CSCI 123 Introduction to Programming Concepts in C++

Brad Rippe

Overloaded Operators, friends, and more arrays
Overview

11.1 Friend Functions

11.2 Overloading Operators

11.3 Arrays and Classes
11.1

Friend Functions
Object-oriented programming (OOP) involves programming using objects. An object represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors. The state of an object consists of a set of data fields (also known as properties) with their current values. The behavior of an object is defined by a set of functions.
Friend Function

• Class operations are typically implemented as member functions

• Some operations are better implemented as ordinary (nonmember) functions
Program Example: An Equals Function

• The Bike class from the last lecture can be enhanced to include an equals function
  – An equals function tests two objects of type Bike to see if their values represent the same bike
  – Two bikes are equal if they represent the same name, frame size, and wheel diameter
Declaration of
The equals Function

• We want the equals function to return a value of type bool that is true if the bikes are the same
• The equals function requires a parameter of type Bike
• The declaration is

    bool equals(Bike& aBike);

    – Notice we need an instance (in memory representation of) of a Bike to call equals
Defining Function equal

- The function equal, is a member function
  - However, we must use the accessors to get data from aBike. We cannot simply use name == aBike.name. This provides the information hiding mechanism we discussed earlier.
  - Here’s the code:

```cpp
bool Bike::equals(Bike& aBike) {
    if (name != aBike.getName() ||
        size != aBike.getSize() ||
        wheelDiameter != aBike.getWheelDiameter()) {
        return false;
    }
    return true;
}
```
Using The Function equals

The equals function can be used to compare Bikes in this manner

```cpp
if(mtnBike.equals(mtnBike2))
    cout << "true\n" << endl;
else
    cout << "false\n" << endl;
```

This seems like a good solution! But wait! C++ allows more than one way to skin a cat!
Is equals Efficient?

- Function equals could be made more efficient
  - Equal uses member function calls to obtain the private data values
  - This efficiency is traded for good coding practices
  - Ensure that no one has access the private data
  - More on this in a bit
  - Direct access of the member variables would be more efficient (faster) ?? Why?
A More Efficient equal

- As defined here, equal is more efficient, but not legal

```cpp
bool equals(Bike& aBike1, Bike& aBike2) {
    if(aBike1.name != aBike2.name ||
       aBike1.size != aBike2.size ||
       aBike1.wheelDiameter !=
       aBike2.wheelDiameter) {
        return false;
    }
    return true;
}
```

- The code is simpler and more efficient
- Direct access of private member variables is **not** legal!
Friend Functions

• Friend functions are not members of a class, but can access private member variables of the class
  – A friend function is declared using the keyword friend in the class definition
    • A friend function is not a member function
    • A friend function is an ordinary function
    • A friend function has extraordinary access to data members of the class
  – As a friend function, the more efficient version of equal is legal
Declaring A Friend

The function equal is declared a friend in the abbreviated class definition here

class Bike {
public:
friend bool equals(Bike& aBike1, Bike& aBike2);
// rest of the public functions
private:
    // rest of the private functions and members
};

// notice the keyword “friend” before the declaration
Using A Friend Function

• A friend function is declared as a friend in the class definition
• A friend function is defined as a nonmember function without using the "::" operator
• A friend function is called without using the '.' operator
Friend Declaration Syntax

• The syntax for declaring friend function is
  class className {
      public:
          friend functionDeclaration1
          friend functionDeclaration2
          ...
          memberFunctionDeclaration1
          memberFunctionDeclaration2
          ...
      private:
          memberFunctionDeclarations
          memberVariableDeclarations
  };
Are Friends Needed?

• Friend functions can be written as non-friend functions using the normal accessor and mutator functions that should be part of the class

• *The code of a friend function is simpler and it is more efficient*

• They can be dangerous because the more functions that access private data of a class the more difficult it can be to remember who has rights to modify the private data and who does not

• Is it worth the small savings of not pushing data into another stack frame in order to get direct access to private members.

• Use friends carefully
Choosing Friends

• How do you know when a function should be a friend or a member function?
  – Choosing to make the nonmember function a friend is a decision of efficiency and personal taste
  – You will have to profile your code and see if you really benefit from using friend functions
  – My personal taste is that they are somewhat dangerous because they allow this direct access. Programmers later might not have the same knowledge and understanding of the system and as a result could write code that makes BAD THINGS HAPPEN
Program Example:
The Money Class (version 1)

• In your book example
• More with friend functions see pages 604 – 612 in your text - 6th Edition
• 620 – 623 in the 7th Edition
Parameter Passing Efficiency

• A call-by-value parameter less efficient than a call-by-reference parameter
  – The parameter is a local variable initialized to the value of the argument
    • This results in two copies of the argument
• A call-by-reference parameter is more efficient
  – The parameter is a placeholder replaced by the argument
    • There is only one copy of the argument
Class Parameters

• It can be much more efficient to use call-by-reference parameters when the parameter is of a class type
• When using a call-by-reference parameter
  – If the function does not change the value of the parameter, mark the parameter so the compiler knows it should not be changed
**const Parameter Modifier**

- To mark a call-by-reference parameter so it cannot be changed:
  - Use the modifier `const` before the parameter type
  - The parameter becomes a constant parameter
  - `const` used in the function declaration and definition
const Parameter Example

• A function declaration with constant parameters
  – friend bool equals(const Bike& aBike1, const Bike& aBike2);

• A function definition with constant parameters
  – bool equals(const Bike& aBike1, const Bike& aBike2) {
    ...
  }

• const modifier flags the parameters to tell the compiler that we are not changing the internal data of the parameters
const Considerations

• When a function has a constant parameter, the compiler will make certain the parameter cannot be changed by the function
• What if the parameter calls a member function?

```cpp
bool Bike::equals(const Bike& aBike) const {
    if(name != aBike.getName() ||
        size != aBike.getSize() ||
        wheelDiameter != aBike.getWheelDiameter()) {
        return false;
    }
    return true;
}
```

• Member functions getName, getSize, getWheelDiameter are declared const functions. This allows their use in the equals function because aBike is declared a const parameter. If they are not declared const functions, the compiler will complain.
More with const

- `bool equals(const Bike& aBike)`
  const modifier tells the compiler that we are not going to modify the parameter `aBike`.

if we use the dot notation in the function definition, as such:
  
  ```cpp
  aBike.getName()
  ```

`getName()` `const` must be declared and defined as a const function. Similar to the following function declaration below

- `bool Bike::equals(const Bike& aBike) const`
  const modifier tells the compiler that the calling object will not be modified. All functions that do not change member variables should be declared as a const function.
const Modifies Functions

• If a constant parameter makes a member function call...
  – The member function called must be marked so the compiler knows it will not change the parameter
  – const is used to mark functions that will not change the value of an object
  – const is used in the function declaration and the function definition
Function Declarations
With const

- To declare a function that will **not** change the value of any member variables:
  - Use const after the parameter list and just before the semicolon

```cpp
class Bike {
    public:
        ...
        bool equals(const Bike& aBike) const;
        ...
```
Function Definitions with const

• To define a function that will not change the value of any member variables:

• Use const in the same location as the function declaration

```cpp
bool Bike::equals(const Bike& aBike) const {
    // function definition
}
```
**const Wrapup**

- Using `const` to modify parameters of class types improves program efficiency
  - `const` is typed in front of the parameter's type
- Member functions called by constant parameters must also use `const` to let the compiler know they do not change the value of the parameter
  - `const` is typed following the parameter list in the declaration and definition
Use const Consistently

• Once a parameter is modified by using const to make it a constant parameter
  – Any member functions that are called by the parameter must also be modified using const to tell the compiler they will not change the parameter
  – It is a good idea to modify, with const, every member function that does not change a member variable
11.2

Overloading Operators
Overloading Operators

- In the Circle class adds new functions that allow us to do common operations on Circles.
- Before we could add, subtract, multiple, integral types, but what happens if we want to add two Circles together.
- Our user defined types don’t have operators +, -, == defined, so we will add those
- Before we couldn’t do:

```cpp
Circle circle1(24.0);
Circle circle2(10.0);
circle1 = circle1 + circle2;
```
Operators As Functions

• An operator is a function used differently than an ordinary function
  – An ordinary function call enclosed its arguments in parenthesis
    add(circle1, circle2)
  
  – With a binary operator, the arguments are on either side of the operator
    circle1 + circle2
Operator Overloading

- Operators can be overloaded
- The definition of operator + for the Circle class is nearly the same as member function add
- To overload the + operator for the Circle class
  - Use the name + in place of the name of a function name
  - Use keyword operator in front of the +
  - Example:
    ```cpp
    friend Circle operator +(const Circle& aCircle, const Circle& aCircle2);
    ```
Operator Overloading Rules

• At least one argument of an overloaded operator must be of a class type
• An overloaded operator can be a friend of a class
• New operators cannot be created
• The number of arguments for an operator cannot be changed
• The precedence of an operator cannot be changed
• ., ::, *., and ? cannot be overloaded
Program Example: Overloading Operators

- The Circle class with overloaded operators +, -, and == is in the file overloading.cpp
Automatic Type Conversion

• With the right constructors, the system can do type conversions for your classes
  – This code from conversion.cpp actually works

    Circle circle1(24.0);
    circle1 = circle1 + 10.0;

  – The double 10.0 is converted to type Circle so it can be added to circle1!
  – How does that happen?
Type Conversion Step 1

- When the compiler sees `circle1 + 10.0`, it first looks for an overloaded `+` operator to perform:

  ```cpp
  Circle + double
  ```

  - If it exists, it might look like this:

  ```cpp
  friend Circle operator +(const Circle& aCircle, double aDub);
  ```
Type Conversion Step 2

- When the appropriate version of + is not found, the compiler looks for a constructor that takes a double
  
  - The Circle constructor that takes a single parameter of type double
  
  - The constructor Circle(double aRadius) converts 10.0 to a Circle object so the two values can be added!
Type Conversion Again

• Although the compiler was able to find a way to add
  
  circle1 + 10.0

  this addition will cause an error
  
  circle1 + “10”;

  – There is no constructor in the Circle class that takes an argument of type string
A Constructor For double

- To permit circle1+ “10”, the following constructor should be declared and defined

```cpp
class Circle {
    public:
        ...
        Circle(const char aRadius[]);
        // Initialize object so its radius is cstring convert to an integer
        ...
    }
```
Overloading Unary Operators

• Unary operators take a single argument
• The unary – operator is used to negate a value
  \[ x = -y \]
• ++ and -- are also unary operators
• Unary operators can be overloaded
  – The new Circle class has the ++ and – operators defined in addition to the binary operators
Overloading -

- Pre and postfix unary operators

```cpp
// prefix
friend Circle operator ++(Circle& aCircle);
// postfix
friend Circle operator ++(Circle& aCircle, int increment);
// insertion operator
friend ostream& operator <<(ostream& aOut, const Circle& aCircle);
```

The int parameter denotes the postfix operator, it’s the compilers way of determining which version you’re overloading.
Overloading `<<` and `>>`

• The insertion operator `<<` is a binary operator
  – The first operand is the output stream
  – The second operand is the value following `<<`

```
cout << "Hello out there.\n";
```

Operand 1  Operator  Operand 2
Replacing Function output

• Overloading the `<<` operator allows us to use `<<` to output a circle
• We could use another function like output Circle, but the `<<` operator is straightforward
• Given the declaration: `Circle circle1(24.0);`

```cpp
Circle circle1(24.0);
cout << circle1 << endl;
// Note the function call the `<<` operator
// operator `<<(cout, circle1);`
```
What Does << Return?

• Because << is a binary operator

cout << "Circle1's radius is (postfix ++) " << circle1++ << endl;

  seems as if it could be grouped as
((cout << "Circle1's radius is (postfix ++) ") << circle1++) << endl;

• To provide cout as an argument for  << circle1++,
  (cout << "Circle1's radius is (postfix ++) ") must return cout
Overloaded $\ll$ Declaration

• Based on the previous example, $\ll$ should return its first argument, the output stream
  – This leads to a declaration of the overloaded $\ll$ operator for the Circle class:

```cpp
class Circle {
public:
  ...
  friend ostream& operator$\ll$(ostream& output, const Circle& aCircle);
  ...
```
Overloaded « Definition

• The following defines the « operator

ostream& operator<<(ostream& aOutput, const Circle& aCircle) {
    output << aCircle.aRadius;
    return aOutput;
}
Return ostream& ?

• The & means a reference is returned
  – So far all our functions have returned values
• The value of a stream object is not so simple to return
  – The value of a stream might be an entire file, the keyboard, or the screen!
• We want to return the stream itself, not the value of the stream
• The & means that we want to return the stream, not its value
Overloading >>

- Overloading the `>>` operator for input is very similar to overloading the `<<` for output

`>>` could be defined this way for the Circle class

```cpp
istream& operator>>(istream& aInput, Circle&aCircle) {
  input >> aCircle.aRadius;
  return aInput;
}
```
11.3

Arrays and Classes
Arrays and Classes

• Arrays can use structures or classes as their base types
  – Example:

    Circle myCircles[5];
Accessing Members

• When an array's base type is a structure or a class...
  – Use the dot operator to access the members of an indexed variable
  – Example:

```cpp
for(int i = 0; i < 5; i++) {
    cout << (i+1) << " ";
    cout << myCircles[i].getRadius() << endl;
}
```
An Array of Circles

• The Circle class can be the base type for an array. In addition, the Bike class can be the base type for an array as well.

• When an array of classes is declared
  – The **default constructor** is called to initialize the indexed variables
Arrays as Structure Members

• A structure can contain an array as a member
• You’ve seen this from last lecture
  – Example:
    ```
    struct Student {
        char name[256];
        char sID[256];
    };
    
    Student student1;
    ```
Accessing Array Elements

• To access the array elements within a structure
  – Use the dot operator to identify the array within the structure
  – Use the [ ]'s to identify the indexed variable desired
  – Example: student1.name[i] references the ith indexed variable of the variable name in the structure student1
  – This retrieves the first character from the student’s name
Arrays as Class Members

• Class Square includes an array
  – The array, named points, contains x and y coordinates
  – Member variable size is the number of items stored

```cpp
class Square {
public:
    static const int MAX_POINTS = 4;
    friend istream& operator>>(istream& aInput, Square& aSquare);
    friend ostream& operator<<(ostream& aOutput, Square& aSquare);
private:
    Point points[MAX_POINTS];
};
```