COP 3331
OBJECT ORIENTED DESIGN
SUMMER 2017

WEEK 3 – WEDNESDAY (MAY 31ST):

- MORE ON CLASSES
- OPERATOR OVERLOADING

USF UNIVERSITY OF SOUTH FLORIDA

MORE ON CLASSES AND OBJECTS

THE INCLUDE GUARD

- When your main program file has an `#include` directive for a header file, there's a possibility that the header file will have an `#include` directive for a second header file

- If the main file also has an `#include` directive for the second header file, then the preprocessor will include the second header file twice

- You can use an include guard to prevent this
  - It prevents the header file from accidentally being included more than once

THE INCLUDE GUARD & DEFINE DIRECTIVE

- The syntax for an include guard is

```
#ifndef CONSTANT
#define CONSTANT
    …
#endif
```

- ifndef means "if not defined"
- The constant represents a version of the class that has already been loaded
- If the constant is not defined, then we use the `#define` directive to define it
- The `#endif` directive is used to enclose the definition of the class (from the `#ifndef` directive)
  - In other words, if not defined, create the class enclosed between the directive

INCLUDE GUARD EXAMPLE

```
// Specification file for the Rectangle class.
#ifndef RECTANGLE_H
#define RECTANGLE_H

// Rectangle class declaration.

class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};

#endif
```

STATIC CLASS MEMBERS

STATIC CLASS MEMBERS

- When we instantiate an object, each object has its own copies of the class variables.

- If a member variable is declared `static`, all instances of the class has access to that variable.
  - Remember, the `static` modifier 'preserves' the memory space

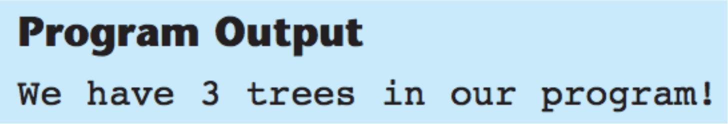
  - Even though static member variables are declared in a class, they are actually defined outside the class declaration.

STATIC CLASS VARIABLE EXAMPLE

```
// Tree class
class Tree
{
private:
   static int objectCount;    // Static member variable.
public:
   // Constructor
   Tree()
      { objectCount++; }

   // Accessor function for objectCount
   int getObjectCount() const
      { return objectCount; }
};

// Definition of the static member variable, written
// outside the class.
int Tree::objectCount = 0;
```

STATIC CLASS VARIABLE EXAMPLE

```
// This program demonstrates a static member variable.
#include <iostream>
#include "Tree.h"
using namespace std;

int main()
{
   // Define three Tree objects.
   Tree oak;
   Tree elm;
   Tree pine;

   // Display the number of Tree objects we have.
   cout << "We have " << pine.getObjectCount()
        << " trees in our program!\n";
   return 0;
}
```



Program Output

```
We have 3 trees in our program!
```

STATIC MEMBER FUNCTIONS

- A function that is a static member of a class can not access any non-static data in its class

- If a member function is declared `static`, it may be called without any instances of the class being defined.

STATIC MEMBER FUNCTIONS

- Why is it useful?
  - A class's static member functions can be called before any instances of the class are created
  - This means that the static member functions can access the static member variables of the class before any instances are defined.
  - This allows existence of static variables before objects creation.
  - Thus, allows us to create very specialized setup routines for class objects

FULL STATIC EXAMPLE

```
#ifndef BUDGET_H
#define BUDGET_H

// Budget class declaration
class Budget
{
private:
   static double corpBudget;  // Static member variable
   double divisionBudget;     // Instance member variable
public:
   Budget()
      { divisionBudget = 0; }

   void addBudget(double b)
      { divisionBudget += b;
        corpBudget += b; }

   double getDivisionBudget() const
      { return divisionBudget; }

   double getCorpBudget() const
      { return corpBudget; }

   static void mainOffice(double);  // Static member function
};

#endif
```

FULL STATIC EXAMPLE

```
#include "Budget.h"

// Definition of corpBudget static member variable
double Budget::corpBudget = 0;

// Definition of corpBudget static member function

void Budget::mainOffice(double moffice)
{
   corpBudget += moffice;
}
```

FULL STATIC EXAMPLE

```
// This program demonstrates a static member function.
#include <iostream>
#include <iomanip>
#include "Budget.h"
using namespace std;

int main()
{
   int count;                      // Loop counter
   double mainOfficeRequest;       // Main office budget request
   const int NUM_DIVISIONS = 4;  // Number of divisions

   // Get the main office's budget request.
   // Note that no instances of the Budget class have been defined.
   cout << "Enter the main office's budget request: ";
   cin >> mainOfficeRequest;
   Budget::mainOffice(mainOfficeRequest);

   Budget divisions[NUM_DIVISIONS]; // An array of Budget objects.
```

FULL STATIC EXAMPLE

```
// Get the budget requests for each division.
   for (count = 0; count < NUM_DIVISIONS; count++)
   {
      double budgetAmount;
      cout << "Enter the budget request for division ";
      cout << (count + 1) << ": ";
      cin >> budgetAmount;
      divisions[count].addBudget(budgetAmount);
   }

   // Display the budget requests and the corporate budget.
   cout << fixed << showpoint << setprecision(2);
   cout << "\nHere are the division budget requests:\n";
   for (count = 0; count < NUM_DIVISIONS; count++)
   {
      cout << "\tDivision " << (count + 1) << "\t$ ";
      cout << divisions[count].getDivisionBudget() << endl;
   }
   cout << "\tTotal Budget Requests:\t$ ";
   cout << divisions[0].getCorpBudget() << endl;
}
```

MEMBERWISE ASSIGNMENT AND COPY CONSTRUCTORS

MEMBERWISE ASSIGNMENT

- The = operator can be used to assign one object to another, or to initialize an object with other’s data

- Given two objects, `obj2 = obj1;` copies all member values from `obj1` and assigns to the corresponding members variables of `obj2`

MEMBERWISE ASSIGNMENT

- Example: consider the following class definition:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
   public:
      Rectangle(double, double);   // Constructor
      void setWidth(double);
      void setLength(double);
      double getWidth() const  { return width; }
      double getLength() const { return length; }
      double getArea() const { return width * length; }
};
#endif
```

MEMBERWISE ASSIGNMENT

- Declaring two objects of type Rectangle:

```
Rectangle box1(10.0, 10.0);
Rectangle box2(20.0, 20.0);
```

- Performing member assignment

```
box2 = box1;
```

- Can also perform memberwise assignment during initialization:

```
Rectangle box2 = box1;
```

MEMBERWISE ASSIGNMENT

- Memberwise assignment works well in most cases, except for one:

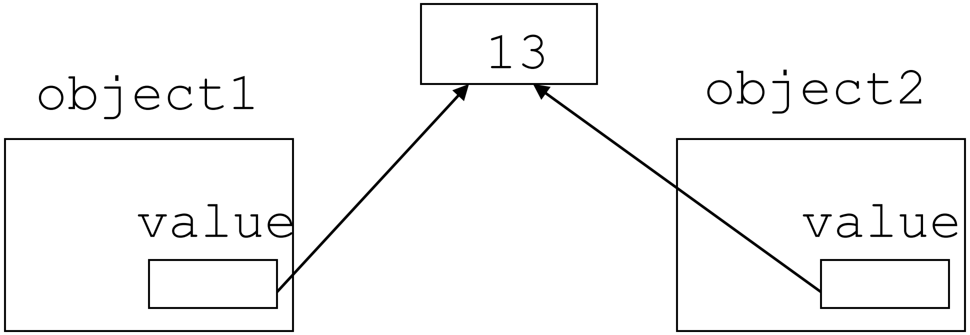
- Consider this class definition consisting of a pointer:

```
class SomeClass
{
   private:
      int *value;
   public:
      SomeClass(int val = 0)
      {value=new int; *value = val;}
      int getVal();
      void setVal(int);
}
```

MEMBERWISE ASSIGNMENT

- When we perform memberwise copy with objects containing dynamic memory

```
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // also 13
```

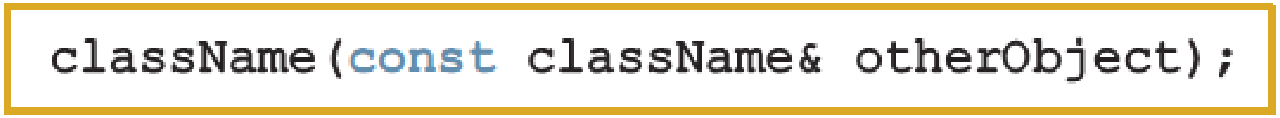


COPY CONSTRUCTORS

- The solution to this problem is to create a *copy constructor*
- A copy constructor is a special constructor that is called when an object is initialized with another object’s data.
- It has the same form as other constructors, except it has a reference parameter of the same type as the object itself
  - reference parameters MUST be used by copy constructors

COPY CONSTRUCTORS

- Syntax:



```
className(const className& otherObject);
```

  - Since copy constructors are required to use reference parameters, the const prevents the constructor from modifying the arguments data

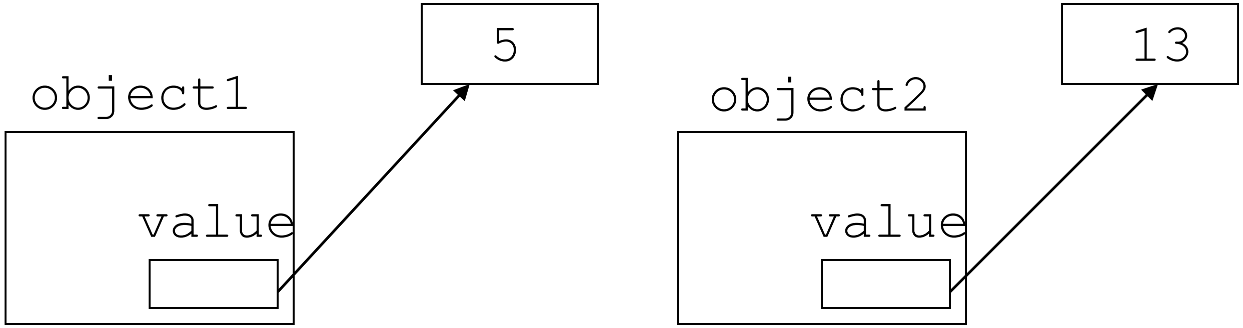
- Example:

```
SomeClass::SomeClass(const SomeClass &obj)
{
   value = new int;
   *value = obj.value;
}
```

COPY CONSTRUCTORS

- Each object now points to separate dynamic memory:

```
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // still 5
```



EXAMPLE:

```
const double DEFAULT_SCORE = 0.0;

class StudentTestScores
{
private:
    string studentName;  // The student's name
    double *testScores;  // Points to array of test scores
    int numTestScores;   // Number of test scores

    // Private member function to create an
    // array of test scores.
    void createTestScoresArray(int size)
    { numTestScores = size;
        testScores = new double[size];
        for (int i = 0; i < size; i++)
            testScores[i] = DEFAULT_SCORE; }

public:
    // Constructor
    StudentTestScores(string name, int numScores)
    { studentName = name;
        createTestScoresArray(numScores); }
```

EXAMPLE:

```
// Copy constructor
    StudentTestScores(const StudentTestScores &obj)
    { studentName = obj.studentName;
        numTestScores = obj. numTestScores;
        testScores = new double[numTestScores];
        for (int i = 0; i < numTestScores; i++)
            testScores[i] = obj.testScores[i]; }
```

OPERATOR OVERLOADING

OPERATOR OVERLOADING

- C++ allows you to redefine standard operators <u>when used with class objects</u>
- Why is this necessary?
  - Assignment and member selections are the only built-in operations on classes
  - Therefore, other operators can’t be applied directly to class objects
- Operator overloading provides a way to create more intuitive code

OPERATOR OVERLOADING…

- Consider:
  - Which would be preferable? (Suppose today is an object)

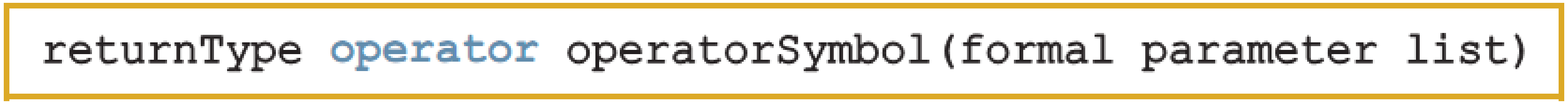
    `today.add(5);` OR `today += 5;`

- Most existing C++ operators can be overloaded to manipulate class objects

- The `operator` function is used to overload the operator

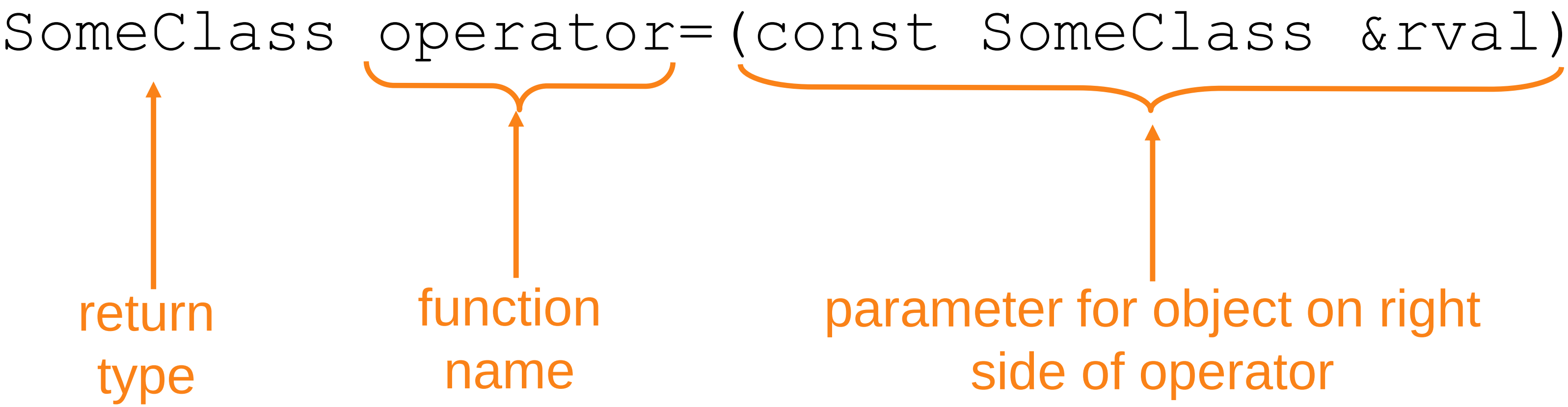
OPERATOR OVERLOADING…

- Syntax:



```
returnType operator operatorSymbol(formal parameter list)
```

- Example:



```
SomeClass operator=(const SomeClass &rval)
```

return type

function name

parameter for object on right side of operator

- Operator is called via object on left side

OPERATOR OVERLOADING…

- To overload an operator for a class:
  - Include operator function in the class definition
  - Write the definition of the operator function

- To call the overloaded operator function you could write:

```
object1.operator=(object2);
```

- However, you can call the overloaded operator in a more conventional form

```
object1 = object2;
```

OPERATOR OVERLOADING EXAMPLE…

- See “Student Test Score” Example on Canvas
  - Under “Code Examples” for Week 3

THE “THIS” POINTER

- Every object of a class maintains a (hidden) predefined *pointer to itself* called `this`

- When an object calls a member function, the `this` pointer is referenced by the member function

- The this pointer always points to the object of the class whose function is being called

OPERATOR OVERLOADING NOTES

- There are several rules/restrictions to consider when using operator overloading:
- C++ does NOT allow new operators to be created
  - This is why operator overloads are an option!
- Operator overloading is NOT automatic
  - Functions must be written to overload an operator

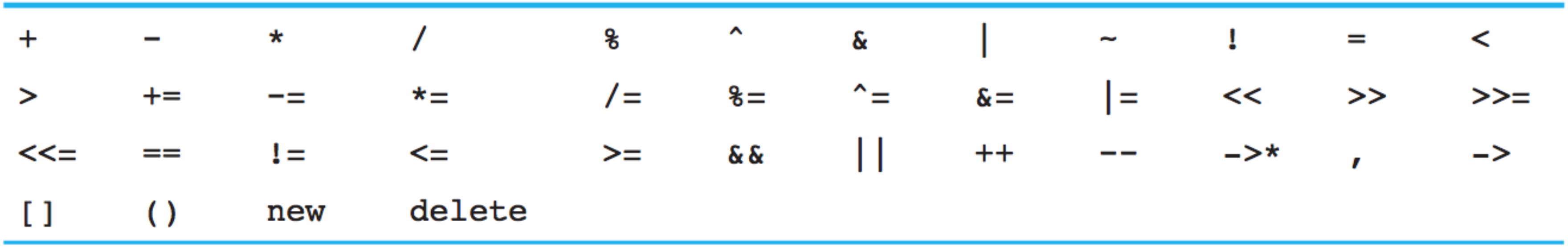
OPERATOR OVERLOADING NOTES

- Operator overloaded functions **must be non-static**, because they must be called on an object of the class and must operate on that object

- You do not have to perform overloaded operations on:
  - = (if you are performing memberwise assignment)
    - If you have a class with a pointer member, you should overload the operator (as shown in the example)
  - & (can return a pointer to the object)

OPERATOR OVERLOADING NOTES

- Most of C++’s operators can be overloaded:



+	-	*	/	%	^	&	|	~	!	=	<
>	+=	-=	*=	/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++	--	->*	,	->
[]	()	new	delete								

- The following operators cannot be overloaded:

```
.   .*   ::   ?:   sizeof
```

- You cannot change an operators precedence, associativity or “arity” (e.g. binary, unary)

OPERATOR OVERLOADING NOTES

- Overloading Math Operators are very useful in classes
- ++, -- operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a `bool` value
- Overloaded stream operators must return reference to `istream`, `ostream` and take `istream`, `ostream` objects as parameters

MATH OPERATOR OVERLOADING EXAMPLE

- See FeetInches code on canvas

PROJECT 2

- Programming assignment 2 will be posted today (Wednesday May 31st) on Canvas.