called interface of the class
- Instances of a class are called objects
  - Mike is an object of class Person

C++ Classes: access levels (public)

```
17 class Person
18 {
19 public:
20     bool canVote()
21     {
22         if (age >= 18)
23             return true;
24         return false;
25     }
26
27     int age;
28     char sex;
29 };
30
31 // main function
32 int main()
33 {
34     Person Mike;
35     Mike.age = 24;
36     Mike.sex = 'M';
37
38     if (Mike.canVote())
39         cout << "Mike is eligible to vote" << endl;
40     else
41         cout << "too bad for Mike" << endl;
42     return 0;
43 }
```

- Access levels are also called access specifiers or access modifiers
- Public
  - Accessible from anywhere (inside and outside the class)
  - Public is not a default mode so you need to explicitly specify it
  - Public methods are the interface of the class
    - This is what the clients can use to manipulate the inner state of the object

C++ Classes: access levels (private)

```
17 class Person
18 {
19 public:
20     bool canVote()
21     {
22         if (age >= 18)
23             return true;
24         return false;
25     }
26 private:
27     int age;
28     char sex;
29 };
30
31 // main function
32 int main()
33 {
34     Person Mike;
35     Mike.age = 24; // ERROR
36     Mike.sex = 'M';// ERROR
37
38     if (Mike.canVote())
39         cout << "Mike is eligible to vote" << endl;
40     else
41         cout << "too bad for Mike" << endl;
42     return 0;
43 }
```

- Private
  - Accessible only from within the class
  - Private is a default mode so you don't necessarily need to explicitly specify it
  - Friend functions are allowed to access private members (we will discuss this later)
  - Best practice is to define data as private and provide a public interface to access the data
  - Used to achieve data hiding

C++ Classes: access levels (private)

```
17 class Person
18 {
19 public:
20     bool canVote()
21     {
22         if (age >= 18)
23             return true;
24         return false;
25     }
26     void setAge(int age)
27     {
28         this->age = age;
29     }
30     void setSex(char sex)
31     {
32         this->sex = sex;
33     }
34 private:
35     int age;
36     char sex;
37 };
```

```
39 // main function
40 int main()
41 {
42     Person Mike;
43     Mike.setAge(24);
44     Mike.setSex('M');
45
46     if (Mike.canVote())
47         cout << "Mike is eligible to vote" << endl;
48     else
49         cout << "too bad for Mike" << endl;
50     return 0;
51 }
```

C++ Classes: access levels (protected)

```
17 class Person
18 {
19 public:
20     bool canVote()
21     {
22         if (age >= 18)
23             return true;
24         return false;
25     }
26     void setAge(int age)
27     {
28         this->age = age;
29     }
30     void setSex(char sex)
31     {
32         this->sex = sex;
33     }
34 protected:
35     int age;
36     char sex;
37 };
```

- Protected
  - Accessible from within the class
  - Accessible also from derived classes (we will discuss this later)
  - Protected is not a default mode so you need to explicitly specify it
  - Friend functions are allowed to access protected members (we will discuss this later)
  - Used to achieve data hiding

C++ Classes: method definition

```
17 class Person
18 {
19 public:
20     bool canVote()
21     {
22         if (age >= 18)
23             return true;
24         return false;
25     }
26     void setAge(int age)
27     {
28         this->age = age;
29     }
30     void setSex(char sex)
31     {
32         this->sex = sex;
33     }
34 protected:
35     int age;
36     char sex;
37 };
```

- Member methods can be defined within the class definition or outside of the class
- Methods defined inside the class declaration are considered “inline” methods even without the use of the “inline” keyword

C++ Classes: method definition

```
17 class Person
18 {
19 public:
20     bool canVote();
21     void setAge(int age);
22     void setSex(char sex);
23 protected:
24     int age;
25     char sex;
26 };
27
28 bool Person::canVote()
29 {
30     if (age >= 18)
31         return true;
32     return false;
33 }
34 void Person::setAge(int age)
35 {
36     this->age = age;
37 }
38 void Person::setSex(char sex)
39 {
40     this->sex = sex;
41 }
```

- To define a method outside of the class, we need to declare it within the class, then we provide the implementation outside of the class using the scope operator ::
- Methods defined outside of the class declaration can still be declared “inline” but with the explicit use of the “inline” keyword

C++ classes: constructors

- When instantiating a class, a special method is implicitly called first
  - This method is the constructor
- Every class has a constructor (at least one)
- If a constructor is not provided by the programmer, the compiler will provide a default implicit constructor (that does basically nothing)
  - This is how the construction of the Person class from the previous example was possible

C++ classes: constructors

- The Constructor method:
  - is used to initialize the data members of a class
  - must have the same name as the class
  - must be declared public in general
    - There are some exceptions when implementing advanced design patterns
  - does not have a return type
    - Constructors don’t return values

C++ classes: constructors

```
17 class Person
18 {
19 public:
20     Person();
21     int  getAge();
22 protected:
23     int age;
24     char sex;
25 };
26
27 Person::Person()
28 {
29     this->age = 0;
30     this->sex = 'U';
31 }
32
33 int Person::getAge()
34 {
35     return age;
36 }
```

```
40 // main function
41 int main()
42 {
43     Person Mike;
44
45     cout << Mike.getAge() << endl;
46 }
```

- Constructor without parameters is called the default constructor
- Usually used to initialize the data members to default values

C++ classes: constructors

```
17 class Person
18 {
19 public:
20     Person();
21     Person(int age, char sex);
22     int  getAge();
23 protected:
24     int age;
25     char sex;
26 };
27
28 Person::Person()
29 {
30     this->age = 0;
31     this->sex = 'U';
32 }
33
34 Person::Person(int age, char sex)
35 {
36     this->age = age;
37     this->sex = sex;
38 }
```

```
47 // main function
48 int main()
49 {
50     Person Mike(24, 'M');
51
52     cout << Mike.getAge() << endl;
53 }
```

- Constructor can be personalized using parameters
- User can define as many different constructors as needed

Constructor: initialization list

```
class Person
{
public:
    Person();
    Person(int age, char sex);
    int getAge() {return age;};
protected:
    int age;
    char sex;
};

Person::Person():age(0), sex('U')
{
}

Person::Person(int age, char sex):age(age), sex(sex)
{
}
```

- Initialization is listed outside of the body of a constructor
- Initialization list is preferred to regular initialization because it yields better performance

C++ classes: destructors

- When an object gets out of scope, a special method is implicitly called
  - This method is the destructor
- Every class has a destructor (and only one)
- If a destructor is not provided by the programmer, the compiler will provide a default implicit destructor (that usually calls the destructor for each data member but will not delete dynamically allocated memory for you)

C++ classes: destructors

- **Destructor method:**
  - **is used to clean and liberate any resource that was being held by the object**
  - **must have the same name as the class preceded by the tilde ~ operator**
  - **must be declared public in general**
    - **There are some exceptions when implementing advanced design patterns**
  - **does not have a return type nor does it take parameters**
    - **Destructors don’t return values**

C++ classes: destructors

```
19 class Person
20 {
21 public:
22     // constructors
23     Person();
24     Person(int age, char sex);
25     Person(int age, char sex, char *name);
26     // destructor
27     ~Person();
28     int  getAge();
29     char* getName();
30 protected:
31     int age;
32     char sex;
33     char *name;
34 };
```

```
43 Person::Person(int age, char sex, char *name)
44 {
45     cout << "Constructor got called" << endl;
46     this->age = age;
47     this->sex = sex;
48     this->name = new char[strlen(name)];
49     strcpy(this->name, name);
50 }
51
52 Person::~Person()
53 {
54     cout << "Destructor got called" << endl;
55     delete [] this->name;
56 }
57
58 char* Person::getName()
59 {
60     return this->name;
61 }
```

C++ classes: destructors

```
43 Person::Person(int age, char sex, char *name)
44 {
45     cout << "Constructor got called" << endl;
46     this->age = age;
47     this->sex = sex;
48     this->name = new char[strlen(name)];
49     strcpy(this->name, name);
50 }
51
52 Person::~Person()
53 {
54     cout << "Destructor got called" << endl;
55     delete [] this->name;
56 }
57
58 char* Person::getName()
59 {
60     return this->name;
61 }
```

```
70 // main function
71 int main()
72 {
73     Person Mike(24, 'M', "Michael");
74
75     cout << Mike.getName() << endl;
76 }
```

```
Constructor got called
Michael
Destructor got called
```

C++ classes: dynamic allocation

```
class GPS
{
public:
    GPS(double altitude, double longitude, double latitude):
        altitude(altitude),
        longitude(longitude),
        latitude(latitude)
    {
        cout << "GPS Constructor" << endl;
    }

    ~GPS()
    {
        cout << "GPS Destructor" << endl;
    }

private:
    double altitude;
    double longitude;
    double latitude;
};

int main()
{
    cout << "Program started ..." << endl;
    GPS* gps;
    cout << "A GPS pointer was being declared but not allocated yet" << endl;
    gps = new GPS(10, 25, 14);
    cout << "GPS Object was being allocated" << endl;
    delete gps;
    cout << "GPS Object was being deleted" << endl;
}
```

```
Program started ...
A GPS pointer was being declared but not allocated yet
GPS Constructor
GPS Object was being allocated
GPS Destructor
GPS Object was being deleted
```

Classes - design pattern example

```
class Singleton
{
public:
    static Singleton& getUniqueInstance()
    {
        static Singleton instance;
        return instance;
    }
    // Add the needed public methods
    void doSomething() {}

private:
    Singleton(){}
    ~Singleton(){}
    Singleton(Singleton &);
    Singleton operator= (Singleton &);
};

int main()
{
    Singleton& mySingleton = Singleton::getUniqueInstance();
    mySingleton.doSomething();
}
```

- The Singleton design pattern provides an example of a case where declaring a constructor private makes sense
- Singleton enforces that only one object of a class can be present during the lifetime of a program

Reading assignment for next week

- **Read a lot about classes in general**
- **Class inheritance in C++**