CS-202

C++ Classes – Operator(s) (Pt.1)

C. Papachristos

Autonomous Robots Lab

University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (4 Sections)	
	CLASS	RL – Session	CLASS	
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	

Your 4th Project will be announced today Thursday 2/15.

3rd Project Deadline was this Wednesday 2/14.

- ➢ NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- ➢ Send what you have in time!
- ➢ Check out **WebCampus** CS-202 Announcements for some **help** !

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

Classes

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.
- ➢ `camelCase` for phrases.
- ➢ General word for Class of Objects.

Class Cheatsheet

Declaration:

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

Access Specifiers

- Provide Protection Mechanism

Encapsulation - Abstraction:

- "Data Hiding"

Classes

Class Cheatsheet

Declaration:

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

Member Variables

- ➢ All necessary Data inside a single Code Unit.

Conventions:

- ➢ Begin with **m_<variable_name>**.

Encapsulation - Abstraction:

- ➢ Abstract Data Structure

Class Cheatsheet

Declaration:

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

Member Function / Class Methods

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

Conventions:

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

Encapsulation - Abstraction:

- Abstract Data Structure

Classes

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.

Classes

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];
};
```

Classes

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

(->)

- Structure (Class) Pointer Dereference

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

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

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

Class Cheatsheet

Class Object Pointer Usage:

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

Arrow Operator – Member-access (->)

- Structure (Class) Pointer Dereference

Why?

Chaining Operator Precedence (. , ->)

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

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

Class Cheatsheet

Class Object in Function – By-Value:

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

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

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

Note:
Will work with Local Object Copy !

Class Cheatsheet

Class Object in Function – By-Reference:

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

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

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

Note:

Will modify Object Data !

Class Cheatsheet

Class Object in Function – By-const-Reference:

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

void printCapPlates(const Car& car){
  char* lP = (char*)malloc(sizeof(
                 car.m_licensePlates));
  strcpy(lP,car.m_licensePlates);

  char* lP_0 = lP;
  while (*lP = toupper(*lP)){ ++lP; }

  cout << lP_0 << endl;
}
```

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

Note:
Not allowed to modify Object Data !

Class Cheatsheet

Class Object in Function – By-Address:

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

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

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

Note:

Will modify Object Data !

Class Cheatsheet

Protection Mechanisms – const Method signature:

A “promise” that Method doesn’t modify Object

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

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

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

Classes

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];
};
```

Classes

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];
};
```

Classes

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  {
   …
   float gasToElectricRatio();
};
```

A *Derived* Class

```
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];
};
```

Classes

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];
};
```

Classes

Class Cheatsheet

Classes and Code File Structure

Class Header File: `Car.h`

```
#ifndef CAR_H
#define CAR_H

#define NUMVALVES 16

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

#endif
```

Class Source File: `Car.cpp`

```
#include <iostream>
#include "Car.h"

#define MAX_GALLONS 20.0

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

Class Cheatsheet

Classes and Code File Structure

Note: Compile all your source (.cpp) files together with

```
g++ car_program.cpp Car.cpp
```

Program File: `car_program.cpp`

```
#include <iostream>
#include <…>

#include "Car.h"

int main(){
  Car myCar;
  Car* myCar_Pt = &myCar;

  strcpy(myCar_Pt->m_licensePlates,"Gandalf");
  printCapPlates(myCar_Pt);
  cout << myCar.m_licensePlates << endl;

  cout << myCar.getMileage() << endl;
  cout << myCar.addGas(10.0F) << endl;
  return 0;
}
```

Classes

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];
};
```

Classes

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];
};
```

Classes

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 !

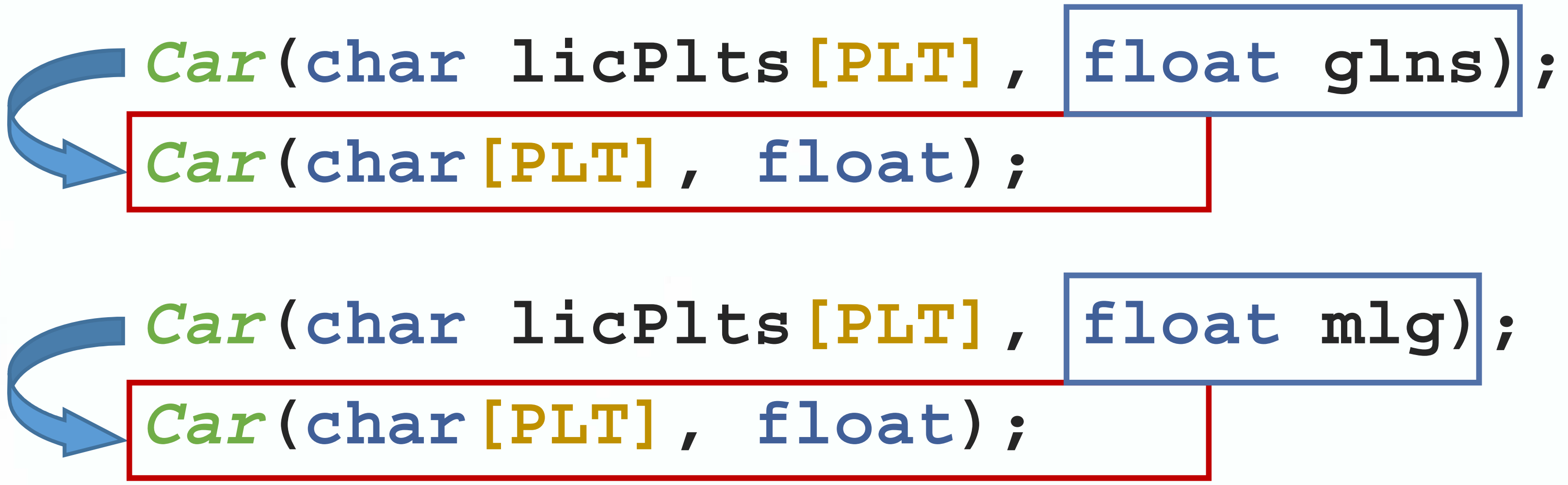
```
class Car  {
 public:
  Car();
  Car(char licPlts[PLT],
  float glns=DFT_GLNS, float mlg=0,
  const double engTim[VLV]=DFT_TIM);
  Car(const Car & car);
  float addGas(float gallons);
  float getGallons() const ;
  float getMileage() const ;
  char m_licensePlates[PLT];
 protected:
  float m_gallons;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

Classes

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];
};
```

Classes

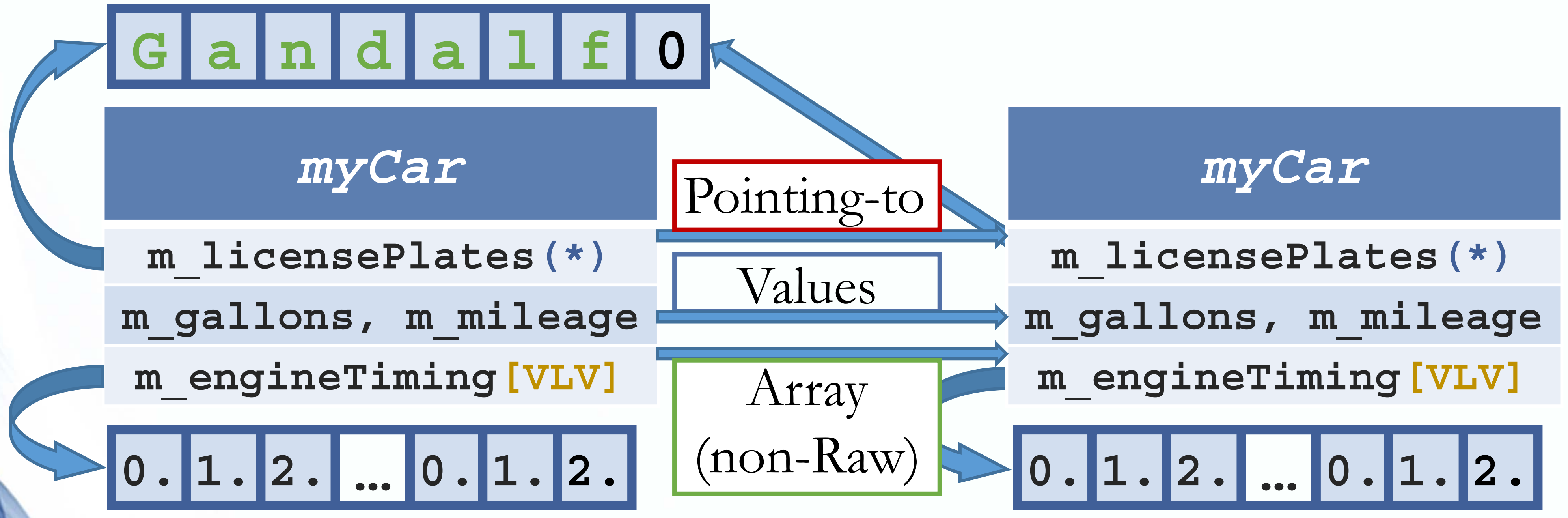
Class Cheatsheet

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);
```



G a n d a l f 0

myCar

m_licensePlates(*)

m_gallons, m_mileage

m_engineTiming[VLV]

0. 1. 2. … 0. 1. 2.

Pointing-to

Values

Array (non-Raw)

myCar

m_licensePlates(*)

m_gallons, m_mileage

m_engineTiming[VLV]

0. 1. 2. … 0. 1. 2.

```
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];
};
```

Classes

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];
};
```

Classes

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",0,0,11000); //11000 years
```

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

Classes

Class Cheatsheet

Initialization List(s):

- Class-with-*Composistion* Initialization.

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

ctor-in-ctor Call

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

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;
};
```

Classes

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;
};
```

Classes

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:
  // declaration of static member
  static int s carFactoryCnt;
};
```

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

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

Classes

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
```

Note: 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 getEngineTiming(double[VLV]);
  double m_engineTiming[VLV];
  const int m_frameId;
  static int s_carFactoryCnt;
};
```

Operator Overloading

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;
};
```

Operator Overloading

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 Overloading

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

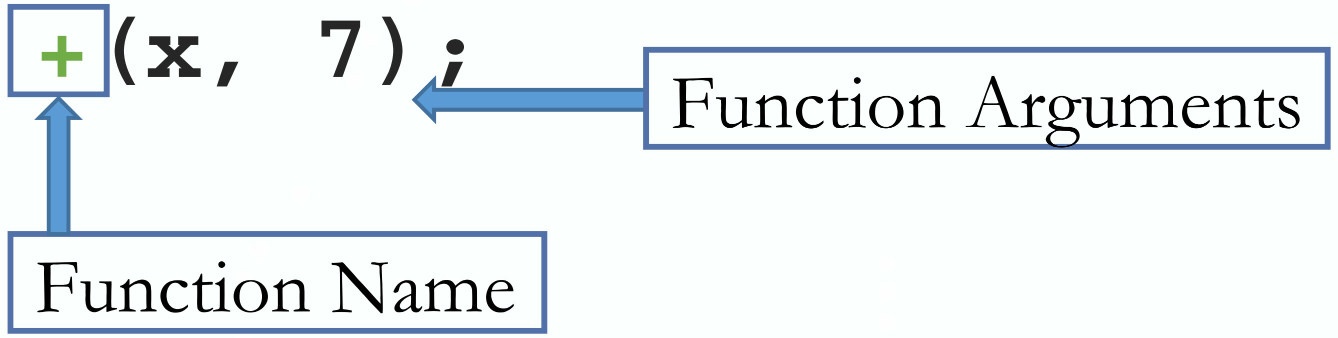
In reality they represent Functions.

- 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 Overloading

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 Overloading

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 Overloading

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 Overloading

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 Overloading

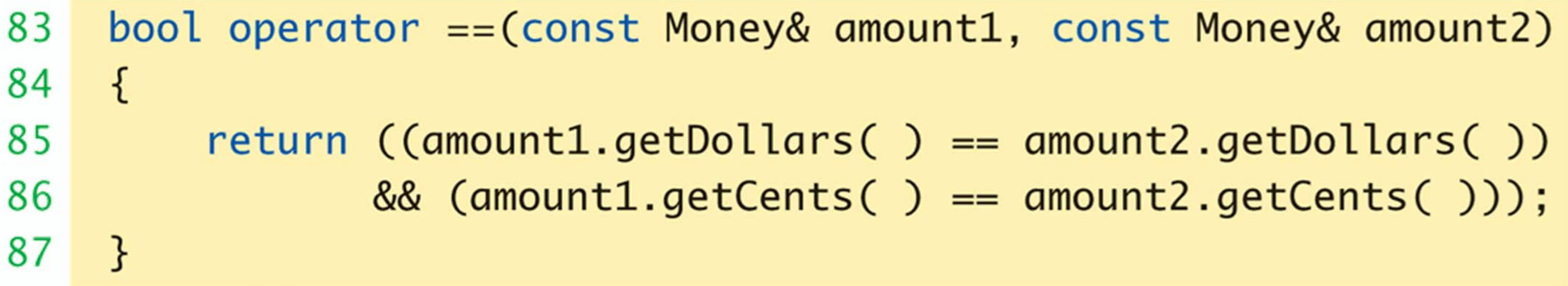
Overloading Operator(s)

Overloading *Binary* Operator (**==**):

- Non-Member Function of Class ***Money***.
- Like overloading functions, Operator is Function name.

Syntax:

```
bool operator ==(const Money& amount1,
                 const Money& amount2);
```



```
bool operator ==(const Money& amount1, const Money& amount2)
{
    return ((amount1.getDollars( ) == amount2.getDollars( ))
           && (amount1.getCents( ) == amount2.getCents( )));
}
```

- "Compares" ***Money*** Objects.

```
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 Overloading

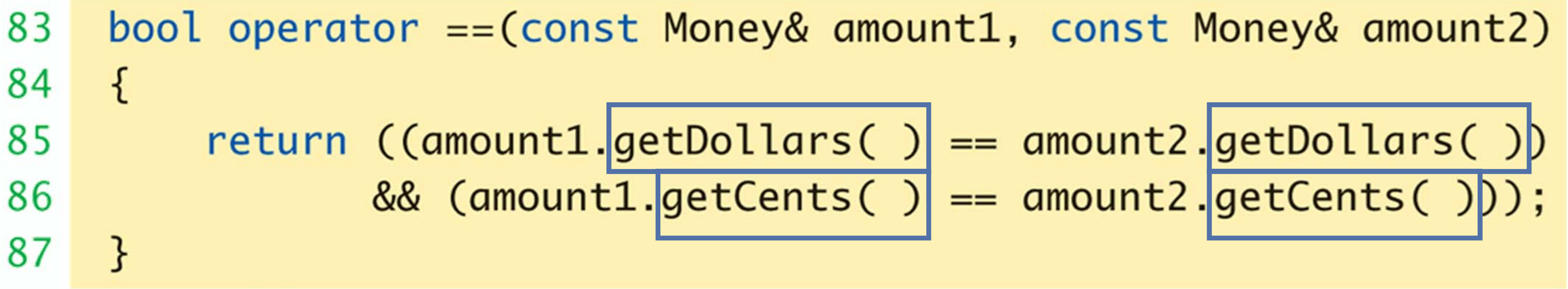
Overloading Operator(s)

Overloading *Binary* Operator (**==**):

- Non-Member Function of Class ***Money***.
- Like overloading functions, Operator is Function name.

Syntax:

```
bool operator ==(const Money& amount1,
                 const Money& amount2);
```



```
bool operator ==(const Money& amount1, const Money& amount2)
{
    return ((amount1.getDollars( ) == amount2.getDollars( ))
            && (amount1.getCents( ) == amount2.getCents( )));
}
```

- "Compares" ***Money*** Objects.

```
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 Overloading

Overloading Operator(s)

Overloading *Unary* Operator (**-**):

- ➢ Non-Member Function of Class ***Money***.
- ➢ Like overloading functions, Operator is Function name.

Syntax:

```
const Money operator -(const Money& amount) {
 return Money(-amount.GetD(),-amount.GetC());
}
```

Example:

```
Money moneyIn(1000, 0);
Money moneyOut = - moneyIn;
```

- ➢ "Negates" a ***Money*** Object.
- ➢ Returns an *Unnamed* 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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading *Unary* Operator (**-**):

- Non-Member Function of Class **Money**.
- Like overloading functions, Operator is Function name.

Syntax:

```
const Money operator -(const Money& amount) {
 return Money(-amount.GetD(),-amount.GetC());
}
```

Example:

```
Money moneyIn(1000, 0);
Money moneyOut = - moneyIn;
```

- "Negates" a **Money** Object.
- Returns an *Unnamed* 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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (+):

- Non-Member Function of Class **Money**.
- Like overloading functions, Operator is Function name.

Syntax:

```
const Money operator + (const Money& amount1,
                        const Money& amount2);
```

"Adds" **Money** Objects:

- Overloads + for operands of type **Money**.
- Uses `const`-Reference Parameters for efficiency.
- Returned value is of type **Money**, *Unnamed* Object.

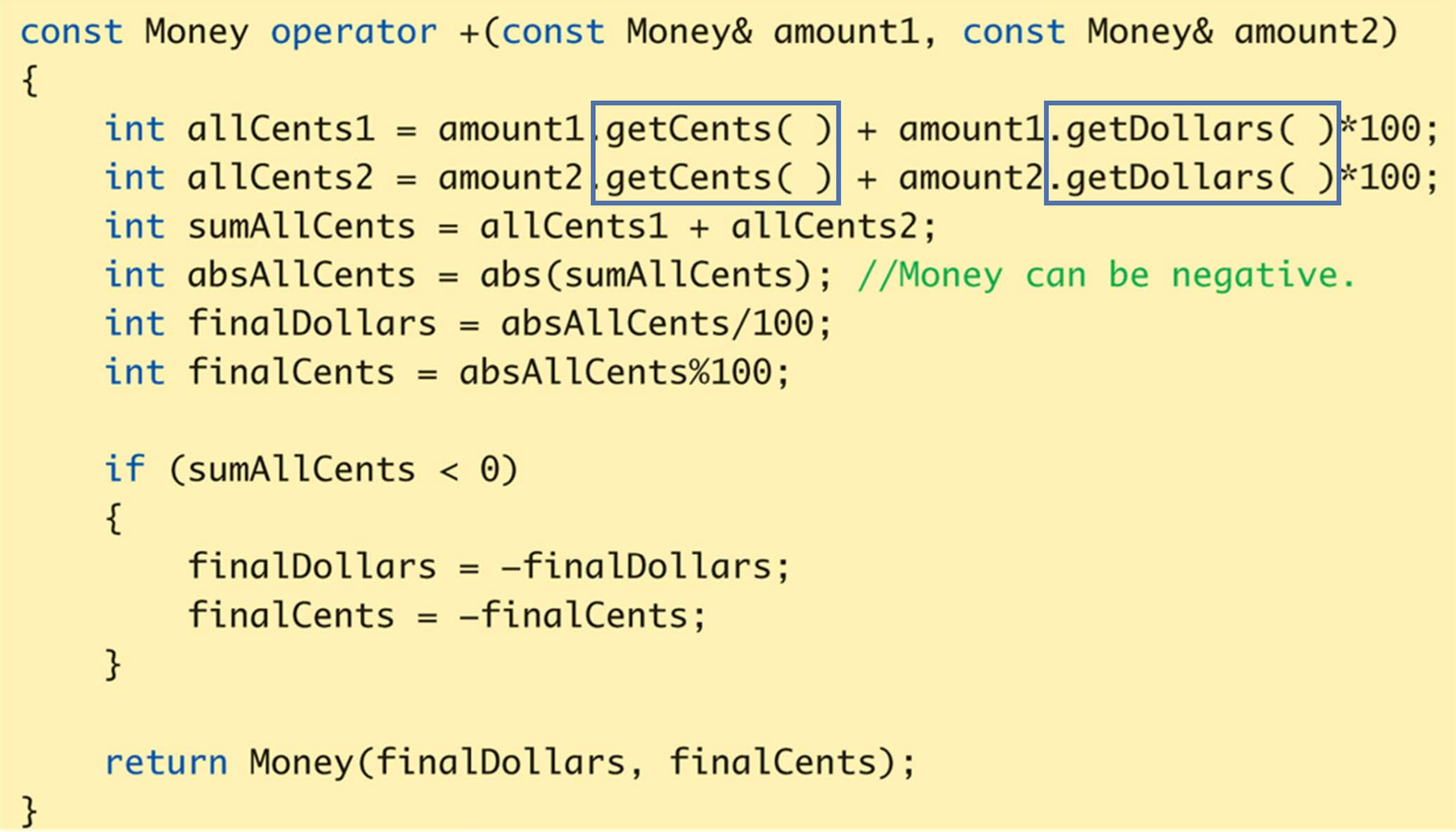
```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(const Money &m);

  void setD/C(int dc);
  int getD/C() const;

 private:
  int m_dollars;
  int m_cents;
};
```

Operator Overloading

Overloading Operator(s)

Still, like a regular *Overloaded* Function:

- Non-Member Function of Class **Money**.
- More "involved" than Member-by-Member adding.



```
const Money operator +(const Money& amount1, const Money& amount2)
{
    int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
    int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
    int sumAllCents = allCents1 + allCents2;
    int absAllCents = abs(sumAllCents); //Money can be negative.
    int finalDollars = absAllCents/100;
    int finalCents = absAllCents%100;

    if (sumAllCents < 0)
    {
        finalDollars = -finalDollars;
        finalCents = -finalCents;
    }

    return Money(finalDollars, finalCents);
}
```

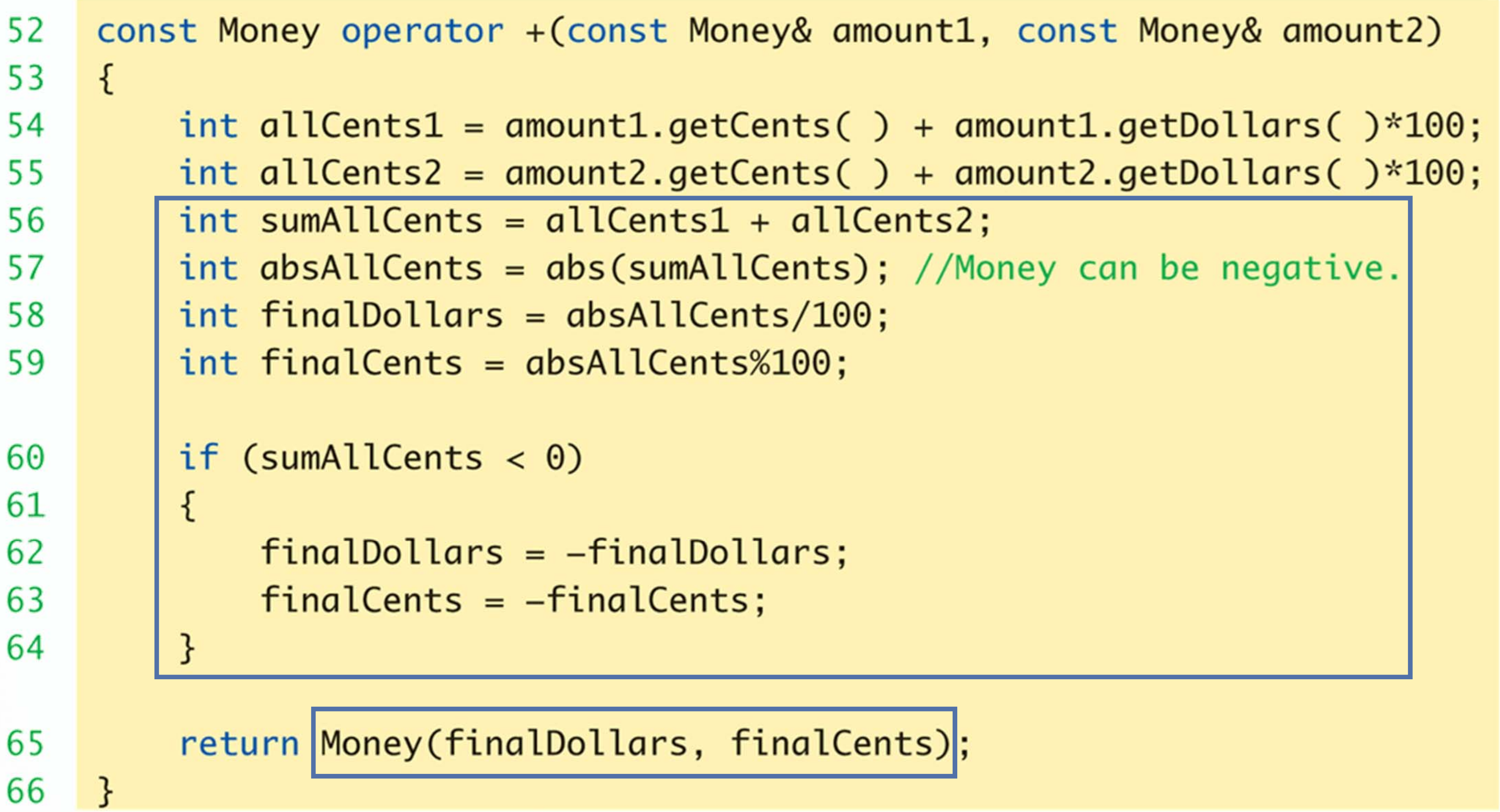
```
class Money{
 public:
  Money();
  Money(int dollars,
            int cents=0);
  Money(const Money &m);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars;
  int m_cents;
};
```

Operator Overloading

Overloading Operator(s)

Still, like a regular *Overloaded* Function:

- Non-Member Function of Class **Money**.
- More "involved" than Member-by-Member adding.



```
const Money operator +(const Money& amount1, const Money& amount2)
{
    int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
    int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
    int sumAllCents = allCents1 + allCents2;
    int absAllCents = abs(sumAllCents); //Money can be negative.
    int finalDollars = absAllCents/100;
    int finalCents = absAllCents%100;

    if (sumAllCents < 0)
    {
        finalDollars = -finalDollars;
        finalCents = -finalCents;
    }

    return Money(finalDollars, finalCents);
}
```

```
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 Overloading

Overloading Operator(s)

Overloading Operator (+):

- ➢ A Member Function of Class ***Money***.

Syntax (Function Prototype):

```
const Money operator +(const Money& m) const;
```

```
class Money{
 public:
  Money();
  Money(int dollars,
            int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (+):

- A Member Function of Class ***Money***.
- Calling Object serves as 1st parameter.

Syntax (Function Prototype):

```
const Money operator +(const Money& m) const;
```

Example:

```
Money cost(1, 50), tax(0, 15), total;
total = cost + tax;
```

Intuitively:

```
total = cost .operator+(tax);
```

Calling Object

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (+):

- ➢ A Member Function of Class ***Money***.
- ➢ Calling Object serves as 1st parameter.

Syntax (Function Prototype):

```
const Money operator +(const Money& m) const;
```

Example:

```
Money cost(1, 50), tax(0, 15), total;
total = cost + tax;
```

Intuitively:

```
total = cost .operator+(tax);
```

Calling Object

Operator Member Function

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (+):

- Non-Member Function version.

```
const Money operator+(const Money&a,const Money&b){
  return Money(a.GetD() + b.GetD(),
               a.GetC() + b.GetC() );
}
```

No access to Parameter private Members

- Member Function of Class *Money* version.

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

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (+):

➢ Non-Member Function version.

```
const Money operator+(const Money&a,const Money&b){
  return Money(a.GetD() + b.GetD(),
               a.GetC() + b.GetC() );
}
```

➢ Member Function of Class *Money* version.

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

Calling Object's Members

Class Method (access to Parameter **private** Members)

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (+) , Twice:

- Non-Member Function version.

```
const Money operator+(const Money&a,const Money&b)
{  return Money(1);   }
```

- Member Function of Class *Money* version.

```
const Money Money::operator+(const Money&b) const
{  return Money(2);   }
```

```
warning: ISO C++ says that these are ambiguous, even
though the worst conversion for the first is better
than the worst conversion for the second.
```

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (+) , Twice:

- ➢ Non-Member Function version.

```
const Money operator+(const Money&a,const Money&b)
{  return Money(1);  }
```

- ➢ Member Function of Class **Money** version.

```
const Money Money::operator+(const Money&b) const
{  return Money(2);  }
```

```
warning: ISO C++ says that these are ambiguous, even
though the worst conversion for the first is better
than the worst conversion for the second.
```

```
Money m1,m2, m3 = m1 + m2;
```

Result: 1

```
Money m1,m2, m3 = m1 .operator+ ( m2 );
```

Result: 2

```
class Money{
 public:
  Money();
  Money(int dollars,
          int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (**-**) , Twice (w/ intention):

- Non-Member Function: *Unary*.

```
const Money operator-(const Money &amount){
  return Money(-amount.GetD() , -amount.GetC());
}
```

- Member Function of Class: *Binary*.

```
const Money Money::operator-(const Money&b) const{
  Money tmpMoney(m_dollars - b.m_dollars,
                 m_cents   - b.m_cents );
  /* create temporary object and work with it
    as we go, code to try and fix rollover. */
  return tmpMoney;
}
```

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (**-**) , Twice (w/ intention):

- ➢ Non-Member Function: *Unary.*

```
const Money operator-(const Money &amount){
  return Money(-amount.GetD() , -amount.GetC());
}
```

- ➢ Member Function of Class: *Binary.*

```
const Money Money::operator-(const Money&b) const{
  Money tmpMoney(m_dollars - b.m_dollars,
                 m_cents   - b.m_cents );
  /* create temporary object and work with it
    as we go, code to try and fix rollover. */
  return tmpMoney;
}
```

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (**-**) , Twice (w/ intention):

- ➢ Non-Member Function: *Unary.*

```
const Money operator-(const Money &amount);
```

- ➢ Member Function of Class: *Binary.*

```
const Money Money::operator-(const Money&b) const;
```

Note:
Cannot change Operator Precedence & Associativity rules.

Example calls:

```
Money myPocket(10), myDebts(6,25);
Money myLiving = myPocket - myDebts;      Binary
       {3,75}
Money notMyDebts = - myDebts;             Unary
       {-6,-25}
```

```
class Money{
 public:
  Money();
  Money(int dollars,
          int cents=0);
  Money(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;
};
```

Operator Overloading

Overloading Operator(s)

Overloading Operator (**=**) (half the story, the rest for later) :

- Must be Member Operator.
- If not specified, defaults to Member-Copy Assignment.
- *Remember **Deep**-Copy* vs ***Shallow**-Copy*.

```
void Money::operator=(const Money & amount){
  m_dollars = amount.dollars;
  m_cents = amount.m_cents;

  strcpy(m_owner, amount.m_owner);
}
```

Value-copy

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

Operator Overloading

Overloading Operator(s)

Overloading Operator (**=**) (half the story, the rest for later) :

- Must be Member Operator.
- If not specified, defaults to Member-Copy Assignment.
- *Remember **Deep**-Copy* vs ***Shallow**-Copy*.

```
void Money::operator=(const Money & amount){
  m_dollars = amount.dollars;
  m_cents = amount.m_cents;
  strcpy(m_owner, amount.m_owner);
}
```

Value-copy

User-guaranteed Data-copy on raw Pointers

Note: Class `ctor` needs to have properly allocated memory for the raw Pointer Data.

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

Operator Overloading

Return by-const-Value

Overloading Operator (+) , again:

- Returned: type ***Money***, *Unnamed* Object.

```
const Money operator+(const Money&a,const Money&b){
  return Money(a.getD() + b.getD(),
                 a.getC() + b.getC() );
}
```

Why const-Value ?

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

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

Evaluates to: *Unnamed* Object

```
class Money{
 public:
  Money();
  Money(int dollars,
          int cents=0);
  Money(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;
};
```

Operator Overloading

Return by-const-Value

Overloading Operator (+) , again:

- Returned: type Money, *Unnamed* Object.

```
const Money operator+(const Money&a,const Money&b){
  return Money(a.getD() + b.getD(),
               a.getC() + b.getC() );
}
```

Why const-Value ?

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

`(a + b);` Evaluates to: *Unnamed* Object

`c = (a + b);` OK…

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Return by-const-Value

Overloading Operator (+) , again:

- ➢ Returned: type ***Money***, *Unnamed* Object.

```
const Money operator+(const Money&a,const Money&b){
  return Money(a.getD() + b.getD(),
               a.getC() + b.getC() );
}
```

Why const-Value ?

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

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

Evaluates to: *Unnamed* Object

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

OK…

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

No !!!

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

```
error: passing ‘const Money’ as ‘this’ argument discards
       qualifiers [-fpermissive]
```

```
class Money{
 public:
  Money();
  Money(int dollars,
        int cents=0);
  Money(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;
};
```

Operator Overloading

Return by-const-Reference (?)

Overloading Operator (+) , again:

- Returned: type **Money&**, *Unnamed* Object Reference.

```
const  Money& operator+(const Money&a,const Money&b)
{  return Money(a.getD() + b.getD(),
                 a.getC() + b.getC() );  }
```

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

- Makes a temporary Object, goes out of scope!

```
class Money{
 public:
  Money();
  Money(int dollars,
            int cents=0);
  Money(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;
};
```

Operator Overloading

Return by-const-Reference (?)

Overloading Operator (+) , again:

- Returned: type Money&, *Unnamed* Object Reference.

```
const Money& operator+(const Money&a,const Money&b)
{  return Money(a.getD() + b.getD(),
                    a.getC() + b.getC() );   }
```

```
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);
  const Money operator-
 (const Money& m) const;
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars;
  int m_cents;
};
```

Operator Overloading

Return by-Reference

Overloading Operator (`[]`):

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

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

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

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

Operator Overloading

Return by-Reference (!)

Overloading 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);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars;
  int m_cents;
  int m_transID[T_HIST];
};
```

Operator Overloading

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…

Operator Overloading

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 !