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		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 4th Project will be announced today Thursday 2/14.

3rd Project Deadline was this Wednesday 2/12.

- ➢ NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- ➢ 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(s)

Operator Overloading

Classes

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];
};
```

Classes

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

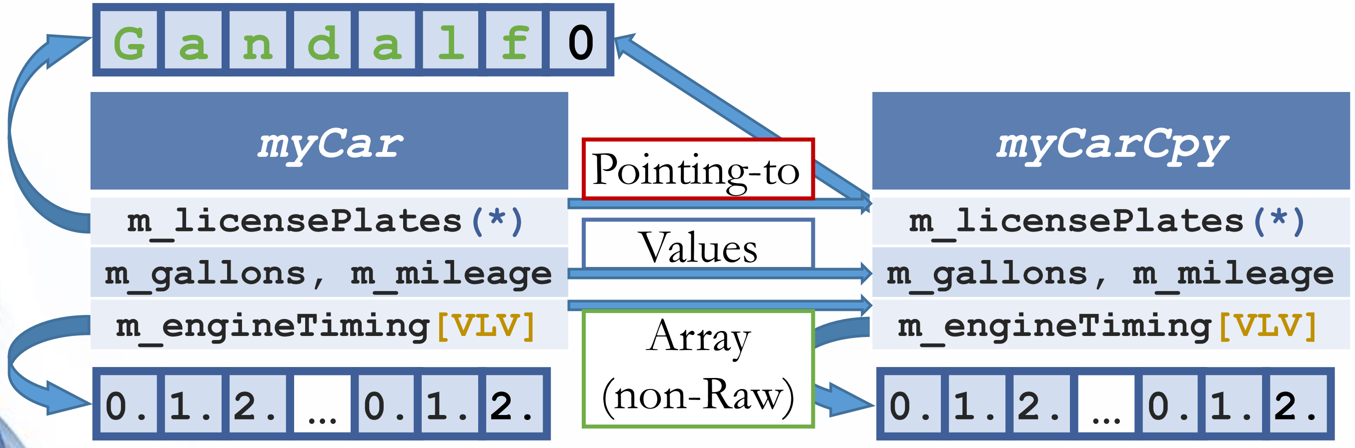
Class Cheatsheet

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);
```



```
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];
};
```

Class Cheatsheet

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

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

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

- ➢ Output: `Gand,Gand`

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

- ➢ Output: `Gand,Gandalf`

Note:

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

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

Classes

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

Initializer List(s):

- ➢ Class-with-*Composistion* 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;
};
```

Classes

Class Cheatsheet

Delegating Constructor (C++11):

- Can have one ctor invoke another ctor.

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

Default Member Initialization (C++11):

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

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

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(){ s_carFactoryCnt++; } //dflt ctor

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

static Member Function:

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

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

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

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

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

Class Cheatsheet

static Local Variables in Class Methods:

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

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

```
Car myCar1, myCar2;
myCar1.addG(10.0);        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 general use to the Class) !

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

Operators in Classes – Introduction

Remember Class-with-*Composistion* 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(s)

Operators in Classes – Introduction

Remember Class-with-*Composistion* 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;
};
```

Operators in Classes – Introduction

Remember Class-with-*Composistion* 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;
}
```

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;
};
```

Note: A problem is encountered even in the simplest of cases !

```
error: non-static const member ‘bool const Date::m_gregorian’
can't use default assignment operator
```

Operator(s)

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

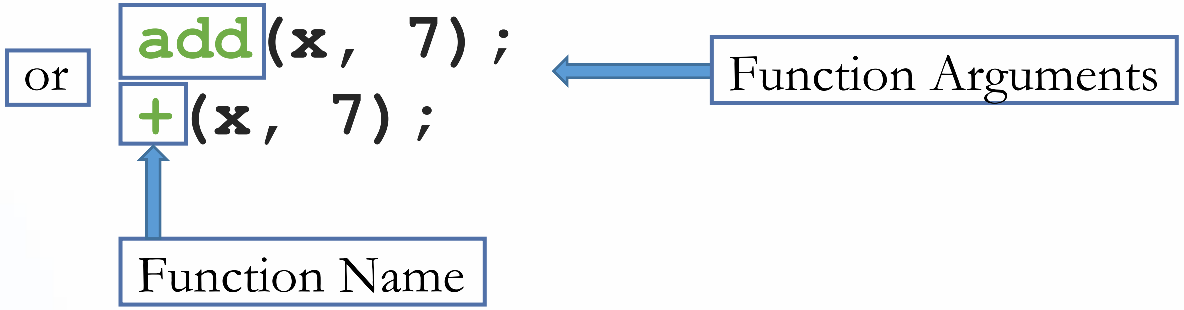
In reality they represent Functions.

➢ Simply "called" with different syntax:

```
x + 7;
```

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

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



```
   add(x, 7);
or
   +(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?

```
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;`
- `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;`

Particular challenges to keep operation 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

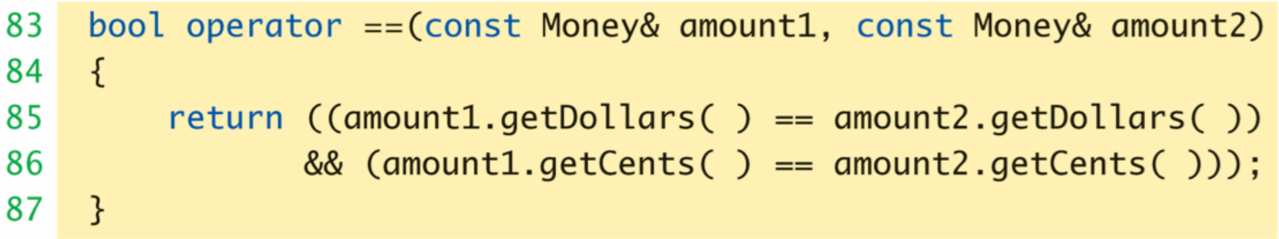
Overloading Operator(s)

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

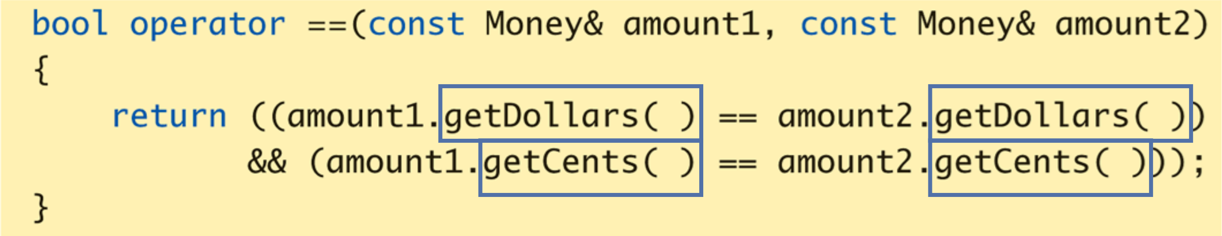
Overloading Operator(s)

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.

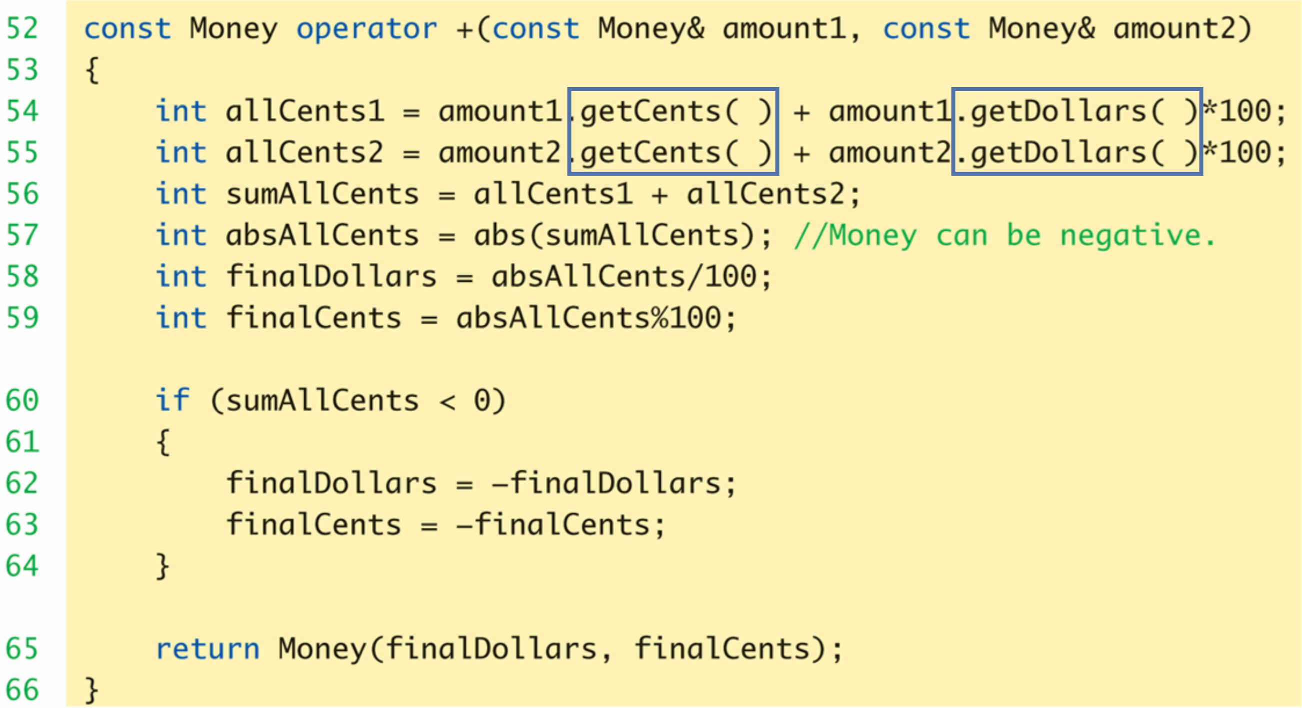
```
class Money{
 public:
  Money();
  Money(int dollars,
          int cents=0);
  Money(const Money & m);
  void setD/C(int dc);
  int getD/C() const;
 private:
  int m_dollars;
  int m_cents;
};
```

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);
}
```

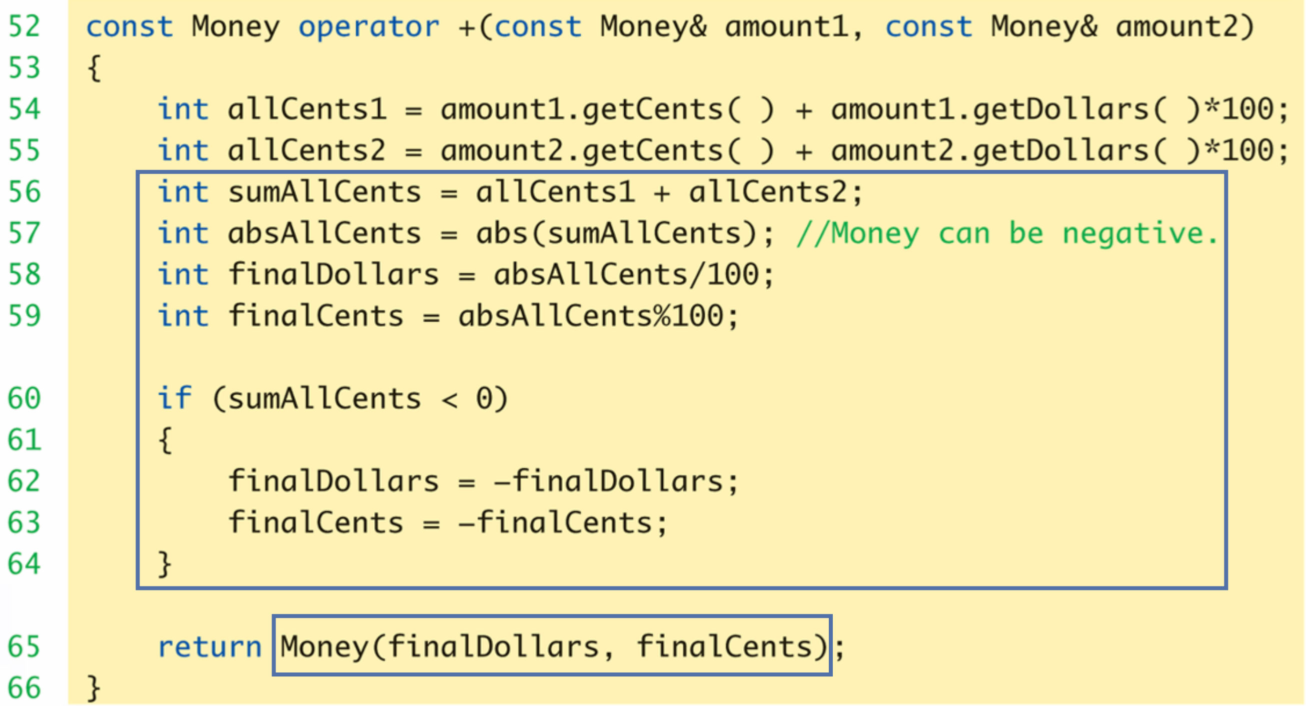
```
class Money{
 public:
  Money();
  Money(int dollars,
            int cents=0);
  Money(const Money & m);

  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 (+):

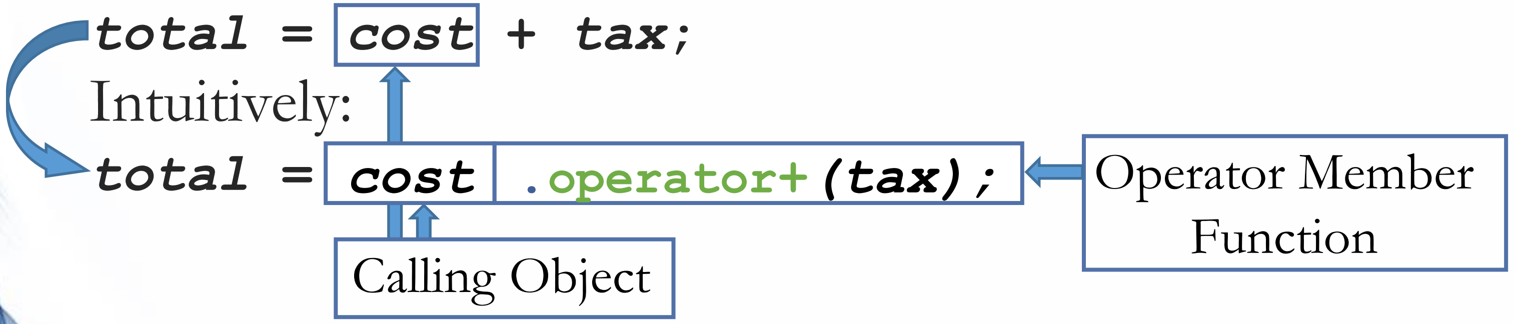
- ➢ A Member Function of Class *Money*.
- ➢ Calling Object serves as 1st parameter.

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);
```

Operator Member Function

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 (+):

- ➢ 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 has to guarantees *Deep* 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!

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

```
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;
};
```

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[](
               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];
};
```

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 !