ECE 462
Object-Oriented Programming using C++ and Java

Lecture 22

Yung-Hsiang Lu

yunglu@purdue.edu

Programming Techniques

- Some tricky concepts that often confuse programmers.
- They really test your understanding of the languages.
- They are also favorite questions in interviews (and of course, exams).
- Knowing these answers help you avoid hard-to-find bugs.
- Please pause the video as necessary and make sure you really understand the answers.

Program Crash Before Main Starts

```
int main(int argc, char * argv[]) {
    cout << "main starts" << endl;
    return 0;
}
```

```
public static void main(String [] args) {
    System.out.println("main starts");
}
```

- Can it print anything else before "main starts"?
- Can this program crash without printing "main starts"?
- What is the implication if the answer is yes?
  - Program does not necessarily start at main.
  - It may take a long time before main starts.
  - An interactive may not be interactive at the beginning.

```
#include <iostream>
using namespace std;
class X {
public:
  X() { cout << "X::X" << endl; }
};

X xobj;

int main(int argc, char * argv[]) {
  cout << "main starts" << endl;
  return 0;
}
```

```
class X {
    public X() {
     System.out.println("X::X"); }
}
class programstart
{
    private static X xobj = new X();
    public static void main( String[]
     args )
    {
     System.out.println("main start");
    }
}
```

What are the outputs?
Pause the video and write your answers.

Non-Virtual and Virtual

- Non-virtual (NV) calls virtual (V): base NV and derived V
- V calls NV: derived V then derived NV
- V calls V: derived V then derived V
- NV calls NV: base NV and base NV

```
#include <iostream>
using namespace std;
class BaseC {
public:
  virtual void vfunc() {
    cout << "BaseC::vfunct" << endl;
  }
  void nvfunc() {
    cout << "BaseC::nvfunct" << endl;
    vfunc();
  }
  virtual void vfunc2() {
    cout << "BaseC::vfunc2" << endl;
    nvfunc();
  }
};
```

```
class DerivedC: public BaseC {
public:
  virtual void vfunc() {
    cout << "Derived::vfunct" << endl;
  }
  void nvfunc() {
    cout << "Derived::nvfunct" << endl;
    vfunc();
  }
  virtual void vfunc2() {
    cout << "DerivedC::vfunc2" << endl;
    nvfunc();
  }
};
```

```
int main(int argc, char * argv[]) {
  BaseC * bobj = new DerivedC;
  bobj -> nvfunc();
  bobj -> vfunc2();
  return 0;
}
```

What are the outputs?
Pause the video and write your answers.

Operator Calls Virtual

```
#include <iostream>
using namespace std;
class X { };
class Y: public X {};
class Z: public Y {};
ostream& operator << ( ostream& os, const X & arg ) {
  os << "operator << X" << endl;
  return os;
}
ostream& operator << ( ostream& os, const Y & arg ) {
  os << "operator << Y" << endl;
  return os;
}
```

```
ostream& operator << ( ostream& os, const Z & arg )
{
  os << "operator << Z" << endl;
  return os;
}
int main(int argc, char * argv[])
{
  X * xptr[3];
  xptr[0] = new X;
  xptr[1] = new Y;
  xptr[2] = new Z;
  cout << (* xptr[0]);
  cout << (* xptr[1]);
  cout << (* xptr[2]);
  return 0;
}
```

What are the outputs?

Pause the video and write your answers.

```
#include <iostream>
using namespace std;
class X {
public:
  virtual void print() const { cout << "Xp " << endl; }
};
class Y: public X {
public:
  void print() const { cout << "Yp " << endl; }
};
class Z: public Y {
public:
  void print() const { cout << "Zp " << endl; }
};
```

```
ostream& operator << ( ostream& os, const X & arg ) {
  os << "operator << X" << endl;
  arg.print();
  return os;
}
int main(int argc, char * argv[])
{
  X * xptr[3];
  xptr[0] = new X;
  xptr[1] = new Y;
  xptr[2] = new Z;
  cout << (* xptr[0]);
  cout << (* xptr[1]);
  cout << (* xptr[2]);
  return 0;
}
```

```
#include <iostream>
#include <string>
using namespace std;
class X {
public:
  virtual void print() const { cout << "Xp " << endl; }
  X() { cout << "X::X" << endl; }
};
class Y: public X {
public:
  void print() const { cout << "Yp " << endl; }
  Y() { cout << "Y::Y" << endl; }
};
class Z: public Y {
public:
  void print() const { cout << "Zp " << endl; }
```

```
 Z() { cout << "Z::Z" << endl; }
};
ostream& operator << ( ostream& os, const X & arg ) {
  os << "operator << X" << endl;
  arg.print();
  return os;
}
ostream& operator << ( ostream& os, const Y & arg ) {
  os << "operator << Y" << endl;
  arg.print();
  return os;
}
ostream& operator << ( ostream& os, const Z & arg ) {
  os << "operator << Z" << endl;
  arg.print();
  return os;
}
```

```
void funct(X * xptr) { cout << "fX " << endl; }
void funct(Y * yptr) { cout << "fY " << endl; }
void funct(Z * zptr) { cout << "fZ " << endl; }
int main(int argc, char * argv[])
{
  X * xptr[3];
  xptr[0] = new X; cout << endl;
  xptr[1] = new Y; cout << endl;
  xptr[2] = new Z; cout << endl;
  if (argc > 1)  { xptr[2] = new Z; }
  else { xptr[2] = new Y; }
  funct(xptr[0]);
  funct(xptr[1]);
  funct(xptr[2]);
  xptr[0] -> print();
  xptr[1] -> print();
  xptr[2] -> print();
```

```
 cout << (* xptr[0]);
  cout << (* xptr[1]);
  cout << (* xptr[2]);
  return 0;
}
```

What are the outputs?
Pause the video and write your answers.

```
class X {     public void print() { System.out.println("  X::print"); } }
class Y extends X {    public void print() { System.out.println("  Y::print"); } }
class Z extends Y {     public void print() { System.out.println("  Z::print"); } }
class virtualoverload
{
    public static void func(X obj) {
     System.out.println("func(X)");
     obj.print();
    }
    public static void func(Y obj) {
     System.out.println("func(Y)");
     obj.print();
    }
```

```
    public static void func(Z obj) {
     System.out.println("func(Z)");
     obj.print();
    }
    public static void main( String[] args )
    {
     X [] xobj = new X[3];
     xobj[0] = new X();
     xobj[1] = new Y();
     xobj[2] = new Z();
     func(xobj[0]);
     func(xobj[1]);
     func(xobj[2]);
    }
}
```

Overload → Method → Overload

```
class Common {
    public static void func1(X obj) {
     System.out.println("func1(X)");
    }
    public static void func1(Y obj) {
     System.out.println("func1(Y)");
    }
    public static void func1(Z obj) {
     System.out.println("func1(Z)");
    }
}
```

```
class X {
    public void print() {
      System.out.println("X::print");
      Common.func1(this);
    }
}
class Y extends X {
    public void print() {
      System.out.println("Y::print");
      Common.func1(this);
    }
}
class Z extends Y {
    public void print() {
      System.out.println("Z::print");
      Common.func1(this);
    }
}
```

```
class virtualoverload2
{
    public static void func2(X obj) {
     System.out.println("\nfunc2(X)");
    }
    public static void func2(Y obj) {
     System.out.println("\nfunc2(Y)");
    }
    public static void func2(Z obj) {
     System.out.println("\nfunc2(Z)");
    }
    public static void main( String[] args )
    {
     X [] xobj = new X[3];
     xobj[0] = new X();
     xobj[1] = new Y();
     xobj[2] = new Z();
```

```
    func2(xobj[0]);
    func2(xobj[1]);
    func2(xobj[2]);
  }
}
```

```
class Common {
    public static void func1(X obj) {
     System.out.println("func1(X)");
    }
    public static void func1(Y obj) {
     System.out.println("func1(Y)");
    }
    public static void func1(Z obj) {
     System.out.println("func1(Z)");
    }
}
class X {
    public void print() {
     System.out.println("X::print");
     Common.func1(this);
    }
}
```

```
class Y extends X {
    public void print() {
      System.out.println("Y::print");
      Common.func1(this);
    }
}
class Z extends Y {
    public void print() {
      System.out.println("Z::print");
      Common.func1(this);
    }
}
class virtualoverload3
{
    public static void func2(X obj) {
      System.out.println("\nfunc2(X)");
      Common.func1(obj); // additional call
    }
```

```
 public static void func2(Y obj) {
  System.out.println("\nfunc2(Y)");
  Common.func1(obj);
 }
 public static void func2(Z obj) {
  System.out.println("\nfunc2(Z)");
  Common.func1(obj);
 }
 public static void main( String[] args )
 {
  X [] xobj = new X[3];
  xobj[0] = new X();
  xobj[1] = new Y();
  xobj[2] = new Z();
  func2(xobj[0]);
  func2(xobj[1]);
  func2(xobj[2]);
 }
}
```

```
class Common {
    public static void func1(X obj) {
      System.out.println("func1(X)");
    }
    public static void func1(Y obj) {
      System.out.println("func1(Y)");
    }
    public static void func1(Z obj) {
      System.out.println("func1(Z)");
    }
}
class X {
    public void print() {
      System.out.println("X::print");
      Common.func1(this);
    }
}
```

```
class Y extends X {
    public void print() {
     System.out.println("Y::print");
     Common.func1(this);
    }
}
class Z extends Y {
    public void print() {
     System.out.println("Z::print");
     Common.func1(this);
    }
}
class virtualoverload4
{
    public static void func2(X obj) {
     System.out.println("\nfunc2(X)");
     Common.func1(obj);
```

```
  obj.print();
 }
 public static void func2(Y obj) {
  System.out.println("\nfunc2(Y)");
  Common.func1(obj);
  obj.print();
 }
 public static void func2(Z obj) {
  System.out.println("\nfunc2(Z)");
  Common.func1(obj);
  obj.print();
 }
 public static void main( String[] args )
 {
  X [] xobj = new X[3];
```

```
        xobj[0] = new X();
        xobj[1] = new Y();
        xobj[2] = new Z();
        func2(xobj[0]);
        func2(xobj[1]);
        func2(xobj[2]);
    }
}
```

Run the Program Yourself

```
class Common {
    public static void func1(X obj) {
      System.out.println("func1(X)");
    }
    public static void func1(Y obj) {
      System.out.println("func1(Y)");
    }
    public static void func1(Z obj) {
      System.out.println("func1(Z)");
    }
}
```

```
class X {
    public void print(X obj) {
      System.out.println("X::print(X)");
      Common.func1(obj);
    }
    public void print(Y obj) {
      System.out.println("X::print(Y)");
      Common.func1(obj);
    }
    public void print(Z obj) {
      System.out.println("X::print(Z)");
      Common.func1(obj);
    }
}
```

```
class Y extends X {
    public void print(X obj) {
      System.out.println("Y::print(X)");
      Common.func1(obj);
    }
    public void print(Y obj) {
      System.out.println("Y::print(Y)");
      Common.func1(obj);
    }
    public void print(Z obj) {
      System.out.println("Y::print(Z)");
      Common.func1(obj);
    }
}
```

```
class Z extends Y {
    public void print(X obj) {
      System.out.println("Z::print(X)");
      Common.func1(obj);
    }
    public void print(Y obj) {
      System.out.println("Z::print(Y)");
      Common.func1(obj);
    }
    public void print(Z obj) {
      System.out.println("Z::print(Z)");
      Common.func1(obj);
    }
}
```

```
class virtualoverload5
{
    public static void func2(X obj) {
     System.out.println("\nfunc2(X)");
     Common.func1(obj);
     obj.print(obj);
    }
    public static void func2(Y obj) {
     System.out.println("\nfunc2(Y)");
     Common.func1(obj);
     obj.print(obj);
    }
    public static void func2(Z obj) {
     System.out.println("\nfunc2(Z)");
     Common.func1(obj);
     obj.print(obj);
    }
```

```
public static void main( String[] args )
    {
     X [] xobj = new X[3];
     xobj[0] = new X();
     xobj[1] = new Y();
     xobj[2] = new Z();
     func2(xobj[0]);
     func2(xobj[1]);
     func2(xobj[2]);
    }
}
```

Friend Class and Inheritance

- If a method is declared as virtual, it is virtual for all the derived classes.
- If a class is declared as a friend in the base class, is it also a friend in the derived class?

```
#include <iostream>
using namespace std;
class X {
public:
  X(int v) { x_val = v; }
private:
  int x_val;
  friend class FC;
};
```

```
class Y: public X {
public:
  Y(int u, int v):X(u) { y_val = v; }
private:
  int y_val;
  friend class FC;
};
class FC {
public:
  void print1(X* obj) { cout << obj -> x_val << endl; }
  void print1(Y* obj) {
    cout << obj -> x_val << " " << obj -> y_val << endl;
  }
  void print2(Y* obj) {
    cout << obj -> x_val << " " << obj -> y_val << endl;
  }
};
```

```
int main(int argc, char * argv[]) {
  X * obj1 = new X(1);
  X * obj2 = new Y(2, 3);
  FC fcobj;
  fcobj.print1(obj1);
  fcobj.print1(obj2);
}
```

Overload and Friend Functions

```
#include <iostream>
using namespace std;
class X {
public:
  X(int v) { xval = v; }
private:
  int xval;
  friend void func(X & objref);
  friend void func(X  objref);
  friend void func(X * objptr);
};
void func(X & objref) { cout << objref.xval << endl; }
void func(X objref) { cout << objref.xval << endl; }
void func(X * objptr) { cout << objptr -> xval << endl; }
```

```
int main(int argc, char * argv[])
{
  X obj1(7);
  // in general, do not use 0 as the test data because 0 can also mean
  // "uninitialized"
  X * obj2 = new X(12);
  func(obj1);
  func(obj2);
  return 0;
}
```

Overloading vs. Default Values

```
void f() { /* function 1 */ }
void f(int x) { /* function 2 */ }
f(); ⇒ calls function 1
f(5); ⇒ calls function 2
```

```
g(int x = 10) { /* function 3 */ }
g(); ⇒ calls function 3 with x = 10
g(5); ⇒ calls function 3 with x = 5
```

What is the difference?

- Is there a reasonable default value? If so, consider using default values.
- Do two functions use the same algorithm? If so, consider using default values.

Overload + Override (Virtual)

- Can a function be overloaded and overridden in derived classes? $\Rightarrow$ Yes.
- Can a global function (not a member in any class) be overloaded? $\Rightarrow$ Yes.
- Can a global function (not a member in any class) be overridden? $\Rightarrow$ No. This question makes no sense.

```
#include <iostream>
#include <string>
using namespace std;
int callAccum = 0;
class X {
public:
  void func1(int v)
  { cout << "X::func1(int) " << v << endl; callAccum += 1; }
  virtual void func1(string v)
  { cout << "X::func1(string) " << v << endl; callAccum += 2; }
};
class Y: public X {
public:
  void func1(int v)
  { cout << "Y::func1(int) " << v << endl; callAccum += 3; }
  virtual void func1(string v)
  { cout << "Y::func1(string) " << v << endl; callAccum += 4; }
};
```

```
/*
  You cannot overload global main in C++
  C++ standard allows two formats for main:
  int main() { ... }
and
  int main(int argc, char* argv[]) {  }
*/
/*
int main(char * arg) {
  cout << "Yes, you can overload main " << endl;
}
*/
int foo(char * arg) {
  cout << "Yes, you can overload foo " << endl;
}
```

```
int foo(int argc, char * argv[]) {
  cout << "Yes, you can overload foo " << endl;
}
int main(int argc, char * argv[]){
  X * xptr1 = new X;
  X * xptr2 = new Y;
  Y * yptr1 = new Y;
  X xobj;
  Y yobj;
  xptr1 -> func1(9);
  xptr2 -> func1(11);
  yptr1 -> func1(-6);
  xptr1 -> func1("hello");
  xptr2 -> func1("Purdue");
  yptr1 -> func1("boiler");
  xobj.func1(7);
```

```
  xobj.func1("Lafayette");
  yobj.func1(8);
  yobj.func1("ECE");
  cout << callAccum << endl;
  // main("Try This");
  foo("Try This");
  return 0;
}
```

Can "main" be a Member Function?

```
#include <iostream>
#include <string>
using namespace std;
class X {
public:
  void main(string msg) {
    cout << "You can have a member function called main- " << msg << endl;
  }
};
int main(int argc, char * argv[]) {
  X obj;
  obj.main("Hello");
}
```

```
// overload + override in Java
class X {
    public static int  callAccum = 0;
    public void func1(int v)
    { System.out.println("X::func1(int) " + v); callAccum += 1; }
    public void func1(String v)
    { System.out.println("X::func1(String) " + v); callAccum += 2; }
    public String toString()
    { return new String (Integer.toString(callAccum)); }
};
class Y extends X {
    public void func1(int v)
    { System.out.println("Y::func1(int) " + v); callAccum += 3; }
    public  void func1(String v)
    { System.out.println("Y::func1(String) " + v); callAccum += 4; }
};
```

```
class overloadoverride {
    public static void main(String args) {
     System.out.println("Yes, you can overload main " + args);
    }
    public static void main(String [] args) {
     X xobj1 = new X();
     X xobj2 = new Y();
     Y yobj1 = new Y();
     X xobj = new X();
     Y yobj = new Y();
     xobj1.func1(9);
     xobj2.func1(11);
     yobj1.func1(-6);
     xobj1.func1("hello");
     xobj2.func1("Purdue");
     yobj1.func1("boiler");
     xobj.func1(7);
```

```
        xobj.func1("Lafayette");
        yobj.func1(8);
        yobj.func1("ECE");
        System.out.println(xobj1);
        main("Try This");
    }
}
```

Overload with Classes

```
class X {
}
class Y extends X {
}
class Z {
    private int accum = 0;
    public void func(X obj)
    { System.out.println("func(X)"); accum += 1; }
    public void func(Y obj)
    { System.out.println("func(Y)"); accum += 2; }
    public void func(X obj1, X obj2)
    { System.out.println("func(X, X)"); accum += 3; }
    public void func(X obj1, Y obj2)
    { System.out.println("func(X, Y)"); accum += 4; }
```

```
    public void func(Y obj1, X obj2)
    { System.out.println("func(Y, X)"); accum += 5; }
    public void func(Y obj1, Y obj2)
    { System.out.println("func(Y, Y)"); accum += 6; }
    public String toString()
    { return new String(Integer.toString(accum)); }
}
class overload
{
    private static X xobj = new X();
    public static void main( String[] args )
    {
      X obj1 = new X();
      X obj2 = new Y();
      Y obj3 = new Y();
      // Y obj4 = new X(); // error
```

```
        Z zobj = new Z();
        zobj.func(obj1);
        zobj.func(obj2);
        zobj.func(obj3);
        zobj.func(obj1, obj2);
        zobj.func(obj2, obj3);
        zobj.func(obj1, obj3);
        zobj.func(obj3, obj3);
        System.out.println(zobj);
    }
}
```