Operator overloading and *this

- we can declare new meanings for an operator in the context of a given class, called overloading
- e.g. for lists perhaps overload = and + to allow L = L1 + L2
- we can only create new meanings for existing operators, we cannot create new operators
- we cannot change the precedence or associativity of an operator
- we cannot change the number of arguments an operator expects
- we cannot overload . or :: or ?:

Ways to overload

- we can overload operators using a class method
  - since it's a method it has access to the private content
  - this is the only way to overload assignment operators
- we can overload operators using a friend function
  - since it's a friend it has access to the private content
  - often used when operand is on right of operator, e.g. with << operator, used like “cout << X;” where X is our object
- we can overload operators using a “normal” function
  - no private access, so needs sufficient public fields/methods accessible to do its job

Example: stack class with +=

- take a stack class, implemented in linked list fashion
- overload = operator so “s1 = s2;” makes s1 a copy of s2
- = returns the value it assigns so works with x = y = z;

```
class stack {
   private:
      struct node { double val; node* next; } *tos;
   public:
      // returns the revised stack, i.e. the value assigned
      // s1 = s2;  ... parameter rhs refers to s2
      // pass s2 by ref for efficiency but as const so we don't alter it
      stack& operator=(const stack& rhs);
      ...
};
```

Stack = implementation

- copy s2 to s1, node by node
- should probably delete any old s1 content (not shown here)
- will discuss the *this shortly

```
stack& stack::operator=(const stack& rhs)
{
  node* curr = rhs.tos;
  tos = NULL;
  node* currNew = tos;
  while (curr) {
    string k = curr->key;
    string v = curr->value;
    curr = curr->next;
    node *n = new node;
    n->key = k;
    n->value = v;
    n->next = NULL;
    if (currNew == NULL) {
      tos = n;
      currNew = tos;
    } else {
      currNew->next = n;
      currNew = n;
    }
  }
  return *this;
}
```

“this” pointer

- whenever a class method is called on an object it is passed a hidden parameter named “this”
- “this” is actually a pointer to the object itself

```
class example
{
   private:
       int i, j;
   public:
       void set(int ival, int jval);
};

void example::set(int ival, int jval)
{
    i = ival; j = jval;
}
```

```
// compiler inserts an extra hidden pointer parameter
void example::set(example *this, int ival, int jval)
{
    i = ival; j = jval;
}

int main()
{
    example e;
    e.set(10,20);
    // compiled call is more like
    // example::set(&e, 10, 20);
}
```

use of this and *this

- within a method we can use “this” as a pointer to the actual object
- comes up most frequently when we either want to
  - return a pointer to the object, i.e. `return this;`
  - or return the object itself, i.e. `return *this;`

Using friend function, unary - op

- suppose we want - to act as negation,
  - e.g. -x; // negates value inside x

```
class simpleData {
   private:
      long data;
   public:
      simpleData(int d = 0) { data = d; }
      // will use a friend function to flip sign of data
      friend void operator-(simpleData& rhs);
};

int main() {
    simpleData x(5);
    -x;
    // x.data is now -5
}
```

```
void operator(simpleData& rhs)
{
    // can access private fields since
    // we're a friend of simpleData
    rhs.data = -rhs.data;
}
```

Using friend function, binary << op

- suppose we want to overload <<, e.g. for cout << x << y;
- on the left of << we have the output stream (type ostream) that we're writing to, on the right of << we have the output data
- << needs to return the updated output stream value
- will use a friend function and our simpleData class again

```
class simpleData {
   ... same as previous slide ...
   // allows for use of chained <<, e.g. cout << “x is “ << x << end;
   // std::ostream available through iostream library
   friend ostream& operator<<(ostream& outstr, const simpleData& rhs);
};
```

Overloaded << continued

```
ostream& operator<<(ostream& outstr, const simpleData& rhs)
{
    // first do the actual output, using the given output stream
    outstr << rhs.data;

    // then return the updated output stream
    return outstr;
}
```