Queens College, CUNY,   Department of Computer Science

Object-Oriented Programming in C++

CSCI 211/611

Summer 2018

Instructor: Dr. Sateesh Mane

due date n/a

Homework: Polymorphism #1a

- **Experience with other classes has demonstrated that in many cases the source of difficulty is not the mathematics or the programming.**
- **The source of difficulty is the English (understanding the text).**
- **If you do not understand the words in the lectures or homework, THEN ASK.**
- **If you do not understand the concepts in the lectures or homework, THEN ASK.**
- **Send me an email, explain what you do not understand.**
- **Do not just keep quiet and then produce nonsense in exams.**

- **Consult your lab instructor for assistance.**
- **You may also contact me directly, but I cannot promise a prompt response.**
- Please submit your inquiry via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.
  1. The zip archive should have either of the names (CS211 or CS611):

     ```
     StudentId_first_last_CS211_hw_polymorphism1.zip
     StudentId_first_last_CS611_hw_polymorphism1.zip
     ```

  2. The archive should contain one "text file" named "hw_polymorphism1.[txt/docx/pdf]" (if required) and cpp files named "Q1.cpp" and"Q2.cpp" etc.
  3. Note that a text file is not always required for every homework assignment.
  4. Note that not all questions may require a cpp file.

General information

- You should include the following header files, to run the programs below.

  ```
  #include <iostream>
  #include <fstream>
  #include <sstream>
  #include <iomanip>
  #include <string>
  #include <cmath>
  ```

- If you require additional header files to do your work, feel free to include them.

- **Include the list of all header files you use, in your solution for each question.**

- The questions below do not require complicated mathematical calculations.

- If for any reason you require help with mathematical calculations, **ask the lab instructor or the lecturer.**

Q1 Virtual operators

- It is not only functions which can be virtual.
- Operators can also be virtual.
- Note that only an operator which is declared as a class method can be tagged as virtual.

Inheritance tree

We employ the same inheritance tree as previously.

CC DD
BB
AA

Q2 Problems

- Writing virtual operators can cause problems.
- Suppose we declare `operator<` as virtual.

```
class AA {
public:
    virtual bool operator<(const AA& rhs) const {     // virtual operator
        // etc
        return (boolean value);
    }
    // etc
};

class BB : public AA {
public:
    virtual bool operator<(const BB& rhs) const {     // virtual operator
        // etc
        return (boolean value);
    }
    // etc
};
```

- The difficulty here is that the function signatures of `operator<` in the classes AA and BB are ***not the same***.
- Therefore `operator<` in BB does *not* override `operator<` in AA.
- To override the base class operator, we would have to write the following in BB.

```
class BB : public AA {
public:
  virtual bool operator<(const AA& rhs) const {    // same operands as base class
    // etc
    return (boolean value);
  }
  // etc
};
```

- *However, what does such an operator mean, in the class BB?*
- Hence declaring operators as virtual can produce meaningless code.

Q3 Virtual `operator++`

- We shall overload `operator++` as a virtual operator.
- **Add the following as a public virtual operator to the class AA.**

```
class AA {
public:
  virtual AA& operator++() {
    ++(*ip);
    return *this;
  }

  // previous code
};
```

- The classes BB and CC do not override the virtual operator.
- **Add the following as a public virtual operator to the class DD.**

```
class DD : public BB {
public:
  virtual DD& operator++() {
    AA::operator++();              // invokes base class operator
    ++dp[0];
    ++dp[1];
    return *this;
  }

  // previous code
};
```

- **Although the return type is different, the operand signature is the same.**
- Therefore this overrides the base class operator.

Q4 Function and main program

- **Write the following function and main program.**
- Observe the following.
  1. The correct virtual operator is invoked in the function.
  2. The return value `*this` from `operator++` is the correct derived class object.
  3. The correct version of `print()` is invoked in all cases.

```
// include relevant headers and class declarations
using namespace std;

void poly_increment(AA &aaref)
{
  aaref.print();
  AA &tmp = ++aaref;   // aaref invokes virtual operator
                       // tmp is reference to correct derived object
  tmp.print();
}

int main()
{
  AA aa(2);
  DD dd(7, "pdstring", 8.2);

  poly_increment(aa);
  poly_increment(dd);   // correct virtual operator and virtual function are invoked

  return 0;
}
```