

CS-202

C++ Classes – Operator(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 4th Project Deadline is *next* Wednesday 10/4!

- 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), Initialization List(s), Destructor
- `static` Members – Variables / Functions
- Operator Overloading

Class `friend`(s)

Keyword `this`

Operator Overloading (continued)

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];  
};
```

Classes

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;
```

Object

```
Car* myCar_Pt;
```

Pointer to Object

```
myCar_Pt = &myCar;
```

```
(*myCar_Pt).GetMileage();
```

- 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 (`.` , `->`)

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

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

```
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];
};
```

Classes

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 !

Classes

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 !

Classes

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 !

Classes

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.

```
class Hybrid : Car { A Derived Class  
    ...  
    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 (tin[i]<... || tin[i]>...) { return false; }
    }
    for (int i=0; i<16; ++i) {
        m_engineTiming[i]=tin[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];
};
```


Classes

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 you have to define a *Default* (empty) constructor.

```
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];  
};
```

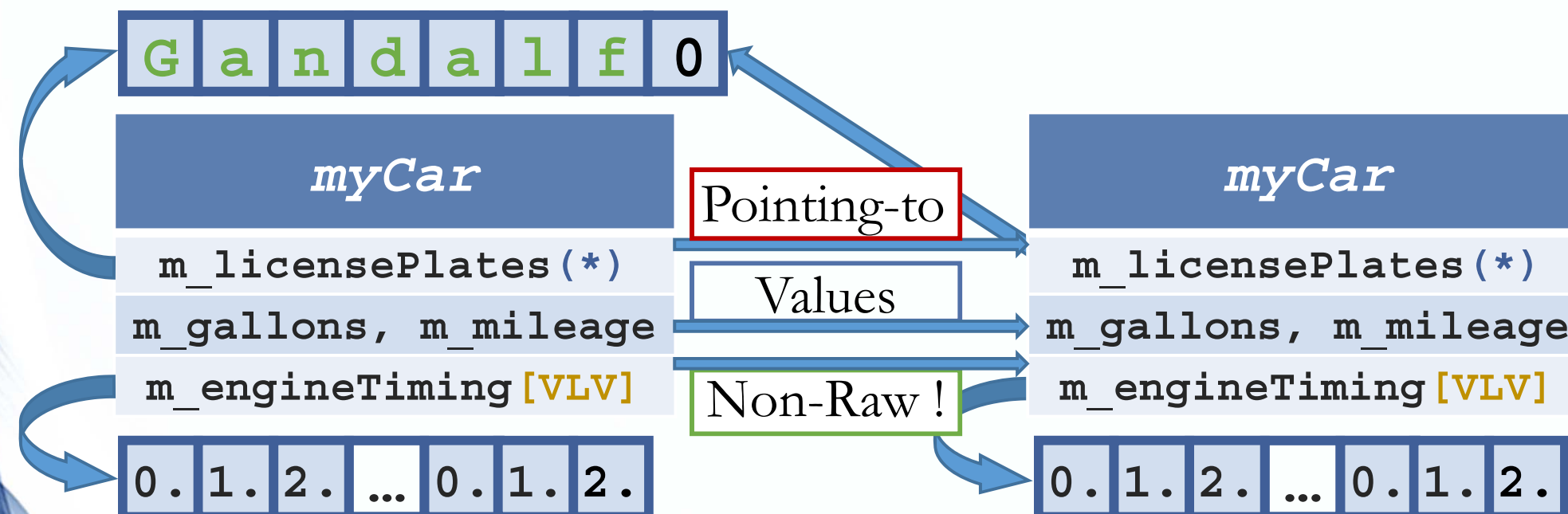

Class Cheatsheet

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**:

➤ Explicitly 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] ;  
};
```

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, 19 ); //forever 19
```

```
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

Initialization List(s):

➤ *Aggregate Class 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 licPlts[PLT], int fId) :  
    Car(licPlts, DFT_GLNS, 0, fId, DFT_TIM);
```

Default Member Initialization (C++11):

- Can set default Member values in Declaration.
- Any *Initialization List* appearance of the member will have 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 licPlts[PLT], int fId) :  
            Car(licPlts, 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() { carFactoryCnt++; } //dflt ctor  
cout << Car::carFactoryCnt; //via class  
Car myCar1; //call dflt ctor, increment cnt  
cout << myCar1.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:  
    ...  
    static int carFactoryCnt;  
};
```

```
#include <Car.h> //Class Source  
int Car::carFactoryCnt = 0; //init  
int Car::GetCarFactoryCnt() {  
    return Car::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);
```

```
Output: Refilled 1 times
```

```
myCar2.AddG(10.0);
```

```
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 use to 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 SetEngineTiming(double [VLV]);  
    double m_engineTiming[VLV];  
    const int m_frameId;  
    static int 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);
```

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

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

Evaluates to: *Unnamed* Object

OK...

No !!!

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

```
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;  
};
```


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!

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

```
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;
};
```


Class Cheatsheet

Operator Overloading

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

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

```
int& Money::operator[] (const 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[] (const
        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 Friend(s)

Friend Functions

Remember: Operator Overloads as no-Member function:

- Access of data through *Accessor* and *Mutator* functions.
- Very inefficient (call overhead).

Class **friend**(s) can directly access **private** Class data.

- Any function can be a Class **friend**.
- Make non-Member Operator Overloads **friend**(s) (no overhead, more efficient).

Operator Overloads as non-Member Class **friend**(s).

- Most common use (avoids need to go through Setter / Getter functions interface).
- Need data access anyway.

Class Friend(s)

Friend Functions

A **friend** Function of a Class is:

- Not a Member Function, but still has direct access to **private** members.
- Specified in Class Declaration (keyword **friend**) but still isn't a Member Function.

Friends and *Purity*:

- “Spirit” of OOP dictates all Operators and Functions must be Member Functions.
(many believe **friend**(s) violate basic OOP principles.

However: Very advantageous for Operators:

- Allow automatic type conversion.
- Encapsulation is retained – **friend** is in Class Declaration.
- Improves efficiency.

Class Friend(s)

Friend Classes

A **friend** Class of another Class:

- Has direct access to **private** members.
- Is specified in Class Declaration (keyword **friend**).

Example: **class F** is **friend** of **class C**

- All **class F** Member Functions are **friends** of **class C**.
- Not reciprocated relationship, **friendship** granted, not taken!

```
class C {  
    friend bool operator==(const C& c1, const C& c2); //A friend regular function  
    friend class F; //A friend class (all its member functions are friends)  
    ...  
}
```

Functions have direct access to any private Members
(Data and Functions of **class C**)

Cascading

Return by-Reference – Cascading

Remember: Overloading Operator (`[]`)

Get `<type_id>&`, internal Member Reference:

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

```
class Money{  
    int& operator[] (const int i);  
    int m_transID[T_HIST];  
};
```

Another utility for Operator Overloading:

➤ Cascading (daisy-chaining):

```
double& chainableFun(double& var)  
{ return var; }
```

```
double x;
```

```
chainableFun(chainableFun(...(chainableFun(x))...));
```

Note: Cannot do `return var+1.0;`

`double&+double` has no Reference!

error: invalid initialization of non-const reference of type 'double&' from an rvalue of type 'double'

Operator Overloading

Return by-Reference – Cascading

Overloading Operator (<<):

- Insertion (*Binary*) Operator.
- Used with `cout` Object (from `<iostream>` library).

Example:

`cout << "Hello world!";`
1st Operand: `class ostream` Object 2nd Operand: C-string Literal

Overload for a `class Money` type 2nd operand:

```
Money hugeCheck(1000000,0);  
cout << hugeCheck;
```

Instead of:

```
hugeCheck.Output();
```

```
class Money{  
    public:  
    Money();  
    Money(int d, int c=0);  
    Money(const Money &m);  
    friend ostream&  
    operator<<(ostream& os  
    , const Money& m);  
    void Output();  
    void SetD/C(int dc);  
    int GetD/C() const;  
    private:  
    int m_dollars,m_cents;  
};
```

Operator Overloading

Return by-Reference – Cascading

Overloading Operator (<<):

- Insertion (*Binary*) Operator.
- Used with **cout** Object (from <iostream> library).

Cascading how-to: Return by-1st-operand-Reference.

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

```
Money myMoney(99,25), yourMoney(0,75);  
cout << "Mine:" << myMoney << " Yours:" << yourMoney;
```

Like calling:

```
operator<<(operator<<(operator<<(operator<<(cout  
, "Mine:") , myMoney) , " Yours:") , yourMoney);
```

```
class Money{  
    public:  
    Money();  
    Money(int d, int c=0);  
    Money(const Money &m);  
    friend ostream&  
    operator<<(ostream& os  
    , const Money& m);  
    void Output();  
    void SetD/C(int dc);  
    int GetD/C() const;  
    private:  
    int m_dollars, m_cents;  
};
```


Operator Overloading

Return by-Reference – Cascading

Overloading Operator (>>):

- Extraction (*Binary*) Operator.
- Used with *cin* Object (from <iostream> library).

Overloading and Cascading (return by-1st-operand-Ref).

```
istream& operator>>(istream& is, Money& mn) {  
    char dollarChar;  
    is >> dollarChar;  
    if (dollarChar=='$') {  
        double dollarsDouble;  
        is >> dollarsDouble;  
        mn.m_dollars = dollarsDouble;  
        mn.m_cents = 100*(dollarsDouble-mn.m_dollars);  
    }  
    return is;  
}
```

```
class Money{  
    public:  
    Money();  
    Money(int d, int c=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;  
};
```


Operator Overloading

Overloading Operators (<<), (>>)

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4  using namespace std;

5  //Class for amounts of money in U.S. currency
6  class Money
7  {
8  public:
9      Money( );
10     Money(double amount);
11     Money(int theDollars, int theCents);
12     Money(int theDollars);
13     double getAmount( ) const;
14     int getDollars( ) const;
15     int getCents( ) const;
16     friend const Money operator +(const Money& amount1, const Money& amount2)
17     friend const Money operator -(const Money& amount1, const Money& amount2)
18     friend bool operator ==(const Money& amount1, const Money& amount2);
19     friend const Money operator -(const Money& amount);
20     friend ostream& operator <<(ostream& outputStream, const Money& amount);
21     friend istream& operator >>(istream& inputStream, Money& amount);
22 private:
23     int dollars; //A negative amount is represented as negative dollars and
24     int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
25     int dollarsPart(double amount) const;
26     int centsPart(double amount) const;
27     int round(double number) const;
28 };
```

```
class Money{
public:
    Money();
    Money(int d, int c=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;
};
```

Operator Overloading

Overloading Operators (<<), (>>)

```
49 ostream& operator <<(ostream& outputStream, const Money& amount)
50 {
51     int absDollars = abs(amount.dollars);
52     int absCents = abs(amount.cents);
53     if (amount.dollars < 0 || amount.cents < 0)
54         //accounts for dollars == 0 or cents == 0
55         outputStream << "$-";
56     else
57         outputStream << '$';
58     outputStream << absDollars;
59     if (absCents >= 10)
60         outputStream << '.' << absCents;
61     else
62         outputStream << '.' << '0' << absCents;
63     return outputStream;
64 }
65
66 //Uses iostream and cstdlib:
67 istream& operator >>(istream& inputStream, Money& amount)
68 {
69     char dollarSign;
70     inputStream >> dollarSign; //hopefully
71     if (dollarSign != '$')
72     {
73         cout << "No dollar sign in Money input.\n";
74         exit(1);
75     }
76     double amountAsDouble;
77     inputStream >> amountAsDouble;
78     amount.dollars = amount.dollarsPart(amountAsDouble);
79     amount.cents = amount.centsPart(amountAsDouble);
80     return inputStream;
81 }
```

In the main function, cout is plugged in for outputStream.

For an alternate input algorithm, see Self-Test Exercise 3 in Chapter 7.

Returns a reference

In the main function, cin is plugged in for inputStream.

Since this is not a member operator, you need to specify a calling object for member functions of Money.

Returns a reference

```
class Money{
public:
    Money();
    Money(int d, int c=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;
};
```


Operator Overloading

Overloading Operators (<<), (>>)

```
29 int main( )
30 {
31     Money yourAmount, myAmount(10, 9);
32     cout << "Enter an amount of money: ";
33     cin >> yourAmount;
34     cout << "Your amount is " << yourAmount << endl;
35     cout << "My amount is " << myAmount << endl;
36
37     if (yourAmount == myAmount)
38         cout << "We have the same amounts.\n";
39     else
40         cout << "One of us is richer.\n";
41
42     Money ourAmount = yourAmount + myAmount;
43     cout << yourAmount << " + " << myAmount
44         << " equals " << ourAmount << endl;
45
46     Money diffAmount = yourAmount - myAmount;
47     cout << yourAmount << " - " << myAmount
48         << " equals " << diffAmount << endl;
49
50     return 0;
51 }
```

Since << returns a reference, you can chain << like this.
You can chain >> in a similar way.

SAMPLE DIALOGUE

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36
```

```
class Money{
public:
    Money();
    Money(int d, int c=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;
};
```

Operator Overloading

Operators (**++**), (**--**) (half the story, the rest for later)

Overloading Pre-Increment Operator(s):

➤ No arguments (for compiler *disambiguation*).

```
Money Money::operator++() {  
    m_cents++;  
    if (m_cents...) { m_dollars=...; m_cents=...; } //and fix  
    return Money(m_dollars,m_cents);  
}
```

Note:

Modifies calling Object and **returns** a Copy of it.

```
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;  
};
```


Operator Overloading

Operators (**++**), (**--**) (half the story, the rest for later)

Overloading Post-Increment Operator(s):

➤ A dummy **int** argument (for compiler *disambiguation*).

```
Money Money::operator++(int dummy) {  
    Money moneyCopy(m_dollars, m_cents);  
    m_cents++;  
    if (m_cents...) { m_dollars=...; m_cents=...; } //fix  
    return moneyCopy;  
}
```

Note: Keeps a Copy of calling Object to **return** and then modifies calling Object.

```
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**

A Pointer to the Calling Object.

- Inside a Class Member Function, we can address Member Variables of the Calling Object by name.
- Keyword **this** provides a way to address the entire Calling Object inside a Member Function call.

```
Money& Money::thisFunction() {  
    this -> m_dollars = 1000; //access data  
    this -> SetC(99);        //call function  
    return *this;  
}
```

Note: A Member Function can return a Reference to the Calling Object that invoked it.

```
class Money{  
    public:  
    Money();  
    Money(int d, int c=0);  
    Money(const Money &m);  
    Money& thisFunction();  
  
    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**

Overloading Pre-Increment Operator(s) (now for the rest) :

- No arguments (for compiler *disambiguation*).

```
Money& Money::operator++ () {
    m_cents++; ... //changes to calling object
    return *this;
}
```

Note:

Modifies calling Object and **returns** 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;
};
```


Keyword **this**

Overloading Post-Increment Operator(s) (now for the rest) :

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

```
Money Money::operator++(int dummy) {
    Money moneyCopy(*this);
    this->m_cents++; ... //changes to 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**

Overloading Assignment Operator (=) (now for the rest) :

- **return** Reference to Calling Object, maintain Assignment Operator sequencing:

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

- Maintain Assignment Operator sequencing:

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

```
a = b = c;
```

Can now do this too!

```
class Money{
public:
    Money();
    Money(int d, int c=0);
    Money(const Money &m);
    Money& operator=(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;
}
```

➤ Protect from (unwillingly) tampering with own self:

```
Money cashiers[100];
Money* active_desk = cashiers;
active_desk += rand_desk_offset;
```

```
for (int i=0; i<100; ++i)
    active_desk->Accrue(cashiers[i]);
```

Sums contents of all onto itself, but should avoid adding its own data twice (skip if Calling Object active_desk is the same as cashiers[i])

```
class Money{
public:
    Money();
    Money(int d, int c=0);
    Money(const Money &m);

    bool thisCheck(const
                    Money& m);

    void Accrue(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;
};
```

CS-202

Time for Questions !