OPERATOR OVERLOADING

Chapter 10

TOPICS

Fundamentals of Operator Overloading

Overloading Binary Operator + (Addition)

Overloading the Binary Operator < (Comparison)

Overloading the Unary Increment Operator ++

FUNDAMENTALS OF OPERATOR OVERLOADING

WHAT IS OVERLOADING?

We've seen overloading applied to functions in chapter 6:

Overloading – Defining functions with the same name, in the same scope, but with different signatures

Often used to perform similar tasks on different data types

For example, computing an average is a task that can be performed on many different numeric data types (int, float, double, etc.)

FUNCTION OVERLOADING – BRIEF EXAMPLE

But it would get confusing to write a different function with a different name for every possible variation of numbers

So we can create overloaded functions that use the same name, and just change up the parameters. For example:

int **computeAverage** (int, int);
double **computeAverage** (double, double);
float **computeAverage** (float, float);
float **computeAverage** (float, float, float);

This can make programs easier to read and understand

OPERATORS ARE SYMBOLS

Operators are the symbols we use for arithmetic, accessing arrays, stream extraction & insertion, etc.

In a statement like this:

y = x + 1;

How does the C++ compiler know how to interpret the addition operator + ?

ADDITION OPERATOR

y = x + 1;

How does the C++ compiler know how to interpret the addition operator + ?

There is code in the C++ compiler that handles the + operator by performing addition

Addition Operator with a Float Type

What if we changed it up, and x is a float instead of an int:

x = x + 1.43;

It's ok, the compiler's addition code can handle other data types too.

Which is just like function overloading – handling different data types using the same name.

In this case, it's an operator, not a function

OPERATORS CAN BE OVERLOADED IN C++

Some are already built into the C++ Standard Library:

Arithmetic operators (**+ – * /** etc.) can handle any numeric data type

Stream insertion and extraction operators **<<** and **>>** can handle any data type

Array brackets **[]** are operators too – arrays can store any data type

Etc.

PROGRAMMERS CAN OVERLOAD OPERATORS TOO – EXAMPLE 1

Add 10,000 to the price of a **Car**, which is a user-defined class that handles car-related functionality:

We could do this:

```
Car car1(15000, "SUV");

newPrice = car1.getPrice();

Car1.setPrice(newPrice + 10000);
```

PROGRAMMERS CAN OVERLOAD OPERATORS TOO – EXAMPLE 2

But what if we could do this instead?:

Car car1(15000, "SUV");

car1 = car1 + 10000;

This code is readable and [relatively] self-explanatory, but **addition using the addition operator + is only defined for fundamental numeric types, not class types**

So, we must **overload** the **+** operator

Operator Overloading – Definition

Adds functionality to a class so that operators can work with class objects

In other words, operator overloading allows **objects of a class** to be used as **operands in arithmetic**

Use a special operator notation syntax

Operator Overloading – Why Bother?

Can use regular functions & function calls instead (usually sets and gets), but operator notation can be easier to read and understand

First, some rules…

RULES FOR OPERATOR OVERLOADING

Cannot change:

- Precedence, associativity, number of operands (unary must stay unary, binary must stay binary, etc.), how operator works with fundamental data types

Cannot create new operators

Related operators, like **+** and **+=**, must be overloaded separately

Overloaded **() [] ->** or any assignment operator must be a class function

All other operators can be overloaded as member or non-member functions

Operators That Cannot Be Overloaded

- `.` dot operator
- `.*` pointer to member
- `::` scope resolution operator
- `?:` ternary conditional operator

Operators That Don't Need to be Overloaded

These are already overloaded for objects:

= assignment operator

& address operator

, comma operator

Software Engineering Observation

Overload operators for class types so they work as closely as possible to the way built-in operators work on fundamental types.

Which is the point of overloading operators anyway…

Overloading the Binary Operator + (Addition)

Overloading an Operator Means Writing a Function

Remember, we're writing a function in a class

We still need:

- **function prototype** in the header file (.h)
- **function definition** in the implementation file (.cpp)

And the function must be called on, and act on, an **object** of the class

- So, declared as class member functions and cannot be declared static

The syntax will look a little bit different, but we still must follow these rules for functions

BACK TO THE CAR EXAMPLE

We want to add 10000 to a Car:

Car car1(15000, "SUV");

car1 = car1 + 10000;

What does it mean to add 10000 to a Car?

- For our purposes, it means the new price is 10000 more than the old price

We need to analyze this expression…

Car Example – Step 1 – Identify Each Part of the Expression

car1 = car1 + 10000;

	What is it?		
car1 left side of =	Car object		
car1 right side of =	Car object		
+	operator		
10000	integer		

Car Example – Step 2 – Identify the Purpose of Each Part

car1 = car1 + 10000;

	What is it?	Purpose	
car1 on left side of =	Car object	result	
car1 on right side of =	Car object	operand 1	
+	operator	operator	
10000	integer	operand 2	

Car Example – Step 3 – Each Part as Part of a Function Call

car1 = car1 + 10000;

	What is it?	Purpose	If expression was a function call
car1 on left side of =	Car object	result	return value
car1 on right side of =	Car object	operand 1	object where function is located
+	operator	operator	name of function
10000	integer	operand 2	argument

OPERATOR OVERLOADING – PROTOTYPE/HEADER SYNTAX

General format:

returnType className::**operator***symbol (parameter list)*

operator is a keyword and is required

symbol is the operator being overloaded: **+ – ++** , etc.

CAR EXAMPLE – PROTOTYPE/HEADER

So the prototype & header for this Car example could look like this:

Car Car::operator+ (const int)

This uses **pass-by-value to return a copy of a Car object**

- Important – we're creating another Car object within the function

CarExampleOverloadAddition.dev

- Car.h, Car.cpp, OverloadedCarDriver.cpp

Overloading the Binary Operator < (Comparison)

Another Overloaded Binary Operators Example – Comparison <

Suppose we have a class called **MyClass**

We have two objects of this class: **obj1** and **obj2**

We want to be able to write:

```
if (obj1 < obj2)
    // do stuff
```

First, we need to decide what it means for one object to be “less than” another object of the same class

What Does It Mean for One Object to Be Less Than Another?

if (obj1 < obj2)

Examples:

- Date objects: is obj1's date before obj2's date
- Car objects: is obj1's price less than obj2's price
- Student objects: is obj1's name before obj2's name in a dictionary

Regardless of the meaning, the result of the expression is **true** or **false**

Now we can overload the **< operator** in the class

CAR EXAMPLE 2 – USE OVERLOADED < TO COMPARE TWO CARS

We want to compare the prices of two Car objects:

```
Car car1(10000, "SUV");

Car car2(15000, "sedan");

if (car1 < car2)
    // print message
```

Again, we need to analyze the expression…

Car Example 2 – Step 1 – Identify Each Part of the Expression

car1 < car2

	What is it?		
car1	Car object		
car2	Car object		
<	operator		

Car Example 2 – Step 2 – Identify the Purpose of Each Part

car1 < car2

	What is it?	Purpose	
car1	Car object	operand 1	
car2	Car object	operand 2	
<	operator	operator	

CAR EXAMPLE 2 – STEP 3 – EACH PART AS PART OF A FUNCTION CALL

car1 < car2

	What is it?	Purpose	If expression was a function call
car1	Car object	operand 1	object where function is located
car2	Car object	operand 2	argument
<	operator	operator	name of function

CAR EXAMPLE 2 – STEP 4 – WHAT ELSE DO WE NEED?

Remember that the expression is part of a selection statement:

if (car1 < car2)

So we also need to handle the result of the comparison, which could be true or false

Car Example 2 – Step 4 – We Need Also Need a Return Value

car1 < car2

	What is it?	Purpose	If expression was a function call
car1	Car object	operand 1	object where function is located
car2	Car object	operand 2	argument
<	operator	operator	name of function
true, false	boolean	result	return value

CAR EXAMPLE 2 – STEP 5 – USE THESE PARTS IN A PROTOTYPE/HEADER

So the prototype & header for this example would look like this:

bool Car::operator< (const Car&) const;

Note use of **const**, since neither operand will be changed

const at the end refers to the Car object where the function is located (operand 1)

const Car& – refers to the second Car object (operand 2)

CAR EXAMPLE 2 – EXAMPLE CODE

CarExampleOverloadComparison.dev

- Car.h, Car.cpp, OverloadedCarDriver.cpp

Overloading the Unary Increment Operator ++

Overloading Prefix and Postfix Operators

Remember that these act slightly differently:

Prefix operator

1. Increments or decrements the operand
2. Then uses the updated value

Postfix operator

1. Uses the current value of the operand
2. Then increments or decrements it

Overloading Prefix and Postfix Operators 2

Remember that these act slightly differently:

Prefix operator

1. Increments or decrements the operand
2. Then uses the updated value

Postfix operator

1. Uses the current value of the operand
2. Then increments or decrements it

Uses means returns in this example

So each overloaded operator function must have a **distinct signature**

Overloading the Unary Increment Prefix Operator ++

Suppose we have a Date object **date1**

- Contains data members month, day, year

We want to add one to its day member by writing **++date1**

Remember what the ++ is a shortcut for:

++date1 is a shorter way to write **date1 += 1**

date1 += 1 is a shorter way to write **date1 = date1 + 1**

Overloading Prefix ++ in Date Example – Analyze the Parts

++date1

(which is really **date1 = date1 + 1**)

	What is it?	Purpose	If expression was a function call
date1 on "left" side of =	Date object	result	return value
date1 on "right" side of =	Date object	operand 1	object where function is located
++	operator	operator	name of function
1 implied	integer	operand 2	argument

Overloading Prefix ++ in Date – Operand1 & Result are the Same Object

++date1

(which is really **date1 = date1 + 1**)

	What is it?	Purpose	If expression was a function call
date1 on "left" side of =	Date object	result	return value
date1 on "right" side of =	Date object	operand 1	object where function is located
++	operator	operator	name of function
1 implied	integer	operand 2	argument

Overloading Prefix ++ in Date – Use a Reference Instead of an Object

Prototype for this function would be:

Date& Date::operator++();

Use pass-by-reference, so we don't need to make a copy of the object

DateExampleOverloadPrefix.dev

- Date.h, Date.cpp, OverloadedDateDriver.cpp

OVERLOADING UNARY POSTFIX ++ OPERATOR

Compiler must be able to distinguish between pre and postfix

Using a member function, by convention, compiler generates function call:

date1.operator++(0)

0 is a dummy value to enable distinction between pre- and postfix

So a prototype for this function would be:

Date& operator++(int);