CS-202

Recapitulation (Pt.1)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Today's Topics

Recapitulation:

- Classes
- Inheritance & Polymorphism
- Dynamic Memory

Prerequisites (not covered in Recap):

- Pass-by-Value

```
void func( DataType obj );
```

- Pass-by-Reference – Pass-by-**const**-Reference

```
void func( DataType& obj ); / void func( const DataType& obj );
```

- Pass-by-Address(Pointer) – Pass-by-**const**-Address(Pointer)

```
void func( DataType* obj ); / void func( const DataType* obj );
```

- Return-a-Value

```
DataType func( );
```

- Return-a-Reference – Return-a-**const**-Reference

```
DataType& func( ); / const DataType& func( );
```

- Return-an-Address(Pointer) – Return-a-**const**-Address(Pointer)

```
DataType* func( ); / const DataType* func( );
```

Class Declaration & Implementation

```
const size_t ID_LEN = 5+1;
const char DEFAULT_ID[ID_LEN] = "00000";   const char * DEFAULT_PLATES = "Default-Plate";
class Car {
  public:
    Car();
    Car(const char * plates, const char id[ID_LEN]=DEFAULT_ID,
        const Engine & engine=Engine(), Driver * driver=NULL, size_t serial=count);
    Car(const Car & other);
    ~Car();
    Car & operator=(const Car & other);
    Engine & getEngine();      const Engine & getEngine() const;
    Driver * getDriver();      const Driver * getDriver() const;
  friend std::ostream & operator<<(std::ostream & os, const Car & car);
  friend std::istream & operator>>(std::istream & is, Car & car);
  private:
    Engine m_engine;     // composition
    Driver * m_driver;   // aggregation
    char m_id[ID_LEN];
    char * m_plates;     // raw pointer
    const size_t m_serial;     // const
    static size_t count;       // static
};
```

Class Declaration & Implementation

```
class Car {
public:
 …
private:
 Engine m_engine;
 Driver * m_driver;
 char m_id[ID_LEN];
 char * m_plates;
 const size_t m_serial;
 static size_t count;
};
```

```
size_t Car::count = 0;

Car::Car() : m_serial( count++ ){
  m_plates = NULL;
  m_driver = NULL;
  //count already incremented
}

Car::Car(const char * plates, const char id[ID_LEN],
          const Engine & engine, Driver * driver, size_t serial)
     :  m_serial(count = serial>count ? serial : count){   //get the bigger number
  m_engine = engine;
  m_driver = driver;
  strcpy(m_id, id);
  m_plates = new char [ strlen(plates)+1 ];  //have to allocate first
  strcpy(m_plates, plates);
  ++count;  //increment at the end, constructor done & no exceptions occurred
}
```

Class Declaration & Implementation

```
Car::Car(const Car & other) : m_serial( count ){
  m_engine = other.m_engine;
  m_driver = other.m_driver; //same (pointer to outside object) driver
  strcpy(m_id, other.m_id);
  m_plates = new char [ strlen(other.m_plates)+1 ]; //allocate new
  strcpy(m_plates, other.m_plates);
  ++count; //increment at the end (constructor done)
}

Car::~Car(){
  //engine is class member object (aggregation) - will be automatically destroyed
  //driver is pointer to external object (composition) - no deleting
  delete [] m_plates; //m_plates uses dynamic memory - delete
  //destroying object, m_plates=NULL unnecessary
  //no decrementing of count (--count;), acts like a unique it generator
}
```

```
class Car {
public:
 …
private:
 Engine m_engine;
 Driver * m_driver;
 char m_id[ID_LEN];
 char * m_plates;
 const size_t m_serial;
 static size_t count;
};
```

Class Declaration & Implementation

```
Car& Car::operator=(const Car & other){
  if (this != &other){ //protect from self-assignment
   m_engine = other.m_engine;
   m_driver = other.m_driver; //same (pointer to outside object) driver
   strcpy(m_id, other.m_id);
   delete [] m_plates; //have to delete dynamic memory first
   m_plates = NULL; //object might outlive an exception on next line
   m_plates = new char [ strlen(other.m_plates)+1 ]; //allocate new
   strcpy(m_plates, other.m_plates);
  }
  return *this;
}
```

```
class Car {
public:
 …
private:
 Engine m_engine;
 Driver * m_driver;
 char m_id[ID_LEN];
 char * m_plates;
 const size_t m_serial;
 static size_t count;
};
```

Class Declaration & Implementation

```
std::ostream & operator<<(std::ostream & os, const Car & car){
  os << car.m_serial<<": "<<car.m_id <<", "
     << car.m_plates<<"-"<<car.m_engine;
  //driver is a pointer, have to check it, and have to dereference it
  if (m_driver){ os << " driver: " << *m_driver; }
  return os;
}

std::istream & operator>>(std::istream & is, Car & car){
  cout << "Expecting engine details (cc)" << endl;
  is >> car.m_engine;
  cout << "Expecting id[" << ID_LEN <<"]" << endl;
  is >> car.m_id;
  if (car.m_plates){
    cout << "Expecting license plates" << endl;
    is >> car.m_plates;
  }
  return is;
}
```

```
class Car {
public:
 …
private:
 Engine m_engine;
 Driver * m_driver;
 char m_id[ID_LEN];
 char * m_plates;
 const size_t m_serial;
 static size_t count;
};
```

Class Declaration & Implementation

```
const Engine & Car::getEngine() const{ //read-out engine (const-access)
  return m_engine;
}
Engine & Car::getEngine(){ //can also read-in engine (non-const-access)
  return m_engine;
}

const Driver * Car::getDriver() const{ //read-out driver (const-access)
  return m_driver;
}
Driver * Car::getDriver(){ //can also read-in driver (non-const-access)
  return m_driver;
}

/* also have to have */
// const char * Car::getID() const{ … }
// const char * Car::getPlates() const{ … }
// size_t Car::getSerial() const{ … }
```

```
class Car {
public:
 …
private:
 Engine m_engine;
 Driver * m_driver;
 char m_id[ID_LEN];
 char * m_plates;
 const size_t m_serial;
 static size_t count;
};
```

Working with Hierarchies

```
class Vehicle {
  public:
    Vehicle();
    Vehicle(const char * plates, const Engine & engine=Engine());
    Vehicle(const Vehicle & other);
    ~Vehicle();
    Vehicle & operator=(const Vehicle & other);
    const Engine & getEngine() const{ return m_engine; }
    void setEngine(const Engine& engine){ m_engine = engine; }
    const char * getPlates() const{ return m_plates; }
    void setPlates(const char * plates){ strcpy(m_plates, plates); }
    void move();
  protected:
    char * m_plates;
    float m_miles;
  private:
    Engine m_engine;
};
```

```
void Vehicle::move(){
  cout << "class Vehicle does not know how to move…" << endl;
}
```

Working with Hierarchies

```
const size_t SEDAN_DEFAULT_GEARS = 5;
const double SEDAN_DEFAULT_ENGINE = 2.0;
class Sedan : public Vehicle {
  public:
    Sedan();
    Sedan(const char * plates,
          bool manual=false, size_t gears=SEDAN_DEFAULT_GEARS,
          const Engine & engine=Engine(SEDAN_DEFAULT_ENGINE));
    Sedan(const Sedan & other);
    ~Sedan();
    Sedan & operator=(const Sedan & other);
    bool getManual() const{ return m_manual; }
    void setManual(bool manual){ m_manual = manual; }
    size_t getGears() const{ return m_gears; }
    void setGears(size_t gears){ m_gears = gears; }
    float move();
    float driveInCity();
  private:
    bool m_manual;
    size_t m_gears;
};
```

```
class Vehicle {
 public:
  Vehicle();
  Vehicle(const char* plates,
  const Engine & engine=Engine());
  Vehicle(const Vehicle & other);
  ~Vehicle();
  …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

Working with Hierarchies

```
/* base class ctor called first, then derived class ctor */
Sedan::Sedan() {
}

Sedan::Sedan(const char * plates, bool manual,
             size_t gears, const Engine& engine)
          : Vehicle(plates, engine){
  m_manual = manual;
  m_gears = gears;
}

Sedan::Sedan(const Sedan & other)
          : Vehicle(other.m_plates, other.getEngine()){
  m_manual = other.m_manual;
  m_gears = other.m_gears;
}

/* derived class dtor called first, then base class dtor */
Sedan::~Sedan() {
}
```

```
class Vehicle {
 public:
  Vehicle();
  Vehicle(const char * plates,
  const Engine & engine=Engine());
  Vehicle(const Vehicle & other);
  ~Vehicle();
  …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

```
class Sedan : public Vehicle {
 public:
  …
  float driveInCity();
 private:
  bool m_manual;
  size_t m_gears;
};
```

Working with Hierarchies

```
Sedan & Sedan::operator=(const Sedan & other){
 if (this != &other){ //protect from self-assignment
  m_manual = other.m_manual;
  m_gears = other.m_gears;

  //handle base class members
  setEngine( other.getEngine() );
  delete [] m_plates; //delete dynamic memory first
  m_plates = NULL; //object might outlive an exception
  m_plates = new char [ strlen(other.m_plates)+1 ];
  strcpy(m_plates, other.m_plates);
 }
 return *this;
}
```

```
class Vehicle {
 public:
  Vehicle();
  Vehicle(const char * plates,
  const Engine & engine=Engine());
  Vehicle(const Vehicle & other);
  ~Vehicle();
  …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

```
class Sedan : public Vehicle {
 public:
  …
  float driveInCity();
 private:
  bool m_manual;
  size_t m_gears;
};
```

Working with Hierarchies

```
float Sedan::driveInCity(){
 float milesThisTrip = 0;
 if (m_manual){
   /* required actions involving m_gears, etc… */
   cout<<"Sedan "<<m_plates<<" manual"<< endl;
 }
 else{
   /* required actions in this case, etc… */
   cout<<"Sedan "<<m_plates<<" automatic"<< endl;
 }
 return milesThisTrip;
}
```

```
float Sedan::move(){
  return (m_miles += driveInCity());
}
```

```
class Vehicle {
 public:
  Vehicle();
  Vehicle(const char * plates,
  const Engine& engine=Engine());
  Vehicle(const Vehicle & other);
  ~Vehicle();
  …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

```
class Sedan : public Vehicle {
 public:
  …
  float driveInCity();
 private:
  bool m_manual;
  size_t m_gears;
};
```

Working with Hierarchies

```
const double SUV_DEFAULT_ENGINE = 3.5;
class Suv : public Vehicle {
  public:
    Suv();
    Suv(const char * plates,
        bool awd=false, Emergencykit * emergencykit=NULL,
        const Engine & engine=Engine(SUV_DEFAULT_ENGINE));
    Suv(const Suv & other);
    ~Suv();
    Suv & operator=(const Suv & other);
    bool getAwd() const{ return m_awd; }
    void setAwd(bool awd){ m_awd = awd; }
    const Emergencykit * getEmergencykit() const;
    void setEmergencykit(const Emergencykit & emergencykit);
    float move();
    float driveInCityOffRoad(bool offroad);
  private:
    bool m_awd;
    Emergencykit * m_emergencykit;
};
```

```
class Vehicle {
 public:
  Vehicle();
  Vehicle(const char * plates,
  const Engine& engine=Engine());
  Vehicle(const Vehicle & other);
  ~Vehicle();
  …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

Working with Hierarchies

```
/* base class ctor called first, then derived class ctor */
Suv::Suv() {
  m_emergencykit = NULL;
}

Suv::Suv(const char * plates, bool awd,
         Emergencykit * emergencykit, const Engine & engine)
       : Vehicle(plates, engine){
  m_awd = awd;  m_emergencykit = emergencykit;
  if (!m_emergencykit && m_awd)
    m_emergencykit = new Emergencykit;
}

Suv::Suv(const Suv & other)
       : Vehicle(other.m_plates, other.getEngine()){
  m_awd = other.m_awd;
  m_emergencykit = new Emergencykit( *other.m_emergencykit );
}

/* derived class dtor called first, then base class dtor */
Suv::~Suv() {
  delete m_emergencykit;
}
```

```
class Vehicle {
 public:
  Vehicle();
  Vehicle(const char * plates,
  const Engine& engine=Engine());
  Vehicle(const Vehicle & other);
  ~Vehicle();
   …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

```
class Suv : public Vehicle {
 public:
   …
 float driveInCityOffRoad(bool);
 private:
  bool m_awd;
  Emergencykit * m_emergencykit;
};
```

Working with Hierarchies

```
Suv & Suv::operator=(const Suv & other){
 if (this != &other){ //protect from self-assignment
  m_awd = other.m_awd;
  delete m_emergencykit; //delete dynamic object first
  m_emergencykit = NULL; //object might outlive an exception
  m_emergencykit = new Emergencykit(*other.m_emergencykit);

  //handle base class members
  setEngine( other.getEngine() );
  delete [] m_plates; //delete dynamic memory first
  m_plates = NULL; //object might outlive an exception
  m_plates = new char [ strlen(other.m_plates)+1 ];
  strcpy(m_plates, other.m_plates);
 }
 return *this;
}
```

```
class Vehicle {
 public:
  Vehicle();
  Vehicle(const char * plates,
  const Engine & engine=Engine());
  Vehicle(const Vehicle & other);
  ~Vehicle();
  …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

```
class Suv : public Vehicle {
 public:
  …
 float driveInCityOffRoad(bool);
 private:
  bool m_awd;
  Emergencykit * m_emergencykit;
};
```

Working with Hierarchies

```
float Suv::driveInCityOffRoad(bool offroad){
 float milesThisTrip = 0;
 if (offroad && m_awd){
   /* required actions to drive offroad, etc… */
   cout<<"Suv "<<m_plates<<" offroad"<< endl;
 }
 else if (m_awd){
   /* required actions to drive in city with awd, etc… */
   cout<<"Suv "<<m_plates<<" awd in city"<< endl;
 }
 else{
   /* required actions to drive in city without awd, etc… */
   cout<<"Suv "<<m_plates<<" normal city drive"<< endl;
 }
 return milesThisTrip;
}
```

```
float Suv::move(){
  return (m_miles += driveInCityOffRoad( false ));
}
```

```
class Vehicle {
 public:
  Vehicle();
  Vehicle(const char * plates,
  const Engine& engine=Engine());
  Vehicle(const Vehicle & other);
  ~Vehicle();
   …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

```
class Suv : public Vehicle {
 public:
   …
 float driveInCityOffRoad(bool);
 private:
  bool m_awd;
  Emergencykit * m_emergencykit;
};
```

Working with Hierarchies

```
int main() {
  Vehicle vehicle0;
  Sedan sedan1("SEDAN1");
  Sedan sedan2("SEDAN2", true, 6, Engine(4.0));
  Suv suv1("SUV1");
  Suv suv2("SUV2", true, new Emergencykit, Engine(5.0));
  Vehicle * vehicle_Pt;
  vehicle_Pt = &vehicle0;  vehicle_Pt->move();
  Sedan * sedan_Pt;
  sedan_Pt = &sedan1;  sedan_Pt->move();
  sedan_Pt = &sedan2;  sedan_Pt->move();
  Suv * suv_Pt;
  suv_Pt = &suv1;  suv_Pt->move();
  suv_Pt = &suv2;  suv_Pt->move();
}
```

```
class Vehicle does not know how to move…
Sedan SEDAN1 automatic
Sedan SEDAN2 manual
Suv SUV1 normal city drive
Suv SUV2 awd in city
```

```
class Vehicle {
 public: …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

```
class Sedan : public Vehicle {
 public: …
  float driveInCity();
 private:
  bool m_manual;
  size_t m_gears;
};
```

```
class Suv : public Vehicle {
 public: …
  float driveInCityOffRoad(bool);
 private:
  bool m_awd;
  Emergencykit * m_emergencykit;
};
```

Inheritance

Working with Hierarchies

```
int main() {
  Vehicle vehicle0;
  Sedan sedan1("SEDAN1");
  Sedan sedan2("SEDAN2", true, 6, Engine(4.0));
  Suv suv1("SUV1");
  Suv suv2("SUV2", true, new Emergencykit, Engine(5.0));
  Vehicle * vehicles_index_array[5];
  vehicles_index_array[0] = &vehicle0;
  vehicles_index_array[1] = &sedan1;
  vehicles_index_array[2] = &sedan2;
  vehicles_index_array[3] = &suv1;
  vehicles_index_array[4] = &suv2;
  for (size_t i=0; i<5; ++i){
    vehicles_index_array[i]->move();
  }
}
```

```
class Vehicle does not know how to move…
class Vehicle does not know how to move…
class Vehicle does not know how to move…
class Vehicle does not know how to move…
class Vehicle does not know how to move…
```

```
class Vehicle {
 public: …
  void move();
 protected:
  float m_miles;
  char * m_plates;
 private:
  Engine m_engine;
};
```

```
class Sedan : public Vehicle {
 public: …
  float driveInCity();
 private:
  bool m_manual;
  size_t m_gears;
};
```

```
class Suv : public Vehicle {
 public: …
  float driveInCityOffRoad(bool);
 private:
  bool m_awd;
  Emergencykit * m_emergencykit;
};
```

Polymorphism

Achieving Polymorphic behavior

```
class Vehicle {
 public: …
  void move();
 protected:
  float m_miles;
  …
};
```

```
float Vehicle::move(){
  cout << "…" << endl;
  return m_miles;
}
```

```
class Vehicle {
 public: …
  virtual float move();
 protected:
  float m_miles;
  …
};
```

```
class Sedan : public Vehicle {
 public: …
  float move();
  float driveInCity();
  …
};
```

```
class Sedan : public Vehicle {
 public: …
  virtual float move();
  float driveInCity();
  …
};
```

```
class Suv : public Vehicle {
 public: …
  float move();
  float driveInCityOffRoad(bool);
  …
};
```

```
class Suv : public Vehicle {
 public: …
  virtual float move();
  float driveInCityOffRoad(bool);
  …
};
```

Working with Hierarchies

```
int main() {
  Vehicle vehicle0;
  Sedan sedan1("SEDAN1");
  Sedan sedan2("SEDAN2", true, 6, Engine(4.0));
  Suv suv1("SUV1");
  Suv suv2("SUV2", true, new Emergencykit, Engine(5.0));
  Vehicle * vehicles_index_array[5];
  vehicles_index_array[0] = &vehicle0;
  vehicles_index_array[1] = &sedan1;
  vehicles_index_array[2] = &sedan2;
  vehicles_index_array[3] = &suv1;
  vehicles_index_array[4] = &suv2;
  for (size_t i=0; i<5; ++i){
    vehicles_index_array[i]->move();
  }
}
```

```
class Vehicle does not know how to move…
Sedan SEDAN1 automatic
Sedan SEDAN2 manual
Suv SUV1 normal city drive
Suv SUV2 awd in city
```

```
class Vehicle {
 public: …
  virtual float move();
 protected:
  float m_miles;
  …
};
```

```
class Sedan : public Vehicle {
 public: …
  virtual float move();
  float driveInCity();
  …
};
```

```
class Suv : public Vehicle {
 public: …
  virtual float move();
  float driveInCityOffRoad(bool);
  …
};
```

Polymorphism

Achieving Polymorphic behavior

```
class Vehicle {
 public: …
  void move();
 protected:
  float m_miles;
  …
};
```

```
float Vehicle::move(){
  cout << "…" << endl;
  return m_miles;
}
```

```
class Vehicle {
 public: …
  virtual float move() = 0;
 protected:
  float m_miles;
  …
};
```

```
class Sedan : public Vehicle {
 public: …
  float move();
  float driveInCity();
  …
};
```

```
class Sedan : public Vehicle {
 public: …
  virtual float move();
  float driveInCity();
  …
};
```

```
class Suv : public Vehicle {
 public: …
  float move();
  float driveInCityOffRoad(bool);
  …
};
```

```
class Suv : public Vehicle {
 public: …
  virtual float move();
  float driveInCityOffRoad(bool);
  …
};
```

Working with Hierarchies

```
int main() {
  Vehicle vehicle0;
  Sedan sedan1("SEDAN1");
  Sedan sedan2("SEDAN2", true, 6, Engine(4.0));
  Suv suv1("SUV1");
  Suv suv2("SUV2", true, new Emergencykit, Engine(5.0));
  Vehicle * vehicles_index_array[4];
  vehicles_index_array[0] = &sedan1;
  vehicles_index_array[1] = &sedan2;
  vehicles_index_array[2] = &suv1;
  vehicles_index_array[3] = &suv2;
  for (size_t i=0; i<4; ++i){
    vehicles_index_array[i]->move();
  }
}
```

```
Sedan SEDAN1 automatic
Sedan SEDAN2 manual
Suv SUV1 normal city drive
Suv SUV2 awd in city
```

```
class Vehicle {
 public: …
  virtual float move() = 0;
 protected:
  float m_miles;
  …
};
```

```
class Sedan : public Vehicle {
 public: …
  virtual float move();
  float driveInCity();
  …
};
```

```
class Suv : public Vehicle {
 public: …
  virtual float move();
  float driveInCityOffRoad(bool);
  …
};
```

Managing Dynamic Memory

```
int * grades_array = NULL;
size_t size_grades_array;
cin >> size_grades_array;

try{
  grades_array = new int[size_grades_array];
}
catch(const std::bad_alloc & ex){
  cerr<<"Bad allocation of "<<size_grades_array<<"integer array…"<<endl;
  size_grades_array = 0;  //defensive
}
…
if (grades_array)
 for (size_t i=0; i<size_grades_array; ++i){ cin>>grades_array[i]; }
…
if (grades_array)
 for (size_t i=0; i<size_grades_array; ++i){ cout<<grades_array[i]; }
…

delete [] grades_array;
grades_array = NULL;
size_grades_array = 0; //defensive
```

Managing Dynamic Memory

```
int ** int_matrix = NULL;
size_t rows, cols; cin>>rows>>cols;
try{
  int_matrix = new int * [rows];
  for (size_t i=0; i<rows; ++i)
    int_matrix[i] = NULL;
  for (size_t i=0; i<rows; ++i){
    try{
      int_matrix[i] = new int [cols];
    }
    catch(const std::bad_alloc & ex){
      for (; i>=0; --i)
        delete [] int_matrix[i];
      throw;
    }
  }
}
catch(const std::bad_alloc & ex)
{  delete [] int_matrix; }
```

```
if (int_matrix){
  for (size_t i=0; i<rows; ++i){
    delete [] int_matrix[i];
  }
  delete [] int_matrix;
}
```

Managing Dynamic Memory

```
Car * inventory = NULL;
size_t size_inventory;
cin >> size_inventory;

try{
  inventory = new Car[size_inventory];
}
catch(const std::bad_alloc & ex){
  cerr<<"Bad allocation of "<<size_inventory<<"Car array…"<<endl;
  size_inventory = 0;  //defensive
}
…
if (inventory)
 for (size_t i=0; i<size_inventory; ++i){
  cin >> inventory[i];
 }
…
delete [] inventory;
inventory = NULL;
size_inventory = 0; //defensive
```

```
class Car {
public:
 …
private:
 Engine m_engine;
 Driver * m_driver;
 char m_id[ID_LEN];
 char * m_plates;
 const size_t m_serial;
 static size_t count;
};
```

CS-202

Time for Questions !