16 - Overloading

March 13, 2023

COMP2404

Darryl Hill

Contents

Function Overloading

Overloading

- give something multiple meanings or definitions

Overloading functions

- functions have the same name BUT
- have different parameters (type, order, number)
- resulting in a different **function signature**
- can be global or member functions

Function Overloading

Characteristics

- Each function must have unique signature
  - The compiler must be able to tell them apart
- Signature includes function name and parameter type and order
  - Different return type is not sufficient

Convention

- Should be used for a functionally related task

Function Overloading

How it works:

- When compiling to assembly, the compiler ***mangles*** every function name
- Each function name becomes
  - name
  - ordered parameter types
- This makes each function unique
- Compiler chooses which function to call based on this (unique) signature

coding example <p1>

- compile with -S for assembly and to see the mangled function name

Operator Overloading

Operator Overloading

1) Purpose
2) Cascading
3) Operators as functions

Common Overloaded Operators

- Stream insertion and extraction
- Unary and binary operators
- Operators on collection classes
- Increment and decrement operators

Operators

Operators are commonly associated with math expressions

- `3 + 4` $\rightarrow$ `7`

Or logic expressions

- `3 < 4` $\rightarrow$ `True`

Or (in C++) stream operations

- `cout << "Hi there!";`

Operators in C++ **are functions** with a special syntax

- Convenience
- `3 + 4` is more natural than `plus(3, 4)`
- Compiler reads `3 + 4` then looks for a special function.
  - An operator is a function with "syntactic sugar"

Operator Overloading

What is **Operator Overloading**?

- We can specify how operators work on ***user-defined types***
  - The operator "syntactic sugar" can be applied to the class of our choice

For example, we often want to compare objects:

```
Student mat, joe;
    if (joe < matt)
        cout<<"mat is better"<<endl;
```

What does the < mean in this context? We get to decide.

coding example <p2>

Coding Example p2 Notes

- Time class with hours, minutes, seconds
- `wakeup == lunch` is changed by the compiler to
  - `wakeup.operator==(lunch)`
- The above function is what the compiler will look for within the Time class
- The `==` can be substitued with `!=, >=`, etc
- Really only need `==` and `<`, since all the rest can be made from these
- Also note, when using `this` we need to dereference: `*this`

Purpose

Why bother?

- Language consistency.
  - Users of your class will expect it.
- Code readability.
  - We all remember math, right?

Consistency is important for polymorphism.

- Operators provide a general, common interface.

Collections will by default use `<,>,==, <=, >=` for objects it stores.

- Sorting a `vector` requires `<` to be overloaded.
- Finding something in a vector requires `==`
- etc.

Example: string

Example: the `string` class provides

- assignment operator `=`
- relational operators `<, >, !=`, etc
- subscript operator `[ ]` - returns a character
- stream insertion and extraction operators `<<,>>`
- etc.

Implicitly Overload

We get some implicitly overloaded (default) operators made for us:

- Address of operator `&`
- Sequencing operator `,`
- Assignment operator `=`
  - Similar to **default copy constructor**, does a **shallow copy** by default.
  - Can be overloaded to do a **deep copy**.

  - Beware! If your object has a **constant member variable**, the default assignment operator is **deleted**.
  - A constant variable cannot have its value re-assigned, so the compiler goes ¯_(ツ)_/¯
  - You may still write your own `=` function.

Explicitly Overloaded

Explicitly overloaded - some operators don't have default functions, however

- Class developer may decide to implement them.
- Almost all operators can be overloaded.
- Each operator must be overloaded separately
  - The compiler will not combine `==` and `<` into `<=` for us

Overloading Operators

Each operator is actually syntactic sugar for a function.

These functions have particular signatures.

- Keyword `operator` followed by the operator symbol
- You can call them using the function name explicitly(though this defeats the purpose)
  - `myname.operator==(yourname)` and `myname == yourname` are the same thing to the compiler.
- The LHS operand is the calling object (also a parameter, with the `this` keyword)
- The RHS operand is the second parameter.
- We are only allowed one explicit parameter in this case.

Operator Overloading Limitations

- Operators can be overloaded as
  - A global function
  - A member function
  - But NOT both
- Cannot change operators for primitives
  - `int, float, double`, etc
- Cannot create new operators.
  - The compiler will not recognize them
- Cannot change the operator arity.
  - You must use the conventional syntax and arity.
- Cannot change precedence or associativity.
- Some operators cannot be overloaded
  - dot, scope resolution, conditional, a few more

Operator Return Type

One operator convention is to enable ***cascading*** where possible.

```
a = b = c = d = 0
```

- Operator should return a value (not an output parameter).
- Here we want a return value. General rule:
  - Should behave the same as `int` operations.
- Should not return void, should return the same type being operated on.

Cascading

The `this` keyword is an object's pointer to itself

- Passed as an implicit parameter to all member functions
  - Side note: in Python it is named `self` and it is explicitly passed as first parameter
- This does not include static functions of course
  - static functions are not associated with an object, there is no `this`

- `this` can be used explicitly or implicitly

Cascading

Cascading

- chaining together member functions in a single statement

How does it work?

- member function returns LHS operand object
  - perhaps using `this`
- next member function operates on the returned object
- Example for a `Time` object we would expect to be able to write:
  `time.setHours().setMinutes().setSeconds()`

coding example <p3>

Operators Overloading as a Member Function

When overloading an operator as a **member** function:

- the LHS operand is the `*this` object

For binary operators:

- the RHS is passed in as a parameter by ***reference***

When possible, implement operators as member functions.

It is ***good*** encapsulation.

Operators Overloading as a Global Function

Sometimes an operator cannot be a member function

- LHS is a primitive
- We cannot change LHS class since it is not OUR class

When overloading an operator as a **global** function:

- all the operands are parameters. Example:

```
Student& s1; Student& s2;
```

`s1 == s2` corresponds to the global function:

```
bool operator==(Student& s1, Student& s2);
```

Both LHS and RHS are given explicitly as parameters.

Operators Overloading as a Global Function

For convenience, the global function may be a ***friend*** of the operand class

- Might be necessary for access.
- This is the only permitted use of friendship in this course.

By convention the global function implementation goes into the .cc file.

coding example <p4>

Operators as Functions

Some operators can **only** be overloaded as member functions

- typecast `( )`
- subscript `[ ]`
- arrow `->`

Always enable cascading where appropriate.

Operators as Functions

Some operators can only be overloaded as ***global*** functions.

- Operators for commutativity
  - If we want to compare `Time` and `int` using `==`, then both versions should work:

```
Time& == int
int == Time&
```

  - Each of these must be a separate function.
  - Must be a global function if
    - The LHS operand is primitive, or
    - the LHS operand is a class that we cannot modify.

- Stream insertion `<<`
- Stream extraction `>>`
  - We cannot add a member function to these classes.

Always enable cascading where appropriate!

Unary Operators

If a unary operator is overloaded as a global function

- it takes one parameter - reference to the operand

If overloaded as a member function

- it takes no parameters
- `this` is implicitly the operand

Binary Operators

If a binary operator is overloaded as a global function

- it takes two parameters
- references to both operands

If a binary operator is overloaded as a member function

- it takes one parameter - RHS operand

Stream Operators

Stream operators are already overloaded for primitives.

- Since LHS is a stream object, must be overloaded as global functions.
- For cascading must return a stream object.

Example:

- `cout << theTime;`

Compiler first looks for `cout.operator<<(Time&)` in `ostream` class

- Since `Time` is our class, this function does not exist

Next the compiler will look for a matching global function.

coding example <p5>

Stream Operators

Stream insertion operator `<<` takes two operands.

- LHS is `cout`
  - an `ostream` object
    - passed in as a reference
  - RHS is object to be output
    - also a reference

```
ostream& operator<<(ostream&, const Time&);
```

Must be a `friend` function of our class if we wish to access private members.

Stream Operators

Stream extraction operator `>>` takes two operands.

- LHS is `cin`.
  - An `istream` object.
  - Passed in as a reference.
- RHS is object to be input.
  - Also a reference.
  - should NOT be `const`

```
istream& operator>>(istream&, Time&);
```

Must be a `friend` function of our class to access private members.

Polymorphic Stream Operations

What if we overload `<<` for an inherited class?

- `operator<<` is a global function
  - can only implement polymorphism on member functions

Solution 1: make a separate function for each derived class

Solution 2: use an ***abstraction layer***

```
virtual void print(ostream& os) const{//print something}

ostream& operator<<(ostream& ost, const Animal& t){
    t.print(ost);
    return ost;
}
```

example <p6>

Binary Operators - Return Values

Recall cascading - we want to return an object to continue operations.

- We have seen cascading where we return a current reference.

```
return *this;
```

- However, there are instances where this is not appropriate.
  - We want operators to work as they work on `ints`. Consider:

```
int a = 2, b = 5;
int c = a + b;
int d = c += a;
```

- What should the `+` and `+=` operators return?

coding example <p7>

Coding Example p7 Notes

- `+=` operator on the `Time` object
  - returns a reference to the changed `Time` object
- When we implement the addition operator
  - `Time operator+(int time);`
  - note we do not return `Time` by reference, we return `Time` by ***value***.
  - A temporary `Time` object with the modified `Time` is made and returned.
  - We do not want to change the current object.
- Notice we don't return an `int` - what if we want to call `(time+80).print();`?

Operators on Collection Classes

Collection classes have many operations that can naturally be represented using operators.

- `[ ]` for accessing elements
- `+=, +, -=, -` for adding and removing elements
- `==, !=` for comparing collections

If you want sorted collections, you may want the data to override `<,>,==`

- By default collections will use these operators to sort.

coding example <p8>

Coding Example p8 Notes

In our collection class `BookArray` we get a default `=` operator.

- Does a member-wise assignment - a shallow copy.

```
BookArray b1; BookArray b2;
b1 = b2;
```

- Now both `BookArrays` will point to the same array in memory.
- This is probably not what we want.

If we want `=` to do a deep copy we must override it.

Increment and Decrement Operators

For example: `++i;` or `i--;`

Lots of variation

- increment or decrement
- prefix or postfix
- global or member

Each has its own syntax!

Increment and Decrement Operators

Prefix ++ or --

- modifies the operand
- returns a reference to the UPDATED object

- if implemented as a member function
  - takes no parameters

- if global function
  - takes one parameter - object to be modified

coding example <p8>

Increment and Decrement

Problem:

- How do we distinguish prefix and postfix?
- Both have the same signature, `operator++` and `operator--`

Solution:

- An `int` parameter is introduced.
  - Serves no purpose
    - though we may use it as a variable
  - Purpose is to simply distinguish pre- and postfix operators
  - lol

Postfix is more complicated.

Postfix ++ and - -

Think carefully:

- Must make a copy of the object before it is modified.
  - This is what is returned - not a reference.
  - We will need a copy constructor.

Observe

- creation of a temp object makes it slower
- always use prefix where possible
  - particularly with objects

<p8> again

Postfix ++ and - -

If global function

- takes 2 parameters
  - reference to the operand
  - dummy `int`
    - really is a "my assignment is due in 30 minutes!" solution

If member function

- takes one parameter - dummy `int`

Smart Pointers

Memory management is a difficult part of C++

In modern C++ there helpful tools, like the ***smart pointer***.

Operator overloading allows a class to intercept pointer operations.

We can overload:

$->$, $*$, $\&$

so we can make a ***class*** that acts like a ***pointer***.

programming example <p10>