CSE 374: Programming Concepts and Tools

Eric Mullen
Spring 2017
Lecture 23: C++: Vtables
Administrivia

• Homework 6 is due this Thursday
• Use turn in instructions in assignment
• Come get help in office hours! We can make your life better
• Homework 7 out on Friday (smaller, in C++)
• No class next Monday (Memorial Day)
• Final is June 7 at 2:30pm-4:20pm in this room
Object Oriented Programming (OOP)

- Popular programming paradigm
- Everything is an object
- Every object owns its own implementation, usually given by its class
- In order to modify an existing program, a programmer extends classes, overrides methods, and so on
- Every part of a program can be extended, everything is very open
C++ = C + OOP

- C++ was originally called C with Classes
- C was a useful language, but more modern object oriented features were desired
- Many of the features were more “nice to have” features, but Classes are core to C++
- Classes were controversial: some thought too slow
  - Computers are much faster now, other criticisms more common
Building Classes

- It is possible to write code with the exact same behavior as a C++ virtual method call in C

- Today, in order to understand how C++ works, we’ll do just that
class Point {
protected:
    int x;
    int y;
public:
    Point();
    Point(int x, int y);
    int getX();
    int getY();
};
Normal Functions

• Class attempt #1:

typedef struct Point {
  int x;
  int y;
} Point;

Point Point_Constr(int x, int y) {
  point* p = (point*)malloc(sizeof(point));
  p->x = x;
  p->y = y;
  return p;
}
Normal Functions

• Class attempt #1:

```c
Point* Point_ConstrD() {
    return Point_Constr(0,0);
}

int getX(Point* p) {
    return p->x;
}

int getY(Point* p) {
    return p->y;
}
```
Normal Functions

• Class attempt #1:

    ```c
    Point* p = Point_Constr(3,2);
    printf("(%d,%d)\n",getX(p),getY(p));
    ```
class Point {
protected:
    int x;
    int y;
public:
    Point();
    Point(int x, int y);
    int getX();
    int getY();
};
Normal Functions

• Class attempt #1:

typedef struct Point {
    int x;
    int y;
} Point;

typedef struct PolarPoint {
    int x;
    int y;
    float r;
    float theta;
} PolarPoint;
Normal Functions

• Class attempt #1:
  int getX(Point* p) {
    return p->x;
  }

  int getY(Point* p) {
    return p->y;
  }

  int getXP(PolarPoint* p) {
    return p->x;
  }

  int getYP(PolarPoint* p) {
    return p->y;
  }
Normal Functions

• Class attempt #1:

```c
PolarPoint* p = PolarPoint_Constr(3,2);
printf("(%d, %d)\n", getXp(p), getYp(p));
```
Dynamic Dispatch

• Core of object oriented programming

• Allows for the code of method calls to be chosen at run time, based on the dynamic type of the receiver object

• This is necessary to make getX, getY work with our new PolarPoint class
How would you implement this?
class Point {
    protected:
        int x;
        int y;
    public:
        Point();
        Point(int x, int y);
        int getX();
        int getY();
};

// Point class
1 per class

// Point object
vtable pointer;
int x;
int y;
class PolarPoint : public Point {
private:
    float r, theta;
public:
    PolarPoint(float r, float t);
    float getR();
    float getTheta();
};

int getX();
int getY();
def getR();
def getTheta();
How it is implemented

Point* p = new PolarPoint(3,2);
std::cout << p->getX() << std::endl;
Questions?
Pros

• If classes are extended, no need to recompile code which uses objects of that class

• Cost of virtual method call is low: 2 pointer lookups, likely in cache anyway, virtually no difference with normal function calls
Cons

• A field of your object now exclusively determines what code gets called

• This is a security hole which has lead to at least one zero-day vulnerability in practice