CS-202

C++ Classes – Constructor(s) (Pt.2)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (9:00-12:50)	
	CLASS		CLASS	
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	

Your 3rd Project Deadline is this Wednesday 9/20.

- PASS Sessions held Monday-Tuesday, get all the help you may 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)
- Destructor

Initialization List(s)

`static` Members – Variables / Functions

Operator(s)

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.
- **`mixedCase`** 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:

- Capitalize first letter.

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

`private` Class Access:
- Class Data

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 - Implicit:

```
<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];
};
```

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 (->)

- 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 (->)

- 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 (tin[i]<… || tin[i]>…) return false;
  }
  for (int i=0;i<16;++i){
    m_engineTiming[i]=tin[i];
  }
  return true;
}
```

Class Cheatsheet

Classes and Code File Structure

Program File: `car_prog.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;

   cout << myCar.GetMileage() << endl;
   cout << myCar.AddGas(10.0F) << endl;
   return 0;
}
```

Class Cheatsheet

Constructor(s):

Special Function:

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

"Constructors have no names and cannot be called directly."
"They are invoked when initialization 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* and *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 (implicitly) provide a *Default* Constructor if none 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 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]){
  /* num of args resolves implementation */
}
```

Note:
If you define an *Overloaded* Constructor the compiler will not automatically generate a *Default*.

```
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);
```

- 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**:

- Function Prototype(s) of different 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];
};
```

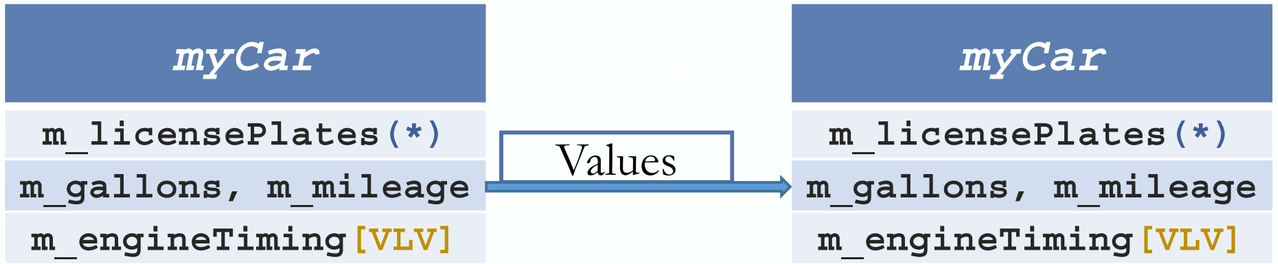
Class Cheatsheet

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];
};
```

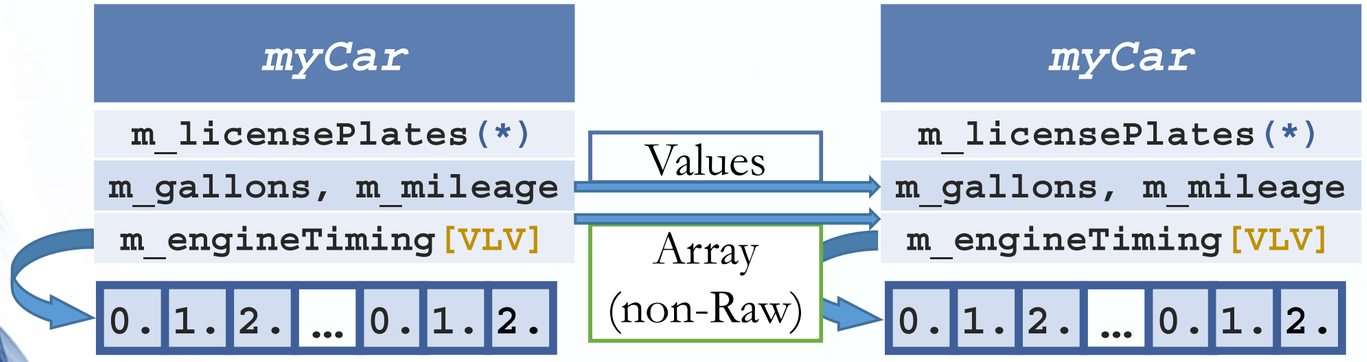
Class Cheatsheet

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

- 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];
};
```

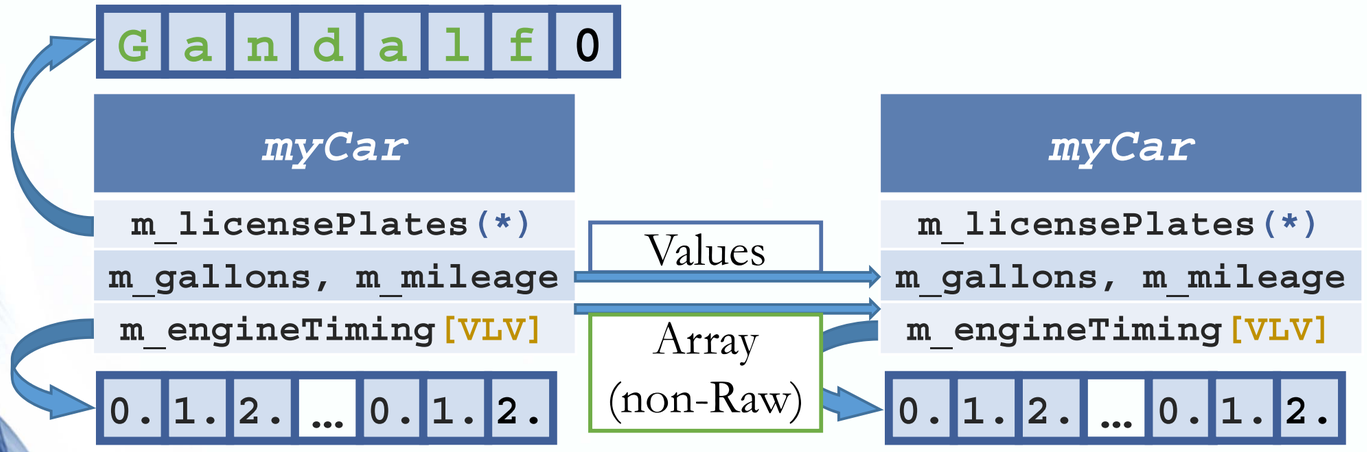
Class Cheatsheet

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

- 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];
};
```

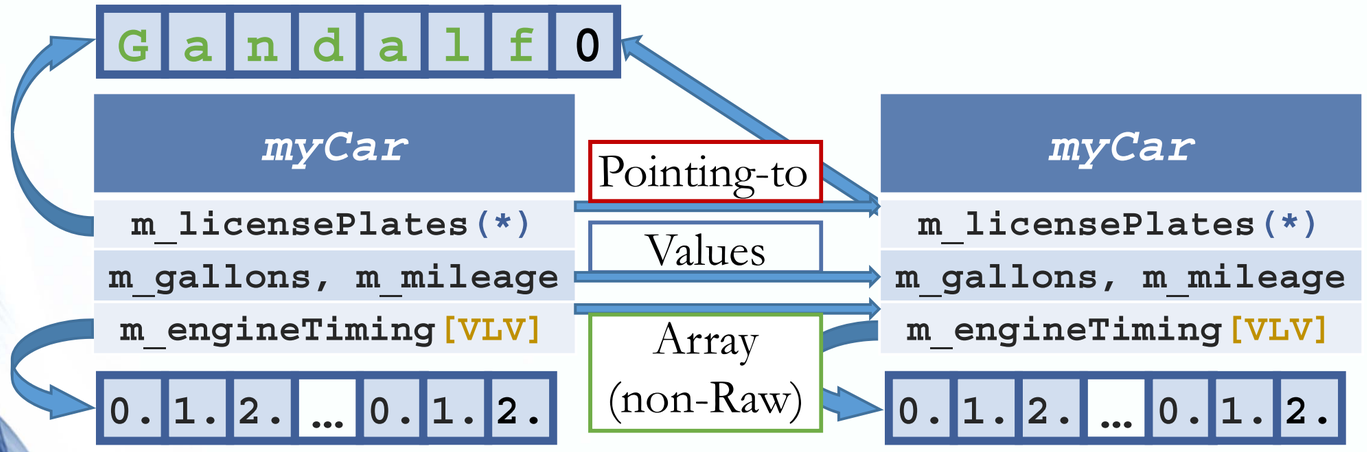
Class Cheatsheet

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

- 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:

- Always undesired? No, C++11 has *Move* **ctor**.
  However user-based raw Pointer solution(s) are 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];
};
```

Constructor(s)

Implementations

Initialization List(s)

Syntax (in Function Implementation only):

- Comma (,) separated list following colon (:).
- After Function Parameter List – parentheses.
- Initializes members by-Name (can use `ctor` Params).

```
Date::Date(int month, int day, int year) :
      m_month(month) ,
      m_day(day) ,
      m_year(year)
{
   /* Overloaded constructor statements */
}
```

```
class Date{
 public:
 Date();
 Date(int month,
  int day=DFT_D,
  int year=DFT_Y);
 Date(const Date &date);
 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month, m_day,
      m_year;
};
```

Constructor(s)

Implementations

Initialization List(s)

Syntax (in Function Implementation only):

- Comma-separated list following colon (:)

```
Date::Date(int month, int day, int year) :
      m_month(month) , m_day(day) , m_year(year) {
  /* no more statements necessary for init */
}
```

Alternative Implementation to assignment statements:

```
Date::Date(int month, int day, int year){
  m_month = month;
  m_day = day;
  m_year = year;
}
```

```
class Date{
 public:
 Date();
 Date(int month,
  int day=DFT_D,
  int year=DFT_Y);
 Date(const Date &date);
 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month, m_day,
     m_year;
};
```

Constructor(s)

Implementations

Initialization List(s)

Syntax (in Function Implementation only):

- Special Purpose – Define Values at *Instantiation*-time.

```
Date::Date(int m, int d, int y, bool gregorian) :
      m_month(m), m_day(d), m_year(y),
      m_gregorian(gregorian)                    {
}
```

Assignment statements Not at *Instantiation*-time:

```
Date::Date(int m, int d, int y, bool gregorian){
  m_month = m;  m_day = d; m_year = y;
  m_gregorian = gregorian;
}
```

`const` Variable:
Not allowed !

```
class Date{
 public:
 Date();
 Date(int month,
  int day=DFT_D,
  int year=DFT_Y,
  bool gregorian=true);
 Date(const Date &date);

 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month, m_day,
     m_year;
 const bool m_gregorian;
};
```

Constructor(s)

Implementations

Default Member Initialization (since C++11):
Syntax (in Class Declaration only)

```
class <class_name> {   …
    <type_id> m_var1 = const_literal_val;
}
```

Example (no Constructors defined for ***Date***):

```
Date myDate;
cout<<myDate.GetM()<<myDate.GetD()<<myDate.GetY();
      DFT_M           DFT_D           DFT_Y
```

Note:
Ignored if it also appears in an *Initializer List* !

```
Date():m_year(2457797),m_gregorian(false){}
```

```
class Date{
 public:
 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month = DFT_M;
 int m_day = DFT_D;
 int m_year = DFT_Y;
 const bool m_gregorian
              = true;
};
```

Constructor(s)

Implementations

Delegating Constructor (since C++11):

➢ Function Prototype (delegation to other Class `ctor`).

```
Date(bool gregorian) :
     Date(DFT_M, DFT_D, DFT_Y, gregorian);
```

Default `ctor` (no arguments list):

```
Date dateDftCtor;
```

Parametrized `ctor`:

```
Date dateParam(DFT_M, DFT_D, DFT_Y, true);
Date dateDftParam(DFT_M);
```

Delegating `ctor`:

```
Date dateDeleg(true);
```

```
class Date{
 public:
 Date();
 Date(int month,
  int day=DFT_D,
  int year=DFT_Y,
  bool gregorian=true);
 Date(bool gregorian):
  Date(DFT_M,DFT_D,DFT_Y
       gregorian);
 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month, m_day,
     m_year;
 const bool m_gregorian;
};
```

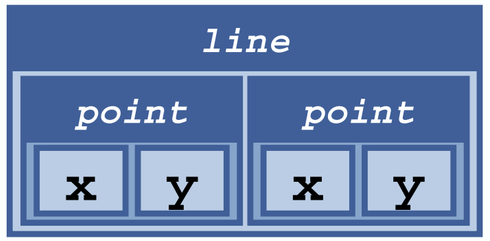
Constructor(s)

Aggregate Class Constructor(s)

Aggregation:

- Objects can hold other Objects!
- "Has-a" relationship.

Remember `struct`s:



Example:

- Class "has a" `private` Data Member of another Class-type.

```
class Vacation {
    …
    private:
    Date m_startDay;
};
```

```
class Date{
 public:
 Date();
 Date(int month,
  int day=DFT_D,
  int year=DFT_Y,
  bool gregorian=true);
 Date(const Date &date);

 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month, m_day,
     m_year;
 const bool m_gregorian;
};
```

Constructor(s)

Aggregate Class Constructor(s)

Initialization List(s)

Aggregate Class Initialization:

```
class Vacation{
    public:
    Vacation(int month, int day, int numDays);
    private:
    Date m_startDay;
    int m_tripLength;
};

Vacation::Vacation(int m, int d, int numDays) :
      m_startDay(m, d) , m_tripLength(numDays) {
   /* constructor code, m_startDay initialized !*/
}
```

Implicit Call to `Date ctor` at `Vacation` *Initialization*-time !

```
class Date{
 public:
 Date();
 Date(int month,
  int day=DFT_D,
  int year=DFT_Y,
  bool gregorian=true);
 Date(const Date &date);

 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month, m_day,
     m_year;
 const bool m_gregorian;
};
```

Static Member(s)

Static Variables

General:

- Local scope, but persist in memory.

```
int nonStaticLocal(){
  int a = 0;
  ++a;
  return a;
}
```

Destroyed when function returns.

```
int b = nonStaticLocal();
int c = nonStaticLocal();
int d = nonStaticLocal();
```

1
1
1

```
int staticLocal(){
  static int a = 0;
  ++a;
  return a;
}
```

Persists across function calls.

```
int b = staticLocal();
int c = staticLocal();
int d = staticLocal();
```

1
2
3

Static Member(s)

Static and Classes

`static` Member Variables:

- *All* Class Objects share *the same* (one-and-only copy of) data .
  If one Object modifies it, all Objects will see the change.

Not "bound" to a specific Object, but mark a state of the Class itself.

Syntax:

```
class <class_name> { …
  static <type_id> static_classVarName;
}
```

Useful for "tracking", i.e. :

- How often a Member Function is called.
- How many Objects exist at given time.

Static Member(s)

Static and Classes

`static` Member Functions:

- No access to specific Object data is needed (still is member of the Class Namespace).

Bound to Class itself, can only use `static` Member Data, `static` Member Functions.

Syntax:

```
class <class_name> { …
  static <ret_type_id> static_classFunctionName( <params_list> );
}
```

Can be called outside of Class.

- With Class-calls: `<class_name>::static_classFunctionName(…);`
  Example: `Server::getTurn();`
- Via Class-Objects: `<classObject_name>.static_classFunctionName(…);`
  Example: `myServer.getTurn();`

Static Member(s)

Static and Classes - Example

```
1   #include <iostream>
2   using namespace std;

3   class Server
4   {
5   public:
6       Server(char letterName);
7       static int getTurn( );
8       void serveOne( );
9       static bool stillOpen( );
10  private:
11      static int turn;
12      static int lastServed;
13      static bool nowOpen;
14      char name;
15  };

16  int Server:: turn = 0;
17  int Server:: lastServed = 0;
18  bool Server::nowOpen = true;
```

```
39  Server::Server(char letterName) : name(letterName)
40  {/*Intentionally empty*/}

41  int Server::getTurn( )
42  {
43      turn++;
44      return turn;
45  }
46  bool Server::stillOpen( )
47  {
48      return nowOpen;
49  }

50  void Server::serveOne( )
51  {
52      if (nowOpen && lastServed < turn)
53      {
54          lastServed++;
55          cout << "Server " << name
56               << " now serving " << lastServed << endl;
57      }
58      if (lastServed >= turn) //Everyone served
59          nowOpen = false;
60  }
```

Since `getTurn` *is static, only static members can be referenced in here.*

Object-Method that accesses & modifies Class `static` Members

Static and Classes - Example

```
19  int main( )
20  {
21      Server s1('A'), s2('B');
22      int number, count;
23      do
24      {
25          cout << "How many in your group? ";
26          cin >> number;
27          cout << "Your turns are: ";
28          for (count = 0; count < number; count++)
29              cout << Server::getTurn( ) << ' ';
30          cout << endl;
31          s1.serveOne( );
32          s2.serveOne( );
33      } while (Server::stillOpen( ));

34      cout << "Now closing service.\n";

35      return 0;
36  }
```

SAMPLE DIALOGUE

```
How many in your group? 3
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? 2
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? 0
Your turns are:
Server A now serving 5
Now closing service.
```

Separate Objects, but their behavior interfaces, due to the unique status of entire Class.

- `static` Class Member(s)

Operators in Classes – Introduction

Remember Aggregate Class Initialization:

```
class Vacation{
  public:
    Vacation(int numDays, const Date& firstDay);
  private:
    int m_tripLength;
    Date m_startDay;
};

Vacation::Vacation(int numDays, const Date& firstDay){
  m_tripLength = numDays;
  m_startDay = firstDay;
}
```

```
class Date{
 public:
 Date();
 Date(int month,
  int day=DFT_D,
  int year=DFT_Y,
  bool gregorian=true);
 Date(const Date &date);

 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month, m_day,
     m_year;
 const bool m_gregorian;
};
```

Operators in Classes – Introduction

Remember Aggregate Class Initialization:

```
class Vacation{
  public:
    Vacation(int numDays, const Date& firstDay);
  private:
    int m_tripLength;
    Date m_startDay;
};

Vacation::Vacation(int numDays, const Date& firstDay){
  m_tripLength = numDays;
  m_startDay = firstDay;
}
```

What would be the "meaning" of this (=) among **Date**s ?

Compiler creates a default *Assignment* Operator (=) for Class Objects: a ***Member**-Copy*.

```
class Date{
 public:
 Date();
 Date(int month,
  int day=DFT_D,
  int year=DFT_Y,
  bool gregorian=true);
 Date(const Date &date);

 void SetM/D/Y(int mdy);
 int GetM/D/Y() const;
 void ShiftNextDay();
 private:
 int m_month, m_day,
     m_year;
 const bool m_gregorian;
};
```

Operator(s)

Operators (`+`, `-`, `%`, `==`, etc.) **and Built-in Types** (`int`, `double`, etc.)

In reality they represent Functions.

- Simply "called" with different syntax:

```
x + 7;
```

(`+`) is binary operator with x and 7 as operands.

- It's just a more intuitive notation for humans, instead of:

```
+(x, 7);
```

Function Arguments

Function Name

Operator(s) and Custom Types

Useful to have an Operator work with user-defined types?

- Operator (**+**) :

```
classObject3 = classObject1 + classObject2;
```

```
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;
};
```

Operator(s) and Custom Types

Useful to have an Operator work with user-defined types?

- Operator (**+**) :

```
classObject3 = classObject1 + classObject2;
```

Meaningful to apply it on a user-defined type?

- `myMoney = myMoney + salaryMoney;` Makes sense?
- `someDate = startDate + endDate;`

```
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;
};
```

Operator(s) and Custom Types

Useful to have an Operator work with user-defined types?

- Operator (**+**) :

```
classObject3 = classObject1 + classObject2;
```

Meaningful to apply it on a user-defined type?

- `myMoney = myMoney + salaryMoney;` Makes sense?
- `someDate = startDate + endDate;` Makes sense?

```
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;
};
```

Operator(s) and Custom Types

Useful to have an Operator work with user-defined types?

- Operator (+) :

```
classObject3 = classObject1 + classObject2;
```

Meaningful to apply it on a user-defined type?

- `myMoney = myMoney + salaryMoney;` Makes sense?
- `someDate = startDate + endDate;` Makes sense?

Particular challenges to keep it meaningful?

- `myMoney = myMoney + salaryMoney;`

```
${1000,125} =  ${0,75}  +  ${1000,50}
```

```
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;
};
```

Operator(s)

Remember All Operators ?

Overload just about anything, but be VERY careful…

- `[ ]`
- `*` : Multiplication, Pointer Dereference
- `/` : Division
- `+` : Addition, Unary Positive
- `-` : Subtraction, Unary Negative
- `++` : Increment, Pre-and-Post
- `--` : Decrement, Pre-and-Post
- `=` : Assignment
- `<=`, `>=`, `<`, `>`, `==`, `!=` : Comparisons
- Many, many others…

Remember All Operators ?

Some are out, some should be kept untouched…

- `?` : Ternary Conditional is not Overloadeable.

- `&&`, `||`, built-in versions are defined for `bool` types.
  Use "Short-Circuit Evaluation", also available in C++.
- When overloaded no longer uses "Short-Circuit", but "Complete Evaluation".
  Generally should not overload these operators,
  (also Operator Overloading had better "make sense").

CS-202

Time for Questions !