OOP-Intro

August 8, 2020

1 Object Oriented Programming (OOP) Introduction

1.1 Table of Contents

1.1.1 all header includes requried for this notebook

[1]:

```
#include <iostream>
#include <string>
#include <cassert>

using namespace std;
```

1.2 Procedural programming

- most of CS1 is procedural programming
- derived from structured programming, based on the concept of the procedure call
- procedures, also called routines, subroutines, or methods define the computational steps to be carried out
- the focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines
- no concept of ownership of data
- functions or procedures passively operate on data

1.3 Object oriented programming

- programming paradigm based on the concept of `objects`, which are **data structures** and **operations**
  - `data structures`: contain data in the form of fields or attributes

    - `operations` : code in the form of methods, procedures, functions
- objects are the core of the problems
    - the focus of OOP is to break down a programming task into `objects` that expose behavior (methods) and data (fields) using interfaces
    - OOP bundles data and methods, so an `object`, actively operates on its "own" data
- objects are instances/variables of some user-defined class types

1.3.1 OOP in C++

- keywords `class` or `struct` can be used to define some class type
- very similar to `struct`
- two-step process:
    - define class and use class via objects

1.4 Principles of OOP

- 4 principles of OOP

1.4.1 Encapsulation

- refers to creation of self-contained modules (classes) that bind data and methods/functions that operate on data
- data within each class is kept private
- class defines rules for what is publicly visible and what modifications are allowed

1.4.2 Abstraction

- denotes a model, a view or some other focused representation or metaphors for real-world objects
- `Encapsulation` hides the details of that implementation

1.4.3 Inheritance

- classes may be created in hiearchies
- inheritance lets the attributes and methods in one class (parent) pass down to the `class hierarchy` (children)
- allows for reuse of code in a parent class (major benefit)

1.4.4 Polymorphism

- allows for a way to determine type of an object during runtime
- E.g., a screen "cursor" may change its shape from an `arrow` to a `line` depending on the program mode
    - polymorphism makes right routine to be called when "cursor" is moved

1.4.5 Syntax to define class

```
class className {
  accessSpecifier1: // can be private, public, or protected
    data members; // variables/attributes
    function members; // methods to access/operate on data
  accessSpecifier2:
     // data and function members
  ...
}; // ends with semicolor
```

1.4.6 private:

- by default all members of class are private
- `private` members are accessible only from within other members of the same class (or from their "friends")

1.4.7 protected:

- `protected` members are accessible from other members of the same class (or from their "friends")
- and also from members of their children/derived classes

1.4.8 public:

- `public` members are accessible from anywhere the object is visible

1.4.9 Declare objects

```
className objectName;
```

1.4.10 Access members

```
objectName.memberName
```

- .(dot) member access operator is used to access members that are accessible

[ ]:

```
// Example 0
class UselessClass {
  int mem1;
  string mem2;
  void print() {
      cout << mem1 << mem2 << endl;
  }
};
```

[ ]:

```
UselessClass uselessObject; // instantiate an object userlessObject of type␣
 ↪UselessClass
```

[ ]:

```
uselessObject.mem1 = 100; // can't access any members as they're by default all␣
 ↪private!!
```

[ ]:

```
// Example 1 - A class with no encapsulation
class Rectangle {
  public:
    // member variables/attributes
    float length;
    float width;

    // member functions
    float getArea() {
        return length*width;
    }

    float getPerimeter() {
        return 2*(length+width);
    }
};
```

[ ]:
```
Rectangle r1; // instantiate an object r1 of type Rectangle
```

[ ]:
```
r1.length = 10;
r1.width = -15;
cout << "area = " << r1.getArea() << endl;
cout << "perimeter = " << r1.getPerimeter() << endl;
```

[ ]:
```
// How about this? Nothing can prevent us from doing the following...
// Can you tell what's wrong in the following code??
Rectangle r2;
r2.length = -20;
r2.width = 0;
cout << "area = " << r2.getArea() << endl;
cout << "perimeter = " << r2.getPerimeter() << endl;
```

[ ]:
```
// Example 2 - another class with no encapsulation
class BadBoyShipping {
    public:
        int weight;
        string address;
        /* remaining code ommitted... */
}
```

[ ]:
```
BadBoyShipping *bad = new BadBoyShipping();
bad->weight = -3;
```

1.5 Constructors

- special methods that let you initialize member variables
- lets you add code to do data validation; -ve or empty data may not be allowed, e.g.
- they're invoked/called automatically based on how objects are initialized (based on parameters)
- constructors have the same name as the class and no return type

- can overload more than 1 constructors
- C++ provides default constructor (constructor without arguments) if no constructor is provided
  - if a constructor is provided, default constructor must be provided as well...
- Syntax:

```
className(parameter list...) {
   // code to initialize member variables
}
```

1.6 Destructors

- special method which destructs or deletes and object
- a destructor is called automatically when the object goes out of scope:
  - the function ends
  - the program terminates
  - a block containing local variables ends
  - a delete operator is called
- like constructors, destructors are different from normal member functions
- Syntax:

```
~ className() {
   // code clean up
   // delete dynamic variables
   // send signals
}
```

1.7 Getter and Setter methods

- member functions or methods that get and set private and protected member variables
- helps in data encapsulation by providing API (Application Programming Interface) to work with data variables

1.8 Keyword this

- `this` represents a pointer to the object whose members are being accessed or executed
- it is used within a class's member function to refer to the object itself

[2]:

```
// Example 2
class Person {
  private:
    // member variables/ attributes
    string name; // some naming conventions use leading _ before attributes
    int * _age; // int * _age; to distinguish from parameters and local␣
 ↪variables

  public:
    // member functions/methods
    // default constructor
    Person() {
        name = "AA BB";
        _age = new int; //dynamic variable
        *_age = 0;
    }
    // overloaded constructor that takes arguments
    Person(string name, int age) {
        assert(name != "");
        assert(age >= 0);
        // if assertion passed, set the attributes
        this->name = name; // this->name is member variable whereas name is local variable
        _age = new int;
        *_age = age;
    }

    // getters
    string getName() const {
        //name = "dsfdsaf";
        return name;
    }

    int getAge() const { return *_age; }

    // setters
    void setName(string name) {
        assert(name != "");
        this->name = name;
    }

    void setAge(int age) {
        assert (age >= 0);
        *_age = age;
    }

    void introduce() const {
        cout << "Hi, my name is " << name << ".\n";
        cout << "I'm " << *_age << " years old. It's pleasure meeting you!\n";
    }

    // destructor
    /*
    ~Person() {
        delete _age; //delete dynamic variable to prevent memory leak
        cout << "I'm destroyed!" << endl;
    }
    */
};
```

[4]:

```
Person p1; // automatically calls default constructor
p1.introduce();
```

```
Hi, my name is AA BB.
I'm 0 years old. It's pleasure meeting you!
```

[5]:

```
Person p2("John Smith", 35); // functional form; very common way
```

[6]:

```
p2.introduce();
```

```
Hi, my name is John Smith.
I'm 35 years old. It's pleasure meeting you!
```

[7]:

```
// explictly using constructor
Person p3 = Person("Jake Jones", 45); // functional form init
```

[8]:

```
p3.introduce();
```

```
Hi, my name is Jake Jones.
I'm 45 years old. It's pleasure meeting you!
```

[9]:

```
p3.setName("Jackson Jones");
cout << "name = " << p3.getName() << endl;
```

```
name = Jackson Jones
```

[10]:

```
p3.introduce();
```

```
Hi, my name is Jackson Jones.
I'm 45 years old. It's pleasure meeting you!
```

[6]:

```
// Uniform init
Person p4 = {"Jane Smith", 29}; // not common
```

[12]:

```
p4.introduce();
```

```
Hi, my name is Jane Smith.
I'm 29 years old. It's pleasure meeting you!
```

1.9 Pointers to classes

- objects can also be pointed to by pointers
- Syntax:

```
className * pointer;
```

- use -> (arrow) operator to access members of class pointers
- allows to allocate memory dynamically

[13]:
```
Person * p5;
```

[14]:
```
p5 = &p4;
p5->introduce();
```

```
Hi, my name is Jane Smith.
I'm 29 years old. It's pleasure meeting you!
```

[15]:
```
p5->setName("Jane Jackson");
```

[16]:
```
p4.introduce();
p5->introduce();
```

```
Hi, my name is Jane Jackson.
I'm 29 years old. It's pleasure meeting you!
Hi, my name is Jane Jackson.
I'm 29 years old. It's pleasure meeting you!
```

[17]:
```
Person * p6 = new Person("Bill Gates", 60);
```

[18]:
```
p6->introduce();
```

```
Hi, my name is Bill Gates.
I'm 60 years old. It's pleasure meeting you!
```

1.10 Array of classes

- very similar to array of built-in types or structs!

[19]:
```
// declare array of 10 Person players
Person players[10];
```

[20]:
```
players[0] = {"Michael Jordan", 50};
players[1] = {"Magic Johnson", 55};
```

```
I'm destroyed!
I'm destroyed!
```

[21]:
```
players[0].introduce();
players[1].introduce();
```

```
Hi, my name is Michael Jordan.
I'm -1467829325 years old. It's pleasure meeting you!
Hi, my name is Magic Johnson.
I'm -1467829325 years old. It's pleasure meeting you!
```

[22]:
```
// access/update member variables of Player object stored at index 0 via setters
players[0].setName("Mike Jordan");
```

```
players[0].setAge(51);
```

[23]:

```
players[0].introduce();
```

```
Hi, my name is Mike Jordan.
I'm -1467996434 years old. It's pleasure meeting you!
```

[24]:

```
// dynamic array of Person
Person * people = new Person[2] {{"Jeff Bezos", 50}, {"Warren Buffet", 75}};
```

[25]:

```
people[0].introduce();
people[1].introduce();
```

```
Hi, my name is Jeff Bezos.
I'm 50 years old. It's pleasure meeting you!
Hi, my name is Warren Buffet.
I'm 75 years old. It's pleasure meeting you!
```

[4]:

```
#include <vector>
```

[5]:

```
vector<Person> people;
```

[6]:

```
Person p;
```

[7]:

```
string name;
int age;
```

[8]:

```
cout << "enter name and age: ";
cin >> name >> age;
```

```
enter name and age: John 35
```

[9]:

```
p.setName(name);
p.setAge(age);
```

[10]:

```
p.introduce();
```

```
Hi, my name is John.
I'm 35 years old. It's pleasure meeting you!
```

[11]:

```
people.push_back(p);
```

[ ]:

```
for (auto p : people) {
    p.introduce();
}
```

1.11 Aggregate operations on class objects

- by default only = (assignment) aggregate operation can be done on class objects
- can't compare two objects even if they're of same type without overloading those comparison operators

[14]:

```
// can copy one object to another; becareful of dynamic members (shallow copy...
 ↪ more on this later)!!
Person p7 = p;
```

[16]:

```
p7.introduce();
p.introduce();
```

```
Hi, my name is John.
I'm 869321564 years old. It's pleasure meeting you!
Hi, my name is John.
I'm 869321564 years old. It's pleasure meeting you!
```

[ ]:

```
// no aggregate comaparisons (==, !=, <=, <, >, >=) are allowed unless␣
 ↪overloaded
if (p7 == p4) cout << "same people!";
else cout << "not the same people!";
```

1.12 Passing classes to functions

- class objects can be passed to functions in two ways (by value and by reference)
- by default, classes are passed by value
- can be passed by reference using address of (&) operator

[5]:

```
// passed by value
void printPerson(Person p) {
    cout << "name: " << p.getName() << endl;
    cout << "age: " << p.getAge() << endl;
}
```

[8]:

```
printPerson(p4); // member variables of p4 are copied to p
```

```
name: Jane Smith
age: 29
```

[11]:

```
// passed by reference
void printPerson1(const Person &p) {
    cout << "name = " << p.getName() << endl; // read-only
    cout << "age = " << p.getAge() << endl;
    //p.setAge(10); // nothing can stop from doing this.. updating/writing data
}
```

```
input_line_18:2:6: error: redefinition of

'printPerson1'
void printPerson1(Person &p) {
     ^

input_line_16:2:6: note: previous definition is
here
void printPerson1(Person &p) {
     ^



        Interpreter Error:
```

[3]:

```
// passed by reference
void printPerson2(const Person &p) {
    cout << "name = " << p.getName() << endl; // read-only
    cout << "age = " << p.getAge() << endl;
    //p.setAge(10); // nothing can stop from doing this.. updating/writing data
}
```

[12]:

```
printPerson1(p4);
```

```
name = Jane Smith
age = 10
```

[13]:

```
p4.introduce(); // what do you think is now the age of p4
```

```
Hi, my name is Jane Smith.
I'm 10 years old. It's pleasure meeting you!
```

[ ]:

```
// How do we fix this problem?
// pass reference as constant if the function is supposed to read-only the data!
// demonstrate with an example...
// getters must be marked const or read-only!!
```

1.13 Return classes from functions

- class object can be returned from regular functions just like fundamental-types (int, float, char, etc.)

[4]:

```
Person createPerson() {
    int age=0;
    string name;
    Person p;
    cout << "Enter new person's name: ";
    getline(cin, name);
    p.setName(name);
    p.setAge(age);
    return p;
}
```

[5]:

```
Person newP = createPerson();
```

```
Enter new person's name: Baby John
```

[6]:

```
newP.introduce();
```

```
Hi, my name is Baby John.
I'm 0 years old. It's pleasure meeting you!
```

1.14 Reading expressions

expression	can be read as
*x	pointed to by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed to by x
(*x).y	member y of object pointed to by x
x[0]	first object pointed to by x
x[1]	second object pointed to by x
x[n]	(n+1)th object pointed to by x

1.15 Classes defined with struct and union

- classes can be defined with keywords `struct` and `union`
- **struct** is generally used to declare plain data structures (only member variables)
  - struct can also include member functions just like class
  - by default members of struct are `public`
- **union** only stores one data member at a time, but can also hold member functions
  - default access in union classes is `public`

1.16 OOP Paradigm - concept map

1.16.1 Exercise

Object-oriented programming is a programming paradigm based around __________. 1. Abstraction 2. Polymorphism - Inheritance - Objects

Find area and circumference of a circle using OOD (Object Oriented Design).

[ ]:

```

```