CS-202

C++ Classes – Overview and Examples

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 4th Project Deadline still stands for *next* Wednesday 2/27!

- PASS Sessions held Friday-Sunday-&-Monday-Tuesday, get all the help you need!
- 24-hrs delay after Project Deadline incurs 20% grade penalty.
- Past that, NO Project accepted. Better send what you have in time!

Today's Topics

C++ Classes Cheatsheet

- Declaration
- Members, Methods, Interface
- Implementation – Resolution Operator (`::`)
- Instantiation – Objects
- Object Usage – Dot Operator (`.`)
- Object Pointer Usage – Arrow Operator (`->`)
- Classes as Function Parameters, Pass-by-Value, by-(`const`)-Reference, by-Address
- Protection Mechanisms – `const` Method signature
- Classes – Code File Structure
- Constructor(s), Initialization List(s), Destructor
- `static` Members – Variables / Functions
- Class `friend`(s)
- Keyword `this`
- Operator Overloading

Overview and Examples

Class Cheatsheet

Declaration:

```
class Car {
  public:
    float addGas(float gallons);
    float getMileage();
    char m_licensePlates[9];
  protected:
    float m_gallons;
    float m_mileage;
  private:
    bool setEngineTiming(double[16]);
    double m_engineTiming[16];
};
```

Class (Type) Name

- Type Name is up to you to declare!

- Members in Brackets
- Semicolon

Conventions:

- Begin with Capital letter.
- Use **CamelCase** for phrases.
- General word for Class of Objects.

Class Cheatsheet

Declaration:

```
class Car  {
 public:
   float addGas(float gallons);
   float getMileage();
   char m_licensePlates[9];
 protected:
   float m_gallons;
   float m_mileage;
 private:
   bool setEngineTiming(double[16]);
   double m_engineTiming[16];
};
```

Access Specifiers

- Provide Protection Mechanism

Encapsulation - Abstraction:

- "Data Hiding"

Class Cheatsheet

Declaration:

```
class Car  {
  public:
    float addGas(float gallons);
    float getMileage();
    char m_licensePlates[9];
  protected:
    float m_gallons;
    float m_mileage;
  private:
    bool setEngineTiming(double[16]);
    double m_engineTiming[16];
};
```

Member Variables

- All necessary Data inside a single Code Unit.

Conventions:

- Begin with **m_<variable_name>**.

Encapsulation - Abstraction:

- Abstract Data Structure

Class Cheatsheet

Declaration:

```
class Car  {
  public:
   float addGas(float gallons);
   float getMileage();
    char m_licensePlates[9];
  protected:
    float m_gallons;
    float m_mileage;
  private:
   bool setEngineTiming(double[16]);
    double m_engineTiming[16];
};
```

Member Function / Class Methods

- All necessary Data & Operations inside a single Code Unit.

Conventions:

- Use **camelCase** (or **CamelCase**).

Encapsulation - Abstraction:

- Abstract Data Structure

Class Cheatsheet

Usual-case Class Interface Design:

```
class Car  {
public:
   float addGas(float gallons);
   float getMileage();
   bool setEngineTiming(double[16]);
private:
   char m_licensePlates[9];
   float m_gallons;
   float m_mileage;
   double m_engineTiming[16];
};
```

`public` Class Interface:

- Class Methods
- (Seldom) Class Data – *Immutable* usually.

`private` Class Access:

- Class Data
- Class (`private`) Methods

Class Interface to Member Data should "go through" Member Functions.

Class Cheatsheet

Class Implementation:

```
class Car  {
    …
    bool addGas(float gallons);
    float getMileage();
};

float Car::addGas(float gallons){
  /* actual code here */
}

float Car::getMileage(){
 /* actual code here */
}
```

An Implementation *needs* to exist for Class Methods

Scope Resolution Operator

(::)

- Indicates which Class Method this definition implements.

Class Cheatsheet

Class Instantiation – Default Construction:

```
<type_name> <variable_name>;
```

```
Car myCar;
```

Object

Create (Construct) a variable of specific Class type.

Will employ "*Default Constructor*"

- Compiler will auto-handle *Member Variables*' initialization !

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note: NOT to be confused with Value-Initialization:

```
Car myCar = Car();
```

- Handled very differently…

Class Cheatsheet

Class Object Usage:

```
<variable_name>.<member_name>;
```

Dot Operator – Member-of

(.)

- Which Object this Member references.

```
Car myCar;
float mileage = myCar.getMileage();
strcpy(myCar.m_licensePlates,"Gandalf");
```

Member Variables & Member Functions

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Class Object Pointers:

```
<type_name> * <variable_name_Pt>;

Car myCar;

Car * myCar_Pt;

myCar_Pt = &myCar;
(*myCar_Pt).getMileage();
```

Object

Pointer to Object

- Dereferencing to get to Object.
  Works the same as any pointer.

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Class Object Pointer Usage:

```
<variable_name_Pt>-><member_name>;
```

Arrow Operator – Member-access (->)

- Structure (Class) Pointer Dereference

```
Car myCar;
Car * myCar_Pt = &myCar;

myCar_Pt->getMileage();
strcpy(myCar_Pt->m_licensePlates,"Gandalf");
```

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Class Object Pointer Usage:

```
<variable_name_Pt>-><member_name>;
```

Arrow Operator – Member-access (->)

- Structure (Class) Pointer Dereference

Why?
Chaining Operator Precedence (. , ->)

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

```
(*(*(*topClass).subClass).subSubClass).method();
topClass->subClass->subSubClass->method();
```

Class Cheatsheet

Class Object in Function – By-Value:

```
Car myCar;
strcpy(myCar.m_licensePlates,"Gandalf");
printCapPlatesMileage(myCar);
cout << myCar.m_licensePlates;

void printCapPlatesMileage(Car car){
  char * lP = car.m_licensePlates;
  while (*lP = toupper(*lP)){ ++lP; }
  cout << car.m_licensePlates << endl;
  cout << car.getMileage() << endl;
}
```

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note:
Will work with Local Object Copy !

Class Cheatsheet

Class Object in Function – By-Reference:

```
Car myCar;
strcpy(myCar.m_licensePlates,"Gandalf");
printModifyCapPlates(myCar);
cout << myCar.m_licensePlates;

void printModifyCapPlates(Car & car){
  char * lP = car.m_licensePlates;
  while (*lP = toupper(*lP)){ ++lP; }
  cout << car.m_licensePlates << endl;
}
```

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note:

Will modify Object Data !

Class Cheatsheet

Class Object in Function – By-const-Reference:

```
Car myCar;
strcpy(myCar.m_licensePlates,"Gandalf");
printCapPlates(myCar);
cout << myCar.m_licensePlates;

void printCapPlates(const Car & car){
  char * lP = (char*)malloc(sizeof(
                car.m_licensePlates));
  strcpy(lP,car.m_licensePlates);
  char * lP_0 = lP;
  while (*lP = toupper(*lP)){ ++lP; }
  cout << lP_0 << endl;
}
```

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note:
Not allowed to modify Object Data !

Class Cheatsheet

Class Object in Function – By-Address:

```
Car myCar;
Car * myCar_Pt = &myCar;
strcpy(myCar_Pt->m_licensePlates,"Gandalf");
printModifyCapPlates(myCar_Pt);
cout << myCar.m_licensePlates;

void printModifyCapPlates(Car * car_Pt){
  char * lP = car_Pt->m_licensePlates;
  while (*lP = toupper(*lP)){ ++lP; }
   cout << car_Pt->m_licensePlates
        << endl;
}
```

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage();
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Note:
Will modify Object Data !

Class Cheatsheet

Protection Mechanisms – `const` Method signature:

A "promise" that Method doesn't modify Object

```
Car myCar;
cout << myCar.getMileage() << endl;
cout << myCar.addGas(10.0F) << endl;

float Car::getMileage() const {
  return m_mileage;
}
float Car::addGas(float gallons) {
  if (m_gallons += gallons > MAX_GALLONS)
    m_gallons = MAX_GALLONS;
  return m_gallons;
}
```

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage() const ;
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Protection Mechanisms – Access Specifiers:

`public`

Anything that has access to a **`Car`** Object (scope-wise) also has access to all **`public`** Member Variables and Functions.

- "Normally" used for Functions.
- Need to have at least one **`public`** Member.

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage() const ;
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Protection Mechanisms – Access Specifiers:

`private`

Members (Variables and Functions) that can ONLY be accessed by Member Functions of the ***`Car`*** Class.

- Cannot be accessed in **`main()`**, in other files, or by other functions.

- If not specified, Members default to **`private`**.
- Should specify anyway – good coding practices!

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage() const ;
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Protection Mechanisms – Access Specifiers:

protected

Members that can be accessed by:

- Member Functions of the ***Car*** Class.
- Member Functions of any *Derived* Class.

A *Derived* Class

```
class Hybrid : Car  {
  …
  float gasToElectricRatio();
};

float Hybrid::gasToElectricRatio(){
    if (m_gallons < …){ return …; }
}
```

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage() const ;
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Member Functions – Accessors ("Getters")

Name starts with **get**, ends with Member name.
Allows retrieval of non-**public** Data Members.

```
float Car::getMileage() const {
  return m_mileage;
}
```

Note: Don't generally take in arguments.

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage() const ;
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Member Functions – Mutators ("Setters")

Name starts with **set**, ends with Member name.
Controlled changing of non-**public** Data Members.

```
bool Car::setEngineTiming(double t_in[16]){
  for (int i=0;i<16;++i){
    if (t_in[i]<… || t_in[i]>…){ return false; }
  }
  for (int i=0;i<16;++i){
    m_engineTiming[i]=t_in[i];
  }
  return true;
}
```

Note: In simple case, don't **return** anything (**void**).
In controlled setting, return success/fail (**bool**).

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage() const ;
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Member Functions – Facilitators ("Helpers")

Provide support for the Class's operations.

```
float Car::addGas(float gallons) {
  if (m_gallons += gallons > MAX_GALLONS)
    m_gallons = MAX_GALLONS;
  return m_gallons;
}
```

Note:
`public` if generally called outside Function.
`private`/`protected` if only called by Member Functions.

```
class Car  {
 public:
  float addGas(float gallons);
  float getMileage() const ;
  char m_licensePlates[9];
 protected:
  float m_gallons;
  float m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[16];
};
```

Class Cheatsheet

Classes and Code File Structure

Class Header File: `Car.h`

```
#ifndef CAR_H_
#define CAR_H_
#define NUMVALVES 16
class Car  {
 public:
  float addGas(float gallons);
  float getMileage() const ;
  char m_licensePlates[9];
 protected:
  float m_gallons, m_mileage;
 private:
  bool setEngineTiming(double[16]);
  double m_engineTiming[NUMVALVES];
};
#endif
```

Class Source File: `Car.cpp`

```
#include <iostream>
#include "Car.h"
#define MAX_GALLONS 20.0
float Car::getMileage() const {
  return m_mileage;
}
float Car::addGas(float gallons) {
  if (m_gallons += gallons > MAX_GALLONS)
    m_gallons = MAX_GALLONS;
  return m_gallons;
}
bool Car::setEngineTiming(double t_in[16]){
  for (int i=0;i<16;++i){
    if (t_in[i]<… || t_in[i]>…) return false;
  }
  for (int i=0;i<16;++i){
    m_engineTiming[i]=t_in[i];
  }
  return true;
}
```

Class Cheatsheet

Classes and Code File Structure

Note: Compile all your source (.cpp) files together with
`g++ car_program.cpp Car.cpp`

Program File: **car_program.cpp**

```
#include <iostream>
#include <...>
#include "Car.h"
int main(){
  Car myCar;
  Car * myCar_Pt = &myCar;
  strcpy(myCar_Pt->m_licensePlates,"Gandalf");
  printCapPlates(myCar_Pt);
  cout << myCar.m_licensePlates << endl;
  cout << myCar.getMileage() << endl;
  cout << myCar.addGas(10.0F) << endl;
  return 0;
}
```

Class Cheatsheet

Constructor(s):

Special Function:

- Prototype is named same as Class.
- Has no **`return`** type.

"Constructors have no names and cannot be called directly."
"They are invoked when instatintation takes place."
"They are selected according to the rules of initialization."

- Constructors that may be called without any argument are *Default* constructors.
- Constructors that take another Object of the same type as the argument are *Copy* (or *Move*) constructors.

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char m_licensePlates[PLT];
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Default (empty) **ctor**:

- Function Prototype:

```
Car();
```

- Function Definition:

```
Car::Car(){
  strcpy(m_licensePlates, DFT_PLTS);
  m_gallons = DFT_GLNS;
  m_mileage = 0;
  m_engineTiming = _def_DFT_TIM;
}
```

Note:

- The compiler will *automatically synthesize* a Default Constructor if no *user-provided* one is specified.

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char m_licensePlates[PLT];
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Overloaded (parametrized) **ctor**:

- Function Prototype (w/ Default Parameters):

```
Car(char licPlts[PLT],
    float glns=DFT_GLNS, float mlg=0,
    const double engTim[VLV]=DFT_TIM);
```

- Function Definition (no Default Parameters):

```
Car::Car(char licPlts[PLT], float glns,
    float mileage, const double engTim[VLV]){
  strcpy(m_licensePlates, licPlts);
  m_gallons = glns;
  m_mileage = mileage;
  for (int i=0; i<VLV; ++i)
    m_engineTiming[i] = engTim[i];
}
```

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char m_licensePlates[PLT];
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Overloaded (parametrized) **ctor**:

- Function Prototype (w/ Default Parameters):

```
Car(char licPlts[PLT],
    float gls=DFT_GLNS, float mlg=0,
    const double engTim[VLV]=DFT_TIM);
```

- Function Definition (no Default Parameters):

```
Car::Car(char licPlts[PLT], float gls,
    float mileage, const double engTim[VLV]){
  /* num of args resolves implementation */
}
```

Note:

- If you define an *Overloaded* Constructor the compiler will *not* automatically synthesize a *Default* one.
- A good coding practice is to always define a 1-liner Default (empty) Constructor as well, as a lot of C++ functionalities depend on the existence of an *accessible* class *Default* Constructor.

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],
 float gls=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char m_licensePlates[PLT];
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Overloaded (parametrized) **ctor**:

- Function Prototype (w/ Default Parameters):

```
Car(char licPlts[PLT],
    float glns=DFT_GLNS, float mlg=0,
    const double engTim[VLV]=DFT_TIM);
```

- Sequential Interpretation of Default Params:

```
Car car("Gandalf", 5. ,0. , new double[VLV]
    {0.,1.,2.,3.,…,3.,0.,1.,2.});
```

or

```
Car car("Gandalf", 5. ,0.);
```

or

```
Car car("Gandalf", 5.);
```

or

```
Car car("Gandalf");
```

No Parameter skipping !

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char m_licensePlates[PLT];
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Overloaded (parametrized) **ctor**:

Caveat:

- Function Prototype(s) for different overloaded versions must not produce same signatures:

```
Car(char licPlts[PLT], float glns);
Car(char[PLT], float);

Car(char licPlts[PLT], float mlg);
Car(char[PLT], float);
```

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char m_licensePlates[PLT];
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Copy (class-object) **ctor**:

- Function Prototype:

```
Car(const Car &car);
```

- Function Definition:

```
Car::Car(const Car & car){
 strcpy(m_licensePlates,car.m_licensePlates);
  m_gallons = car.m_gallons;
  m_mileage = car.m_mileage;
  for (int i=0; i<VLV; ++i)
    m_engineTiming[i] = car.m_engineTiming[i];
}
```

Same Class:

- Access to **private** Members of input Object.

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char m_licensePlates[PLT];
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Copy (class-object) **ctor**:

- The compiler will (implicitly) provide a ***Shallow****-Copy* Constructor if none is specified.

Class now contains raw Pointer Member (**char***):

- Handle memory allocation for Member Data.

```
Car::Car(){
  m_licensePlates = (char*)malloc(PLT);
  /* rest of Default ctor statements */
}
Car::Car(const char* licPlts, float glns,
    float mileage, const double engTim[VLV]){
  m_licensePlates = (char*)malloc(PLT);
  /* rest of Overloaded ctor statements */
}
```

```
class Car  {
public:
 Car();
 Car(const char * licPlts,
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);

 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char * m_licensePlates;
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

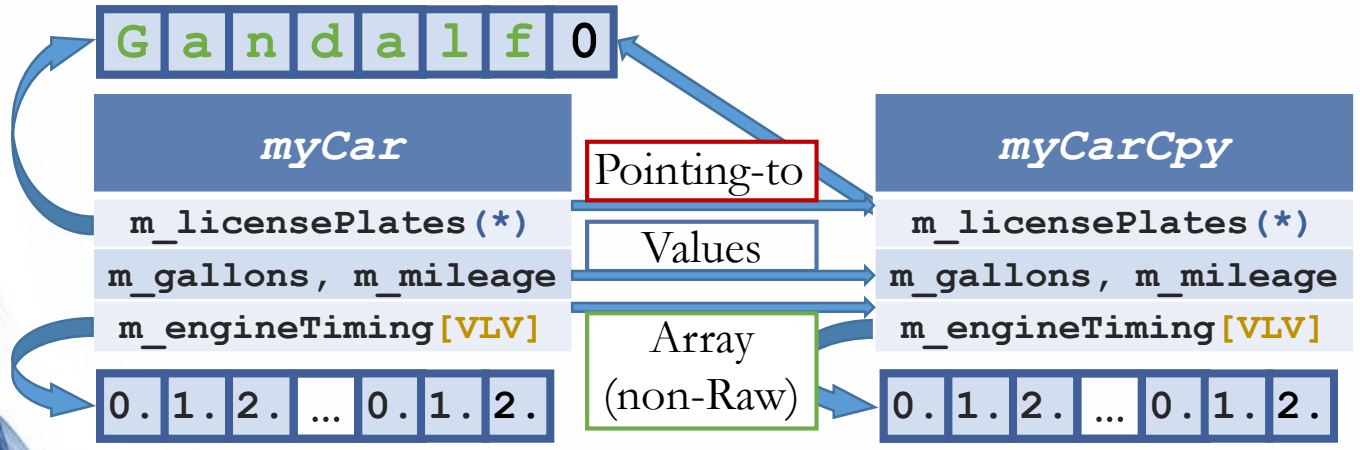
Class Cheatsheet

Copy (class-object) **ctor**:

- The compiler will (implicitly) provide a ***Shallow***-*Copy* Constructor if none is specified.

Shallow-*Copy* **ctor** copies raw Pointer, not Data!

```
Car myCar("Gandalf");
Car myCarCpy(myCar);
```




```
class Car  {
public:
 Car();
 Car(const char * licPlts,
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);

 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char * m_licensePlates;
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Copy (class-object) **ctor**:

- Explictly Implement ***Deep****-Copy* Constructor.

Deep*-Copy* **ctor** will allocate-&-copy Data!

Function Definition:

```
Car::Car(const Car &car){
  m_licensePlates = (char*)malloc(PLT);
  strcpy(m_licensePlates,car.m_licensePlates);
  m_gallons = car.m_gallons;
  m_mileage = car.m_mileage;
  for (int i=0; i<VLV; ++i)
    m_engineTiming[i] = car.m_engineTiming[i];
}
```

```
class Car  {
public:
 Car();
 Car(const char * licPlts,
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char * m_licensePlates;
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Copy (class-object) **ctor**:

```
Car myCar("Gandalf");
Car myCarCpy(myCar);
myCar.m_licensePlates[4] = 0;
cout << myCar.m_licensePlates << ","
     << myCarCpy.m_licensePlates << endl;
```

***Shallow**-Copy* **ctor** will only copy raw Pointer:

- Output: **Gand,Gand**

Explicit ***Deep**-Copy* **ctor** will allocate-copy Data:

- Output: **Gand,Gandalf**

Note:

- Is ***Deep**-Copy*ing always desired? No, C++11 introduces *Move* **ctor**.
  However user-based raw Pointer solution(s) are usually unsafe !

```
class Car  {
public:
 Car();
 Car(const char * licPlts,
 float glns=DFT_GLNS, float mlg=0,
 const double engTim[VLV]=DFT_TIM);
 Car(const Car &car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char * m_licensePlates;
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
};
```

Class Cheatsheet

Initialization List(s) (`ctor` Definition only):

- By-name Initialization of Data Members.
- Allows *Instantiation-time* Initialization.

```
Car::Car(const char * licPlts, float glns,
      float mlg, int fId,
      const double engTim[VLV]) :
   m_gallons( glns ) , m_mileage( mlg ) ,
   m_frameId( fId ) {
  // m_frameId = fId; wouldn't work (const)!
}
```

Note: With a `const` Member, needs to exist an *Initialization List* for *every* Constructor !

```
Car myCar("Gandalf",0,0,11000); //11000 years
```

```
class Car  {
public:
 Car();
 Car(const char* licPlts,float glns
 =DFT_GLNS,float mlg=0,int fId=NO_F
 ,const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addGas(float gallons);
 float getGallons() const ;
 float getMileage() const ;
 char * m_licensePlates;
protected:
 float m_gallons;
 float m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
 const int m_frameId;
};
```

Class Cheatsheet

Initializer List(s):

- Class-with-*Composition* Initialization.

```
class Driver {
  public:
    Driver(){}
    Driver(char name[PLT], int fId);
  private:
    char m_name[PLT];
    Car m_car;
};

Driver::Driver(const char* name, int fId=NO_F) :
    m_name(name) , m_car(name,0,0,fId) {
  // Driver & m_car instantiated & initialized
}
```

`ctor`-in-`ctor` Call

`Driver` `ctor` Parameter re-used for `Car` `ctor`.

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],float glns
 =DFT_GLNS,float mlg=0,int fId=NO_F
 ,const double engTim[VLV]=DFT_TIM);
 Car(const Car & car);
 float addG/M(float gal/mil);
 float getG/M() const ;
 char m_licensePlates[PLT];
protected:
 float m_gallons, m_mileage;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
 const int m_frameId;
};
```

Class Cheatsheet

Delegating Constructor (C++11):

- Can have one `ctor` invoke another `ctor`.

```
Car(char lP[PLT], int fId) :
 Car(lP, DFT_GLNS,0, fId, DFT_TIM)
{  /* delegating ctor body … */  }
```

Default Member Initialization (C++11):

- Can set default Member values in Declaration.
- Any *Initializer List* appearance of the member will hold precedence over this default.

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],float glns
 =DFT_GLNS,float mlg=0,int fId=NO_F
 ,const double engTim[VLV]=DFT_TIM);
 Car(char lP[PLT], int fId) :
Car(lP,DFT_GLNS,0,fId,DFT_TIM){ … }
 float addG/M(float gal/mil);
 float getG/M() const ;
 char m_licensePlates[PLT] = "Gdf";
protected:
 float m_gallons = DFT_GLNS;
 float m_mileage = 0;
private:
 bool setEngineTiming(double[VLV]);
 double m_engineTiming[VLV] = {…};
 const int m_frameId;
};
```

Class Cheatsheet

`static` Data Members:

- Class state properties, not bound to an Object.
- Manipulated via the Class or an Object (if not `private`).

```
Car::Car(){ s_carFactoryCnt++; } //dflt ctor

cout << Car::s_carFactoryCnt;     //via class
Car myCar1; //call dflt ctor, increment cnt
cout << myCar1.s_carFactoryCnt;   //via object
```

`static` Member Function:

- Can only manipulate & address `static` Data Members and `static` Member Functions.

```
Car myCar2; //call dflt ctor, increment cnt
cout << Car::getCarFactoryCnt() << "==" <<
     << myCar1.getCarFactoryCnt() << "==" <<
     << myCar2.getCarFactoryCnt() ;  //2==2==2
```

```
class Car  {  //Class Header
public:
 Car();
 Car(char licPlts[PLT],float glns
 =DFT_GLNS,float mlg=0,int fId=NO_F
 ,const double engTim[VLV]=DFT_TIM);
 …
 static int getCarFactoryCnt();
private:
 // declaration of static member
 static int s_carFactoryCnt;
};
```

```
#include <Car.h> //Class Source

// definition of static member
int Car::s_carFactoryCnt = 0;
int Car::getCarFactoryCnt(){
  return Car::s_carFactoryCnt;
} …
```

Class Cheatsheet

`static` Local Variables in Class Methods:

- Statically allocated data.
- Initialized the first time Class Function block is entered.
- Lifetime until program exits!

```
float Car::addG(float gallons){
  static int refill_cnt = 0;
  cout<<"Refilled "<< ++refill_cnt <<" times"<<endl;
  m_gallons += gallons;
}

Car myCar1, myCar2;
myCar1.addG(10.0);
myCar2.addG(10.0);
```

Output: `Refilled 1 times`

Output: `Refilled 2 times`

Notes (Why is it usually such a "bad" design choice):

- Aliasing! The same variable is referenced within a member function that is to be called by different Calling Objects!
- Visible only in Function block (of no general use to the Class) !

```
class Car  {
public:
 Car();
 Car(char licPlts[PLT],float glns
 =DFT_GLNS,float mlg=0,int fId=NO_F
 ,const double engTim[VLV]=DFT_TIM);
 Car(const Car &car);
 float addG/M(float gallons);
 float getG/M() const ;
 static int getCarFactoryCnt();
 char m_licensePlates[PLT];
protected:
 float m_gallons, m_mileage;
private:
 bool getEngineTiming(double[VLV]);
 double m_engineTiming[VLV];
 const int m_frameId;
 static int s_carFactoryCnt;
};
```

Class Cheatsheet

Operator Overloading – **non-Member** of Class.

- **Unary** Operator(s):

```
const Money operator-(const Money & mn)
{ return Money(-mn.getD(),-mn.getC()); }
Money myMoney(99,25), notMyMoney = - myMoney;
```

- **Binary** Operator(s):

```
bool operator==(const Money& mn1, const Money& mn2)
{ return mn1.getD()==mn2.getD() && mn1.getC() == mn2.getC(); }

const Money operator+(const Money& mn1, const Money& mn2)
{ return Money(mn1.getD()+mn2.getD(),mn1.getC()+mn2.getC()); }
Money myMoney(99,25), yourMoney(0,75);
bool ourMoneyEqual = myMoney == yourMoney;
Money ourMoney = myMoney + yourMoney;
```

return: a **const** *Unnamed* Class Object

```
class Money{
public:
 Money();
 Money(int dollars,
       int cents=0);
 Money(const Money & m);
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars;
 int m_cents;
};
```

Note:
Operator(s) should handle Class specifications (e.g. prevent ***m_cents*** rollover)

Class Cheatsheet

Operator Overloading – **Class Member** Function.

- **Assignment** Operator (half the story, the rest for later) :

```
void Money::operator=(const Money& mn)
{ m_dollars = mn.m_dollars; m_cents = mn.m_cents; }
Money myMoney(99,25), myMoneyAgain = myMoney;
```

A Class method, like saying: `myMoneyAgain.operator=(myMoney);`

Note: If none specified, compiler creates a default Assignment Operator (***Member***-*Copy)* for Class Objects. *Remember*: ***Shallow***-Copy vs ***Deep***-Copy.

- **Binary** Operator(s):

```
const Money Money::operator+(const Money& mn) const
{ return Money(m_dollars+mn.m_dollars, m_cents+mn.m_cents); }
Money myMoney(99,25), yourMoney(0,75);
Money ourMoney = myMoney + yourMoney;
```

Calling Object is like 1st parameter: `myMoney.operator+(yourMoney);`

```
class Money{
public:
 Money();
 Money(int dollars,
       int cents=0);
 Money(const Money & m);
 void Money operator=
(const Money & m);
 const Money operator+
(const Money & m) const;
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars;
 int m_cents;
 char * m_owner;
};
```

Class Cheatsheet

- Operator Overloading – Both versions (*Ambiguous*):

```
const Money operator+(const Money &a,const Money &b)
{  return Money(1);  }  //non-Member
const Money Money::operator+(const Money &b) const
{  return Money(2);  } //Class Member
warning: ISO C++ says that these are ambiguous …
Money m1,m2, m3 = m1 + m2;
Money m4 = m1 .operator+ ( m2 );
```

Result: 1
Result: 2

- Operator Overloading – Both versions (*Different Calls*):

```
const Money operator-(const Money & mn)
{ return Money(-mn.getD(), -mn.getC()); }
const Money operator-(const Money & m) const
{ return Money(m_dollars-mn.m_dollars, m_cents-mn.m_cents); }
Money m5 = - m1 ;      //Unary call
Money m6 = m1 - m2 ;   //Binary call
```

```
class Money{
public:
 Money();
 Money(int dollars,
       int cents=0);
 Money(const Money & m);
const Money operator+
(const Money & m) const;
const Money operator-
(const Money & m) const;
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars;
 int m_cents;
};
```

Class Cheatsheet

Operator Overloading

- Return by-`const`-Value

```
const Money Money::operator+(const Money & mn)const{
  return Money(m_dollars + mn.m_dollars,
               m_cents   + mn.m_cents);
}
```

Why `const`-Value ?

```
Money a(4, 50), b(3, 25), c(2, 10);
```

```
(a + b);
```

Evaluates to: *Unnamed* Object

```
c = (a + b);
```

OK…

```
(a + b) = c;
```

No !!!

Prevents (&protects) us from altering the returned value…

```
class Money{
public:
 Money();
 Money(int dollars,
       int cents=0);
 Money(const Money & m);
 void operator=
(const Money & m);
 const Money operator+
(const Money & m) const;
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars;
 int m_cents;
};
```

Class Cheatsheet

Operator Overloading

- Return by-`const`-Reference (?)

```
const Money& Money::operator+(const Money& mn) const
{ return Money(m_dollars + mn.m_dollars,
               m_cents   + mn.m_cents);   }
```

```
warning: returning reference to temporary.
```

- Makes a temporary Object, goes out of scope!

```
Money a(4, 50), b(3, 25);
const Money* ab_Pt = &(a + b);

cout << ab_Pt->getD()
<<","<< ab_Pt->getC();
```

```
7
75
```

No !
This is UNSAFE !

```
class Money{
public:
 Money();
 Money(int dollars,
       int cents=0);
 Money(const Money & m);
 void Money operator=
(const Money & m);
 const Money & operator+
(const Money & m) const;
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars, m_cents;
};
```

Function `return` does not guarantee an immediate *Stack* frame wipe!

Note: Especially if the return type is *not* a `const`-Reference ! (…)

Class Cheatsheet

Operator Overloading

- Return by-Reference – Operator (`[]`)

Returned: `<type_id> &`, internal Member Reference.

```
int & Money::operator[](unsigned int index)
{ return m_transID [ index ]; }
```

- Accessing (`private`) Data Member by-Reference:

```
Money hugeCheck(1000000);
int transCnt = 0;
hugeCheck [ transCnt++ ] = BANK_TRANS;
hugeCheck [ transCnt++ ] = BRIBE_TRANS;
hugeCheck [ transCnt++ ] = BANK_TRANS;
if (hugeCheck [ 1 ] == BRIBE_TRANS)
{ cout << "Illegal Activity!"; }
```

Write-to

Read-from

```
class Money{
public:
 Money();
 Money(int dollars,
       int cents=0);
 Money(const Money &m);
 int & operator[](
     unsigned int index);
 const Money& operator+
(const Money & m) const;
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars, m_cents;
 int m_transID[T_HIST];
};
```

Class Cheatsheet

Operator Overloading w/ Cascading

- Return by-Reference – Operator(s) (<<) , (>>)

Returned: <i/o>stream &, Reference to passed 1st Parameter.

```
ostream & operator<<(ostream & os, const Money & mn){
   os << "$" << mn.m_dollars << "." << mn.m_cents;
   return os;
}

istream & operator>>(istream & is, Money & mn){
   char dollar, point;
   is >> dollar >> mn.m_dollars >> point  >> mn.m_cents;
   return is;
}
```

Note: Non-Member **friend** functions granted **private** Data access.

Example:

```
Money myMoney;
cin >> myMoney;
```

w\ Cascading:

```
cout << "I have: " << myMoney << "right now";
```

```
class Money{
public:
 Money();
 Money(int dollars,
       int cents=0);
 Money(const Money & m);
 friend ostream &
 operator<<(ostream & os,
            const Money & m);
 friend istream &
 operator>>(istream & is,
                  Money & m);
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars, m_cents;
};
```

Class Cheatsheet

Operator Overloading w/ Cascading

- Return by-Reference – Assignment Operator (=)

Returned: `<class_type> &`, Reference to Calling Object.

```
Money & Money::operator=(const Money & m){
   this->m_dollars = m.m_dollars;
   this->m_cents = m.m_cents;
   this->output();
   return *this;
}
```

`this` : A pointer to the currently Calling Object inside a Member Function

Example:

```
Money moneyPack1,moneyPack2, moneyPack3(49,99);
moneyPack1 = moneyPack2 = moneyPack3;
```

Chaining Assignment Operator by `return`ing Calling Object Reference

Output:
```
$49.99
$49.99
```

```
class Money{
public:
 Money();
 Money(int dollars,
       int cents=0);
 Money(const Money & m);
 Money & operator=(const
                   Money & m);
 void output();
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars, m_cents;
};
```

Class Cheatsheet

Overloading Pre-Increment Operator(s) (++) , (--):

- No arguments (for compiler *disambiguation*).

```
Money & Money::operator++(){
  m_cents++; … //mutates calling object
  return *this;
}
```

Note:
Modifies calling Object and **return**s a Reference to it.
No Object Copy operation!

```
Money myMoney(0,99);
Money myMoreMoney = ++ myMoney;
```

{100,0} {100,0}

```
class Money{
public:
 Money();
 Money(int d, int c=0);
 Money(const Money & m);
 Money & operator++();
 Money & operator--();
 Money operator++(int);
 Money operator--(int);
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars, m_cents;
};
```

Class Cheatsheet

Overloading Post-Increment Operator(s) (++) , (--):

- A dummy `int` argument (for compiler *disambiguation*).

```
Money Money::operator++(int dummy){
   Money moneyCopy(*this);
   this->m_cents++; … //mutates calling object
   return moneyCopy;
}
```

Note: Keeps a Copy of calling Object to `return` and then modifies calling Object (same as before).

```
Money myMoney(0,99);
Money mySameMoney = myMoney ++;
```

{99,0} {100,0}

```
class Money{
public:
 Money();
 Money(int d, int c=0);
 Money(const Money & m);
 Money & operator++();
 Money & operator--();
 Money operator++(int);
 Money operator--(int);
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars, m_cents;
};
```

Keyword this

Checking if the Calling Object is *exactly* the same as the Object passed as argument!

```
bool Money::thisCheck(const Money & m){
  if (this == &m)
    return true;
  else
    return false;
}
```

- Usual Application: Protect from self-Assignment

```
Money & Money::operator=(const Money & m){
  if (this != &m){ //check if trying to assign from self
    //perform assignment from m to calling object
    //only if other object m is a separate object
  }
  return *this;  //return calling object by-Reference
}
```

```
class Money{
public:
 Money();
 Money(int d, int c=0);
 Money(const Money & m);
 Money & operator=(const
                   Money & rhs);
 bool thisCheck(const
                Money & m);
 void setD/C(int dc);
 int getD/C() const;
private:
 int m_dollars, m_cents;
};
```

CS-202

Time for Questions !