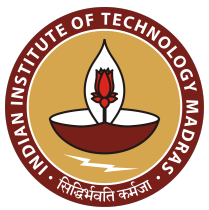
Operator Overloading

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

September 3, 2024



Operator Overloading

- Operator overloading is one of the most exciting features of object-oriented programming.
- It can transform complex, obscure program listings into intuitively obvious ones.

- statements like:
  `d3.addobjects(d1, d2);`
  or the similar but equally obscure `d3 = d1.addobjects(d2);`
  can be changed to the much more readable
  `d3 = d1 + d2;`

The term *operator overloading* refers to giving the normal C++ operators, such as $+$, $*$, $<=$, and $+=$, additional meanings when they are applied to user-defined data types.

Operator Overloading

- Normally
  `a = b + c;`
  works only with basic types such as `int` and `float`
- attempting to apply it when a, b, and c are objects of a user-defined class will cause complaints from the compiler.
- using overloading, one can make this statement legal even when a, b, and c are user-defined types
- If one finds oneself limited by the way the C++ operators work, one can change them to do whatever they want.
- operator overloading gives one the opportunity to redefine the C++ language
- using classes to create new kinds of variables
- operator overloading to create new definitions for operators
- one can extend C++ to be, in many ways, a new language of their own design.

Overloading Unary Operators

- An operand is simply a variable acted on by an operator
- unary operators act on only one operand
- Examples of unary operators are the increment and decrement operators $++$ and $--$, and the unary minus, as in -33.
- In the ”COUNTER” example, we created a class Counter to keep track of a count. Objects of that class were incremented by calling a member function:
  `c1.inc_count();`
- That did the job, but the listing would have been more readable if we could have used the increment operator ++ instead:
  `++c1;`
- programmers would guess immediately that this expression increments c1
- Let’s rewrite COUNTER to make this possible (see countpp1.cpp)

Overloading Unary Operators

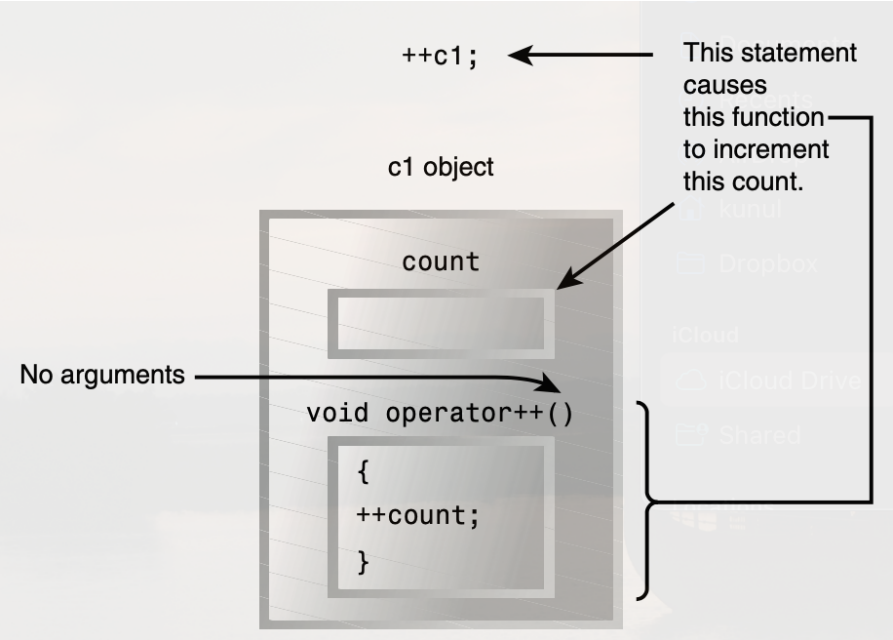
- How do we teach a normal C++ operator to act on a user-defined operand?
- The keyword operator is used to overload the ++ operator in this declarator:
  `void operator ++ ()`
- The return type (void in this case) comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses (which are empty here).
- This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand (the variable operated on by the ++) is of type Counter.

Overloading Unary Operators

- How does the compiler distinguish between standard operators and overloaded operators?
- In case of “Functions,” the only way the compiler can distinguish between overloaded functions is by looking at the data types and the number of their arguments.
- Similarly, with overloaded operators by looking at the data type of their operands.
- If the operand is a basic type such as an `int`, as in
  `++intvar;`
  then the compiler will use its built-in routine to increment an int.
- But if the operand is a ”`Counter`” variable, the compiler will know to use our user-written `operator++()` instead.

Operator Arguments

- In `main()` the ++ operator is applied to a specific object, as in the expression ++c1.
- Yet `operator++()` takes no arguments.
- What does this operator increment?
- It increments the count data in the object of which it is a member.
- But how ?
- Since member functions can always access the particular object for which they've been invoked, this operator requires no arguments.



Operator Return Values

- The `operator++()` function in the "COUNTPP1" program has a subtle defect.
- One will discover it if you use a statement like this in `main()`:
  `c1 = ++c2;`
- The compiler will complain.
- Why? Because we have defined the ++ operator to have a return type of void in the `operator++()` function, while in the assignment statement it is being asked to return a variable of type Counter.
- That is, the compiler is being asked to return whatever value c2 has after being operated on by the `++ operator`, and assign this value to c1.
- So as defined in "COUNTPP1", we can't use ++ to increment Counter objects in assignments; it must always stand alone with its operand.
- Of course the normal ++ operator, applied to basic data types such as int, would not have this problem

Overloading Unary Operators

- To make it possible to use our homemade operator ++() in assignment expressions, we must provide a way for it to return a value.
- Here the `operator++()` function creates a new object of type Counter, called temp, to use as a return value.
- It increments the count data in its own object as before, then creates the new temp object and assigns count in the new object the same value as in its own object.
- Finally, it returns the temp object.
- see ”countpp2.cpp”.

Nameless Temporary Objects

- In COUNTPP2 we created a temporary object of type Counter, named temp, whose sole purpose was to provide a return value for the ++ operator. This required three statements.
- There are more convenient ways to return temporary objects from functions and overloaded operators. (see countpp3.cpp)

Overloading Unary Operators

- In this program a single statement
  `return Counter(count);`
  does what all three statements did in COUNTPP2.
- This statement creates an object of type Counter.
- This object has no name; it won't be around long enough to need one.
- This unnamed object is initialised to the value provided by the argument count.
- Doesn't this require a constructor that takes one argument?
- It does, and to make this statement work we inserted just such a constructor into the member function list in COUNTPP3.

```
Counter(int c) :  count(c) //constructor,
one arg
{ }
```

- Once the unnamed object is initialized to the value of count, it can then be returned.

Postfix Notation

- So far we've shown the increment operator used only in its prefix form.
  `++c1`
- What about postfix, where the variable is incremented after its value is used in the expression?
  `c1++`
- To make both versions of the increment operator work, we define two overloaded ++ operators
- see "postfix.cpp"
- Now there are two different declarators for overloading the ++ operator. The one we've seen before, for prefix notation, is
  `Counter operator ++ ()`
- The new one, for postfix notation, is
  `Counter operator ++ (int)`
- The only difference is the `int` in the parentheses.
- It's simply a signal to the compiler to create the postfix version of the operator.
- The designers of C++ are fond of recycling existing operators and keywords to play multiple roles, and int is the one they chose to indicate postfix.

Overloading Binary Operators

Binary operators can be overloaded just as easily as unary operators.

We'll look at examples that overload:

- arithmetic operators,
- comparison operators,
- arithmetic assignment operators.

Arithmetic Operators

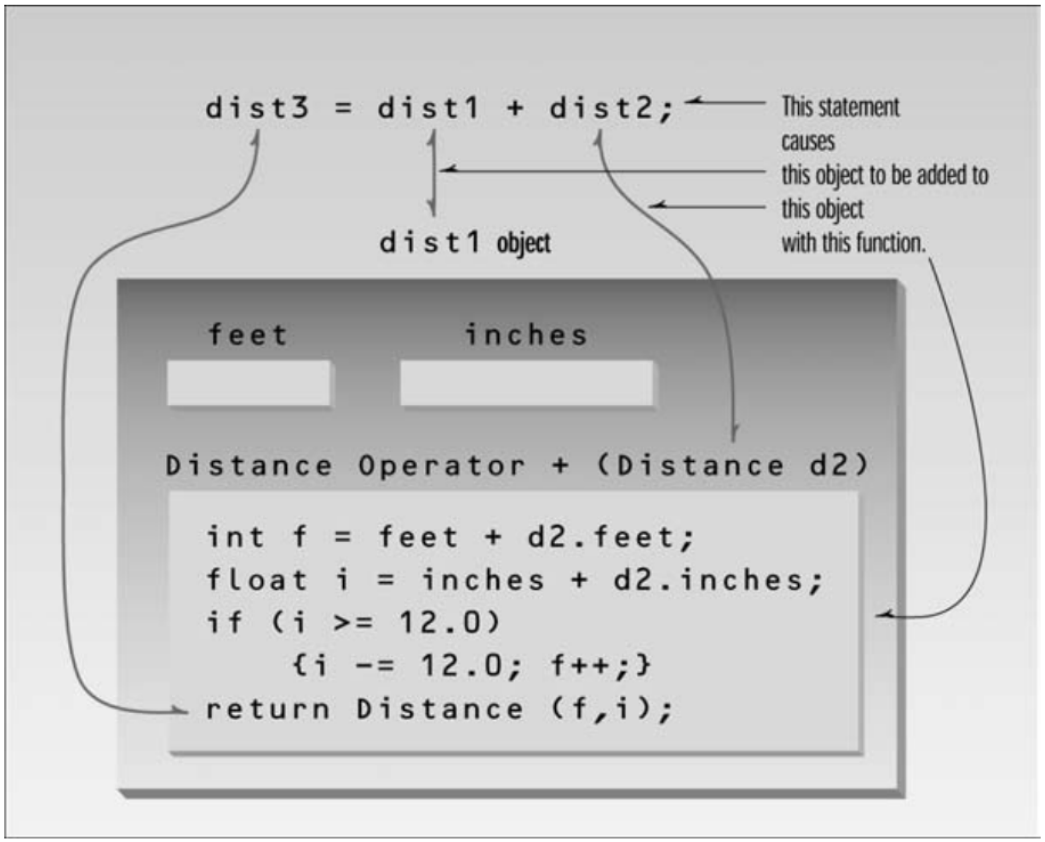
- (ENGLCON) two English Distance objects could be added using a member function `add_dist()`:
  ```
  dist3.add_dist(dist1, dist2);
  ```
- By overloading the + operator we can reduce this dense-looking expression to
  ```
  dist3 = dist1 + dist2;
  ```
- see "englplus.cpp"

Overloading Binary Operators

- In class Distance the declaration for the `operator+ ()` function looks like this:
  `Distance operator + ( Distance );`
- This function has a return type of `Distance`, and takes one argument of type `Distance`.
- In expressions like
  `dist3 = dist1 + dist2;`
  how the return value and arguments of the operator relate to the objects ?
- When the compiler sees this expression it looks at the argument types, and finding only type `Distance`, it realizes it must use the `Distance` member function `operator+()`.
- But what does this function use as its argument—`dist1` or `dist2`?
- Doesn't it need two arguments, since there are two numbers to be added?

Overloading Binary Operators

- The argument on the left side of the operator (`dist1` in this case) is the object of which the operator is a member.
- The object on the right side of the operator (`dist2`) must be furnished as an argument to the operator.
- The operator returns a value, which can be assigned or used in other ways; in this case it is assigned to `dist3`.



Comparison Operators

- we'll overload the less than operator $(<)$ in the `Distance` class so that we can compare two distances.
- see "engless.cpp"
- The approach used in the `operator<()` function is similar to overloading the + operator, except that here the `operator<()` function has a return type of `bool`.
- The return value is false or true, depending on the comparison of the two distances.
- The comparison is made by converting both distances to floating-point feet, and comparing them using the normal ¡ operator.

Arithmetic Assignment Operators

- the += operator, combines assignment and addition into one step
- see ”englpleq.cpp”
- In this program the addition is carried out in main() with the statement
  `dist1 += dist2;`
- This causes the sum of `dist1` and `dist2` to be placed in `dist1`.
- Notice the difference between the function used here, `operator+=()`, and `operator+()` used earlier.
- In the earlier `operator+()` function, a new object of type `Distance` had to be created and returned by the function so it could be assigned to a third `Distance` object, as in
  dist3 = dist1 + dist2;
- In the `operator+=()` function the object that takes on the value of the sum is the object of which the function is a member.

Arithmetic Assignment Operators

- The `operator+=()` function has no return value; it returns type `void`.
- A return value is not necessary with arithmetic assignment operators such as +=, because the result of the assignment operator is not assigned to anything.
- The operator is used alone, in expressions like the one in the program.

```
dist1 += dist2;
```

- If we wanted to use this operator in more complex expressions, like

```
dist3 = dist1 += dist2;
```

- then you would need to provide a return value.
- This could be done by ending the `operator+=()` function with a statement like

```
return Distance(feet, inches);
```

in which a nameless object is initialized to the same values as this object and returned.

The Subscript Operator ([])

- The subscript operator, [], which is normally used to access array elements, can be overloaded.
- This is useful if we want to modify the way arrays work in C++.
- For example, we might want to make a "safe" array:
  - One that automatically checks the index numbers we use to access the array
  - to ensure that they are not out of bounds
- To be useful, the overloaded subscript operator must return by reference.
- To see why this is true, we'll see three example programs that implement a safe array, each one using a different approach to inserting and reading the array elements:
  - Separate `put()` and `get()` functions
  - A single `access()` function using return by reference
  - The overloaded [] operator using return by reference
- All three programs create a class called `safearay`, whose only member data is an array of 100 int values, and all three check to ensure that all array accesses are within bounds.

Separate `get()` and `put()` Functions

- The first program provides two functions to access the array elements:
- `putel()` to insert a value into the array, and
- `getel()` to find the value of an array element.
- Both functions check the value of the index number supplied to ensure it's not out of bounds; that is, less than 0 or larger than the array size (minus 1).
- See "arrover1.cpp"
- "`exit(0)` indicates successful program termination to the OS
- "`exit(1)` indicates unsuccessful program termination to the OS
- The data is inserted into the safe array with the `putel()` member function, and then displayed with `getel()`.
- This implements a safe array; we'll receive an error message if we attempt to use an out-of-bounds index.
- However, the format is a bit crude.

Single `access()` Function Returning by Reference

- We can use the same member function both to insert data into the safe array and to read it out.
- The secret is to return the value from the function by reference.
- This means we can place the function on the left side of the equal sign, and the value on the right side will be assigned to the variable returned by the function.
- See ”arrover2.cpp”
- The statement

  ```
  sa1.access(j) = j*10; // *left* side of
  equal sign
  ```

  causes the value j*10 to be placed in `arr[j]`, the return value of the function.
- It’s slightly more convenient to use the same function for input and output of the safe array than it is to use separate functions; there’s one less name to remember.

Overloaded [] Operator Returning by Reference

- To access the safe array using the same subscript ([]) operator that's used for normal C++ arrays, we overload the subscript operator in the `safearay` class.
- However, since this operator is commonly used on the left side of the equal sign, this overloaded function must return by reference.
- See "arrover3.cpp"
- In this program we can use the natural subscript expressions
  `sa1[j] = j*10;`
  and
  `temp = sa1[j];`
  for input and output to the safe array.