Introduction to C++

Operator Overloading

Topic #6

- **Intro to Operator Overloading**
  - Copy Constructors, Issues of Memberwise Copy
  - Constant Objects and Constant Member Functions
  - Friend Functions
  - When to define operators as Members vs. Non-Members
  - Lvalue vs. Rvalue Expressions
  - Return by Value vs. Return by Reference

- **Designing Effective User Defined Data Types**
  - How to design User Defined Types that behave as expected
  - Practical Rules for Operator Overloading

Copy Constructors

- **Shallow Copy:**
  - The data members of one object are copied into the data members of another object without taking any dynamic memory pointed to by those data members into consideration. ("memberwise copy")
- **Deep Copy:**
  - Any dynamic memory pointed to by the data members is duplicated and the contents of that memory is copied (via copy constructors and assignment operators -- when overloaded)

Copy Constructors

- In every class, the compiler automatically supplies both a copy constructor and an assignment operator if we don't explicitly provide them.
- Both of these member functions perform copy operations by performing a memberwise copy from one object to another.
- In situations where pointers are not members of a class, memberwise copy is an adequate operation for copying objects.
- However, it is not adequate when data members point to memory dynamically allocated within the class.

Copy Constructors

- Problems occur with shallow copying when we:
  - initialize an object with the value of another object: name s1; name s2(s1);
  - pass an object by value to a function or when we return by value:

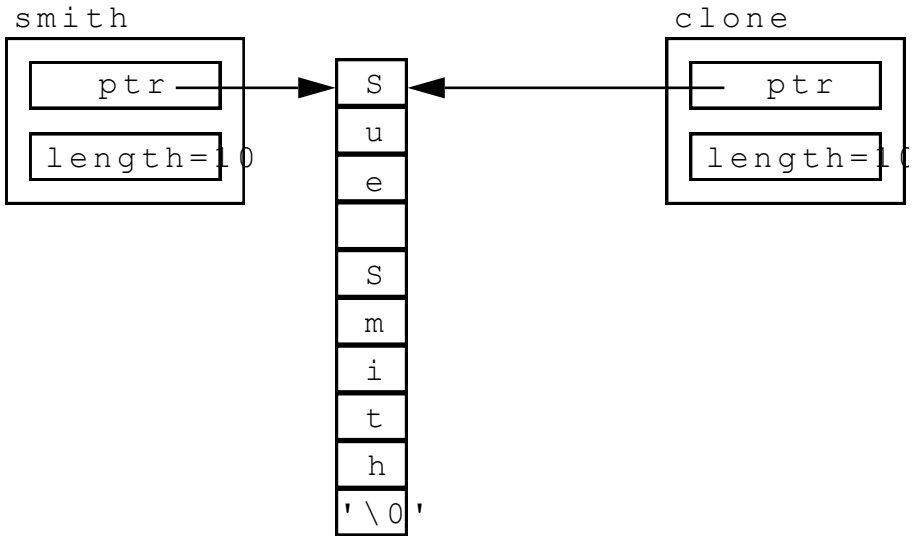
    name function_proto (name)
  - assign one object to another:

    s1 = s2;

Copy Constructors

- If name had a dynamically allocated array of characters (i.e., one of the data members is a pointer to a char),
  - ↗ the following shallow copy is disastrous!

```
name smith("Sue Smith"); //one arg constructor used
name clone(smith);       //default copy constructor used
```



Copy Constructors

- To resolve the pass by value and the initialization issues, we must write a copy constructor whenever dynamic member is allocated on an object-by-object basis.
- They have the form:

```
class_name(const class_name &class_object);
```

- Notice the name of the "function" is the same name as the class, and has no return type
- The argument's data type is that of the class, passed as a constant reference (think about what would happen if this was passed by value?!)

Copy Constructors

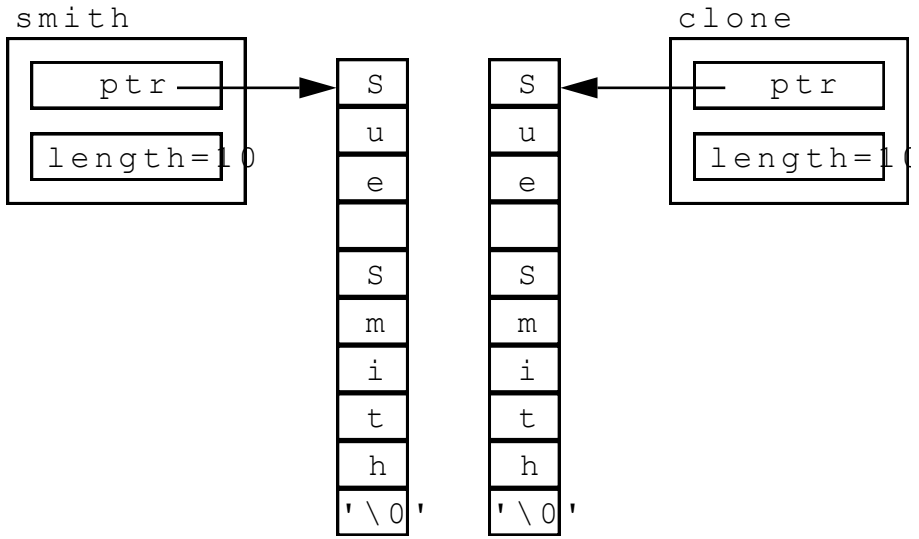
```
//name.h interface
class name {
  public:
    name(char* = "");        //default constructor
    name(const name &);      //copy constructor
    ~name();                 //destructor
    name &operator=(name &); //assignment op
  private:
    char* ptr;  //pointer to name
    int length; //length of name including nul char
};

#include "name.h"                              //name.c implementation
name::name(char* name_ptr) {   //constructor
  length = strlen(name_ptr);   //get name length
  ptr = new char[length+1];    //dynamically allocate
  strcpy(ptr, name_ptr);       //copy name into new space
}
name::name(const name &obj) {  //copy constructor
  length = obj.length;         //get length
  ptr = new char[length+1];    //dynamically allocate
  strcpy(ptr, obj.ptr);        //copy name into new space
}
```

Copy Constructors

- Now, when we use the following constructors for initialization, the two objects no longer share memory but have their own allocated

```
name smith("Sue Smith"); //one arg constructor used
name clone(smith);       //default copy constructor used
```



Copy Constructors

- Copy constructors are also used whenever passing an object of a class by value: (get_name returns a ptr to a char for the current object)

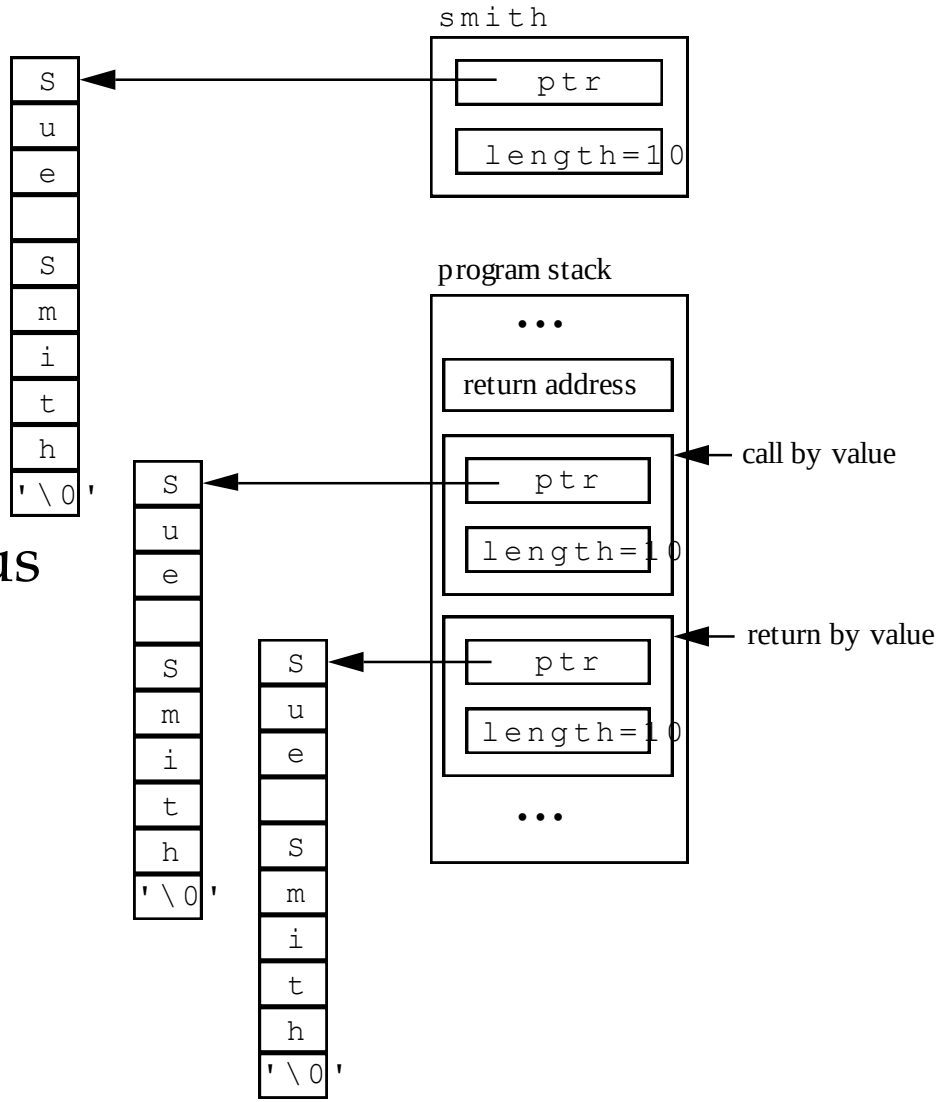
```
int main() {
  name smith("Sue Smith"); //constructor with arg used

  //call function by value & display from object returned
  cout <<function(smith).get_name() <<endl;
  return (0);
}

name function(name obj) {
  cout <<obj.get_name() <<endl;
  return (obj);
}
```

Copy Constructors

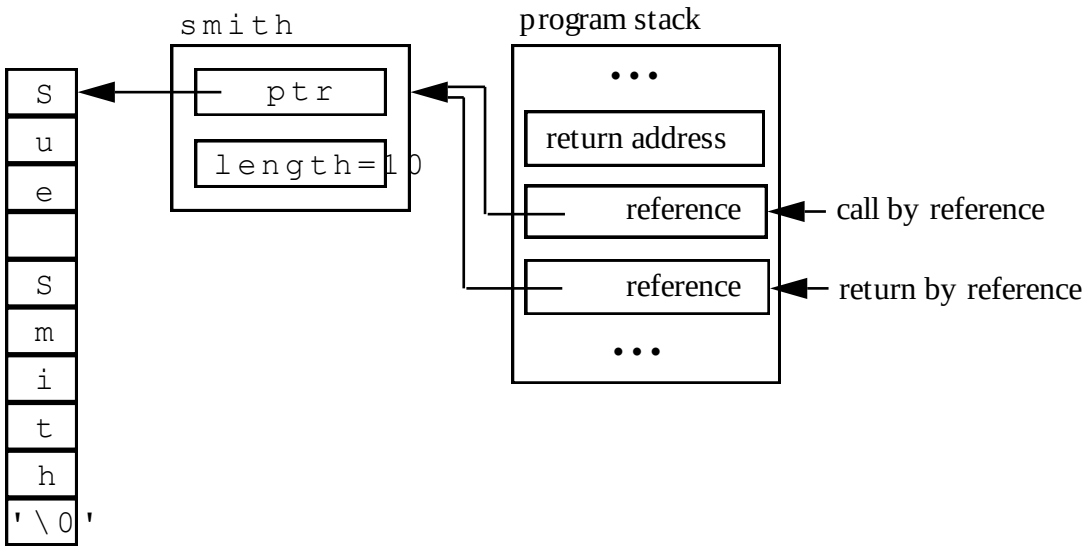
- Using a copy constructor avoids objects "sharing" memory -- but causes this behavior
- This should convince us to avoid pass by value whenever possible -- when passing or returning objects of a class!



Copy Constructors

- Using the reference operator instead, we change the function to be: (the function call remains the same)

```
name &function(name &obj) {
  cout <<obj.get_name() <<endl;
  return (obj);
}
```



Introduction to C++

What is..Operator Overloading

- **Operator Overloading:**
  - Allows us to define the behavior of operators when applied to objects of a class
  - Examine what operators make sense for a "new data type" we are creating (think about data abstraction from last lecture) and implement those that make sense as operators:
  - input_data is replaced by >>
  - display is replaced by <<
  - assign or copy is replaced by =

Operator Overloading

- Operator Overloading does not allow us to alter the meaning of operators when applied to built-in types
  - one of the operands must be an object of a class

- Operator Overloading does not allow us to define new operator symbols
  - we overload those provided for in the language to have meaning for a new type of data...and there are very specific rules!

Operator Overloading

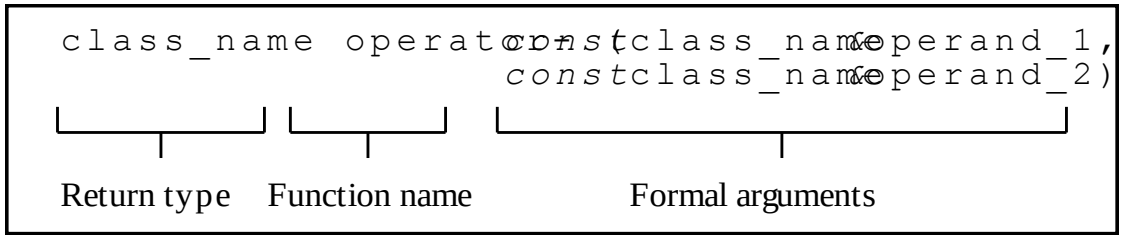
- It is similar to overloading functions
  - ↗ except the function name is replaced by the keyword operator followed by the operator's symbol
  - ↗ the return type represents the type of the residual value resulting from the operation
    - – rvalue? -lvalue?
    - – allowing for "chaining" of operations
  - ↗ the arguments represent the 1 or 2 operands expected by the operator

Operator Overloading

- We cannot change the....
  - number of operands an operator expects
  - precedence and associativity of operators
  - or use default arguments with operators
- We should not change...
  - the meaning of the operator
    (+ does not mean subtraction!)
  - the nature of the operator (3+4 == 4+3)
  - the data types and residual value expected
  - whether it is an rvalued or lvalued result
  - provide consistent definitions (if + is overloaded, then += should also be)

Understanding the Syntax

- This declaration allows us to apply the subtraction operator to two objects of the same class and returns an object of that class as an rvalue.
- The italics represent my recommendations, if followed, result in behavior that more closely matches that of the built-in types.
- Since the predefined behavior of the subtraction operator does not modify its two operands, the formal arguments of the operator- function should be specified either as constant references or passed by value.



Operator Overloading

- An overloaded operator's operands are defined the same as arguments are defined for functions.
- The arguments represent the operator's operands.
- Unary operators have a single argument and binary operators have two arguments.
- When an operator is used, the operands become the actual arguments of the "function call".
- Therefore, the formal arguments must match the data type(s) expected as operands or a conversion to those types must exist.
- I recommend that unary operators always be overloaded as members, since the first argument must be an object of a class (except....as discussed in class)

Operator Overloading

- The return type of overloaded operators is also defined the same as it is for overloaded functions.
- The value returned from an overloaded operator is the residual value of the expression containing that operator and its operands.
- It is extremely important that we pay close attention to the type and value returned.
- It is the returned value that allows an operator to be used within a larger expression.
- It allows the result of some operation to become the operand for another operator.
- A return type of void would render an operator useless when used within an expression. (I suggest that we never have an operator return void!)

Operator Overloading

- Binary operators have either a single argument if they are overloaded as members (the first operand corresponds to the implicit this pointer and is therefore an object of the class in which it is defined)
- Or, binary operators have two operands if they are overloaded as non-members
  - ↗ (where there is no implicit first operand)
- In this latter case, it is typical to declare the operators as friends of the class(es) they apply to -- so that they can have access privileges to the private/protected data members without going thru the public client interface.

As Non-members

- Overloading operators as non-member functions is like defining regular C++ functions.
- Since they are not part of a class' definition, they can only access the public members. Because of this, non-member overloaded operators are often declared to be friends of the class.
- When we overload operators as non-member functions, all operands must be explicitly specified as formal arguments.
- For binary operators, either the first or the second must be an object of a class; the other operand can be any type.

Operator Overloading

- All arithmetic, bitwise, relational, equality, logical, and compound assignment operators can be overloaded.
- In addition, the address-of, dereference, increment, decrement, and comma operators can be overloaded.
- Operators that cannot be overloaded include:

::	scope resolution operator
.	direct member access operator
.*	direct pointer to member access operator
?:	conditional operator
sizeof	size of object operator

- Operators that must be overloaded as members:

=	assignment operator
[]	subscript operator
()	function call operator
->	indirect member access operator
->*	indirect pointer to member access operator

Guidelines:

- Determine if any of the class operations should be implemented as overloaded operators: does an operator exists that performs behavior similar in nature to our operations? If so, consider overloading those operators. If not, use member functions.
- Consider what data types are allowed as operands, what conversions can be applied to the operands, whether or not the operands are modified by the operation that takes place, what data type is returned as the residual value, and whether the residual value is an rvalue (an object returned by value), a non-modifiable lvalue (a const reference to an object), or a modifiable lvalue (a reference to an object).

Guidelines:

- If the first operand is not an object of the class in all usages: (e.g., +)
  - ↗ overload it as a friend non-member
- As a non-member, if the operands are not modified by the operator (and are objects of a class)
  - ↗ the arguments should be const references
- If the first operand is always an object of the class: (+=)
  - ↗ overload it as a member
- As a member, if the operator does not modify the current object (i.e., data members are not modified)
  - ↗ overload it as a const member

Guidelines:

- If the operator results in an lvalued expression
  - the return type should be returned by referenced
  - for example -= results in an lvalued expression
- If the operator results in an rvalued expression
  - the return type should be returned by reference if possible but usually we are "stuck" returning by value (causing the copy constructor to be invoked when we use these operators.....)
  - for example - results in an rvalued expression

Guidelines: (example)

- As a member, operator - could be overloaded as:

```
class class_name {
  public:
    class_name operator+(const
};
```

Return type

The operand is not modified

- As a non-member, operator - resembles:

```
class class_name {
  public:
    class_name operator+(const class_name& op_2)const;
};
```

Return type

Second operand is not modified

Second operand

First operand is not modified

Efficiency Considerations

- Temporary objects are often created by implicit type conversions or when arguments are returned by value.
- When an operator and its operands are evaluated, an rvalue is often created.
- That rvalue is a temporary on the stack that can be used within a larger expression. The lifetime of the temporary is from the time it is created until the end of the statement in which it is used.
- While the use of temporaries is necessary to protect the original contents of the operator's operands, it does require additional memory and extra (and sometimes redundant) copy operations.

Efficiency Considerations

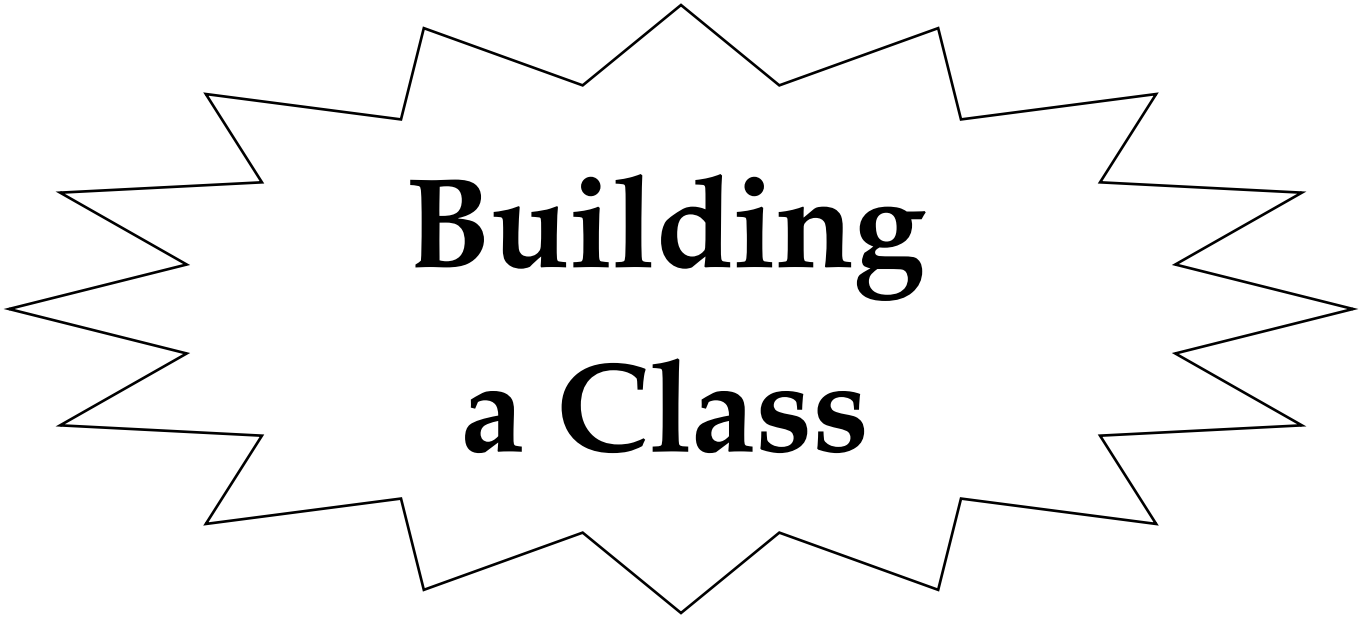
- Whenever we overload the arithmetic or bitwise operators, we should also overload the corresponding compound assignment operators.
- When we do, it is tempting to reuse the overloaded arithmetic or bitwise operators to implement the compound assignment operator.

```
//assumes the + operator is overloaded for string class
inline string &string::operator+=(char * s) {
  *this = *this + s; //concatenate a literal
  return (*this);    //return modified current object
}
```

Don't Program this Way!

Efficiency Considerations

- While the code on the previous slide looks clean and simple, it has serious performance drawbacks.
- This is because it creates a temporary string object from the argument, creates a second temporary object as a result of the concatenation, and then uses the copy constructor to copy that temporary back into the original object (*this).
- If the object was a large object, this simple operation could end up being very expensive!


Building
a Class

String Class Example

- Let's build a complete class using operator overloading to demonstrate the rules and guidelines discussed
- We will re-examine this example again next lecture when discussing user defined type conversions
- The operations that make sense include:

  = for straight assignment of strings and char *'s

  >> and << for insertion and extraction

  + and += for concatenation of strings and char *'s

  <, <=, >, >=, !=, == for comparison of strings

  [] for accessing a particular character in a string

Overloading = Operators

- Whenever there is dynamic memory allocated on an object-by-object basis in a class, we should overload the assignment operator for the same reasons that require the copy constructor
- The assignment operator must be overloaded as a member, and it doesn't modify the second operand (so if it is an object of a class -- it should be a const ref.)
- The assignment operator can be chained, so it should return an lvalued object, by reference
- It modifies the current object, so it cannot be a const member function

Overloading = Operator

```
class string {
  public:
    string(): str(0), len(0) {};  //constructor
    string(const string &);     //copy constructor
    ~string();                                  //destructor
    string & operator = (const string & );  //assignment
    •••
  private:
    char * str;
    int len;
};

string & operator = (const string & s2) {
  if (this == &s2)     //check for self assignment
    return *this;
  if (str)             //current object has a value
    delete [] str;     //deallocate any dynamic memory
  str = new char [s2.len+1];
  strcpy(str,s2.str);
  len = s2.len;
  return *this;
}
```

Overloading <<, >> Operators

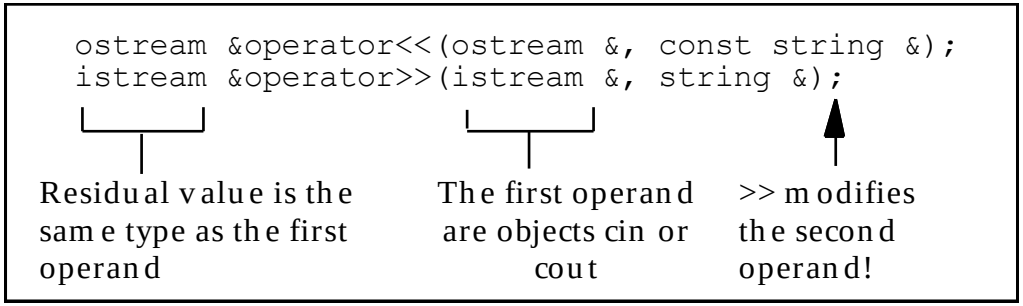
- We overload the << and >> operators for insertion into the output stream and extraction from the input stream.
- The iostream library overloads these operators for the built-in data types, but is not equipped to handle new data types that we create. Therefore, in order for extraction and insertion operators to be used with objects of our classes, we must overload these operators ourselves.
- The extraction and insertion operators must be overloaded as non-members because the first operand is an object of type istream or ostream and not an object of one of our classes.

Overloading <<, >> Operators

- We know from examining how these operators behave on built-in types that extraction will modify the second operand but the insertion operator will not.
- Therefore, the extraction operation should declare the second operand to be a reference.
- The insertion operator should specify the second operator to be a constant reference.
- The return value should be a reference to the object (istream or ostream) that invoked the operator for chaining.

  cin >> str >>i;

  cout << str <<i;



Overloading >>, << Operators

- It is tempting when overloading these operators to include prompts and formatting.
- This should be avoided. Just imagine how awkward our programs would be if every time we read an int or a float the extraction operator would first display a prompt. It would be impossible for the prompt to be meaningful to all possible applications.
- Plus, what if the input was redirected from a file? Instead, the extraction operator should perform input consistent with the built-in types.
- When we read any type of data, prompts only occur if we explicitly write one out (e.g., cout <<"Please enter...").

Overloading >>, << Operators

```
class string {
  public:
    friend istream & operator >> (istream &, string &);
    friend ostream & operator << (ostream &, const string&);
    •••
  private:
    char * str;
    int len;
};

istream & operator >> (istream &in, string &s) {
  char temp[100];
  in >>temp;   //or, should this could be in.get?!
  s.len = strlen(temp);
  s.str = new char[s.len+1];
  strcpy(s.str, temp);
  return in;
}

ostream & operator << (ostream &o, const string& s){
  o << s.str;   //notice no additional whitespace sent....
  return o;
```

Overloading +, +=Operators

- If the + operator is overloaded, we should also overload the += operator
- The + operator can take either a string or a char * as the first or second operands, so we will overload it as a non-member friend and support the following:
  - string + char *, char * + string, string + string
- For the += operator, the first operand must be a string object, so we will overload it as a member
- The + operator results in a string as an rvalue temp
- The += operator results in a string as an lvalue
- The + operator doesn't modify either operand, so string object should be passed as constant references

Overloading +, += Operators

```
class string {
  public:
    explicit string (char *);  //another constructor
    friend string operator + (const string &, char *);
    friend string operator + (char *, const string &);
    friend string operator + (const string&, const string&);
    string & operator += (const string &);
    string & operator += (char *);
    •••
};
string operator + (const string &s, char *lit) {
    char * temp = new char[s.len+strlen(lit)+1];
    strcpy(temp, s.str);
    strcat(temp, lit);
    return string(temp);
}
```

- This approach eliminates the creation of a temporary string "object" in the + function by explicitly using the constructor to create the object as part of the return statement. When this can be done, it saves the cost of copying the object to the stack at return time.

Overloading +, += Operators

```
class string {
  public:
    explicit string (char *);  //another constructor
    friend string operator + (const string &, char *);
    friend string operator + (char *, const string &);
    friend string operator + (const string&, const string&);
    string & operator += (const string &);
    string & operator += (char *);
    •••
};
string operator + (const string &s,const string &s2) {
    char * temp = new char[s.len+s2.len+1];
    strcpy(temp, s.str);
    strcat(temp, s2.str);
    return string(temp);  //makes a temporary object
}
string & string::operator += (const string & s2) {
    len += s2.len;
    char * temp = new char[len+s2.len+1];
    strcpy(temp, str);
    strcat(temp, s2.str);
    str = temp;    //copy over the pointer
    return *this;  //just copying an address
}
```

Overloading +, += Operators

- Alternative implementations, not as efficient:

```
string operator + (const string &s, char *lit) {
    string temp;
    temp.len = s.len+strlen(lit);
    temp.str = new char[temp.len+1];
    strcpy(temp.str, s.str);
    strcat(temp.str, lit);
    return temp;
}
```

Don't do the following....

```
string & string::operator += (const string & s2) {

    return *this=*this+s2;  //Extra unnecessary deep copies
}
```

Overloading +, += Operators

- If the + operator was overloaded as a member, the first operand would have to be an object of the class and we should define the member as a const because it doesn't modify the current object (i.e., the first operand is not modified by this operator!

```
string string::operator + (char *lit)const { //1 argument
    char * temp = new char[len+strlen(lit)+1];
    strcpy(temp, str);
    strcat(temp, lit);
    return string(temp);  //makes a temporary object
}
```

- Defining member functions as const allows the operator to be used with a constant object as the first operand. Otherwise, using constant objects would not be allowable resulting in a syntax error.

Relational/Equality Operators

- The next set of operators we will examine are the relational and equality operators
- These should be overloaded as non-members as either the first or second operands could be a non-class object: string < literal, literal < string, string < string
- Neither operand is modified, so all class objects should be passed as constant references.
- The residual value should be a bool, however an int will also suffice, returned by value.
- If overloaded as a member -- make sure to specify them as a const member, for the same reasons as discussed earlier.

Relational/Equality Operators

```
class string {
  public:
    friend bool operator < (const string &, char *);
    friend bool operator < (char *, const string &);
    friend bool operator < (const string &, const string &);

    friend bool operator <= (const string &, char *);
    friend bool operator <= (char *, const string &);
    friend bool operator <= (const string &,const string &);

    friend bool operator > (const string &, char *);
    friend bool operator > (char *, const string &);
    friend bool operator > (const string &, const string &);

    friend bool operator >= (const string &, char *);
    friend bool operator >= (char *, const string &);
    friend bool operator >= (const string &,const string &);

    friend bool operator != (const string &, char *);
    friend bool operator != (char *, const string &);
    friend bool operator != (const string &,const string &);

    friend bool operator == (const string &, char *);
    friend bool operator == (char *, const string &);
    friend bool operator == (const string &,const string &);
```

Relational/Equality Operators

```
bool operator < (const string & s1, char * lit) {
  return (strcmp(s1.str, lit) < 0);
}

bool operator < (const string & s1, const string & s2) {
  return (strcmp(s1.str, s2.str) < 0);
}
```

Then, you could implement the > either of the following:

```
bool operator >= (char * lit, const string & s1) {
  return (strcmp(lit, s1.str) >= 0);
}
```

or,

```
bool operator >= (char * lit, const string & s1) {
  return (s1 < lit);
}
```

Which is better?

Overloading [] Operator

- The subscript operator should be overloaded as a member; the first operand must be an object of the class
- To be consistent, the second operand should be an integer index. Passed by value as it isn't changed by the operator.
- Since the first operand is not modified (i.e., the current object is not modified), it should be specified as a constant member -- although exceptions are common.
- The residual value should be the data type of the "element" of the "array" being indexed, by reference.
- The residual value is an lvalue -- not an rvalue!

Overloading [] Operator

```
class string {
  public:
    char & operator [] (int) const;
    •••
};

char & string::operator [] (int index) const {
  return str[index];
}
```

- Consider changing this to add
  - bounds checking
  - provide access to "temporary" memory to ensure the "private" nature of str's memory.

Function Call Operator

- Another operator that is interesting to discuss is the (), function call operator.
- This operator is the only operator we can overload with as many arguments as we want. We are not limited to 1, 2, 3, etc. In fact, the function call operator may be overloaded several times within the same scope with a different number (and/or type) of arguments.
- It is useful for accessing elements from a multi-dimensional array: matrix (row, col) where the [] operator cannot help out as it takes 2 operands always, never 3!

Function Call Operator

- The function call operator must be a member as the first operand is always an object of the class.
- The data type, whether or not operands are modified, whether or not it is a const member, and the data type of the residual value all depend upon its application. Again, it is the only operator that has this type of wildcard flexibility!
- return_type class_type::operator () (argument list);
- For a matrix of floats:

```
float & matrix::operator () (int row, int col) const;
```

Increment and Decrement

- Two other operators that are useful are the increment and decrement operators (++ and --).
- Remember these operators can be used in both the prefix and postfix form, and have very different meanings.
- In the prefix form, the residual value is the post incremented or post decremented value.
- In the postfix form, the residual value is the pre incremented or pre decremented value.
- These are unary operators, so they should be overloaded as members.

Increment and Decrement

- To distinguish the prefix from the postfix forms, the C++ standard has added an unused argument (int) to represent the postfix signature.
- Since these operators should modify the current object,they should not be const members!
- Prefix: residual vlaue is an lvalue

```
counter & counter::operator ++ () { .... //body }
counter & counter::operator -- () { .... //body }
```

- Postfix: residual value is an rvalue, different than the current object!

```
counter counter::operator ++ (int) { .... //body }
counter counter::operator -- (int) { .... //body }
```


A List
Data Type

List Class Example

- Let's quickly build a partial class using operator overloading to demonstrate the rules and guidelines discussed
- We will re-examine this example again next lecture when discussing user defined type conversions
- The operations that make sense include:

  = for straight assignment of one list to another

  >> and << for insertion and extraction

  + and += for concatenation of two lists & strings

  !=, == for comparison of lists

  [] for accessing a particular string in a list

  ++ for iterating to the next string

Class Interface

```
class node;      //node declaration
class list {     //list.h
  public:
    list(): head(0){}
    list (const list &);
    ~ list();
    list & operator = (const list &);
    friend ostream & operator << (ostream &, const list &);
    friend istream & operator >> (istream &, list &);
    friend list operator + (const list &, const list &);
    friend list operator + (const list &, const string &);
    friend list operator + (const string &, const list &);
    list & operator += (const list &);
    list & operator += (const string &);
    bool operator == (const list &) const;
    bool operator != (const list &) const;
    string & operator [] (int) const;
    string & operator ++ ();   //prefix
    string operator ++ (int);  //postfix
    •••
  private:
    node * head, *ptr, *tail;  //discuss pro's con's

};
```

Copy Constructor

```
//List Class Implementation file:   list.c

class node {      //node definition
  string obj;
  node * next;
};

list::list (const list & l) {
  if (!l.head)
    head = ptr = tail = NULL;
  else {
    head = new node;
    head->obj = l.head->obj;

    node * dest = head;    //why are these local?
    node * source = l.head;
    while (source) {
      dest->next = new node;
      dest = dest->next;
      dest->obj = source->obj;  //what is this doing?
    }
    dest->next = NULL;
    tail = dest;  ptr = head;
  }
}
```

Assignment Operator

```
list & list::operator = (const list & l) {
  if (this == &l) return *this;   //why not *this == l?
  //If there is a list, destroy it
  node * current;
  while (head) {
    current = head->next;
    delete head;
    head = current;
  }
  if (!l.head)
    head = ptr = tail = NULL;
  else {
    head = new node;
    head->obj = l.head->obj;

    node * dest = head;    //why are these local?
    node * source = l.head;
    while (source) {
      dest->next = new node;
      dest = dest->next;
      dest->obj = source->obj;  //what is this doing?
    }
    dest->next = NULL;
    tail = dest;  ptr = head;
  }
}
```

Destructor, Insertion

```
list::~list() {
  node * current;
  while (head) {
    current = head->next;
    delete head;    //what does this do?
    head = current;
  }
  ptr = tail = NULL;
}

ostream & operator << (ostream & out, const list & l) {
  node * current = l.head;  //how can it access head?
  while (current) {
    out <<current->obj <<‘ ‘;  //what does this do?
    current = current->next;
  }
  return out;
}
```

>> Operators

- What interpretation could there be of the >> operator?
  - we could insert new "strings" until a \n is next in the input stream to wherever a current ptr (influenced by ++ and -- operators)
  - we could deallocate the current list and replace it with what is read in
  - we could tack on new nodes at the end of the list
  - others?

>> Operators

```
istream & operator >> (istream & in, list & l) {
        node * current = l.tail;
if (!current) {      //empty list starting out
                        l.head = current = new node;
                        in >>l.head->obj;
        l.tail = l.ptr = l.head;
        l.head->next = NULL;
}

        node * savelist = l.tail->next;
        char next_char;
        while ((next_char = in.peek()) != '\n' &&
    next_char != EOF) {
        current->next = new node;
        current = current->next;
        in >>current->obj;  //what does this do?
        }
        current->next = savelist; ptr = current;
if (!savelist) l.tail = current;
        return in;
}
```

+ Operators

```
list operator + (const list & l1, const list & l2) {
 //remember, neither l1 nor l2 should be modified!
 list temp(l1);   //positions tail at the end of l1
 temp += l2;        //how efficient is this?
 return temp;
}
```

Or, should we instead:

```
list operator + (const list & l1, const list & l2) {
 list temp(l1);   //positions tail at the end of l1
 if (!temp.head) temp = l2;
 else {
   node * dest = temp.tail;
   node * source = l2.head;
   while (source) {
     dest->next = new node;
     dest = dest->next;
     dest->obj = source->obj;
     source = source->next;
   }
   dest->next = NULL; temp.tail = dest;
   temp.ptr = temp.head;
  } return temp;
}
```

+= Operators

```
list & list::operator += (const list & l2) {
  //why wouldn't we program this way?
  *this = *this + l2;
  return *this;
}
```

Or, would it be better to do the following?

```
list & list::operator += (const list & l2) {
 if (!head) *this = l2;  //think about this...
 else {
   node * dest = tail;
   node * source = l2.head;
   while (source) {
     dest->next = new node;
     dest = dest->next;
     dest->obj = source->obj;
     source = source->next;
   }
   dest->next = NULL; tail = dest; //ptr = temp.head; yes?
  } return *this;
}
```

== and != Operators

Notice why a "first" and "second" shouldn't be data members:

```
bool list::operator == (const list & l2) const {
  node * first = head;
  node * second = l2.head;
  while (first && second && first->obj == second->obj) {
    first = first->next;
    second = second->next;
  }
  if (first || second) return FALSE;
  return TRUE;
}
```

Evaluate the efficiency of the following:

```
bool list::operator != (const list & l2) const {
  return !(*this == l2);
  }
```

[] Operator

```
string & list::operator [] (int index) const {
  node * current = head;
  for (int i=0; i< index && current; i++)
    current = current->next;
  if (!current) {
  //consider what other alternatives there are
    string * temp = new string; //just in case
    return *temp;
  }
  return current->obj;
}
```

- Notice how we must consider each special case (such as an index that goes beyond the number of nodes provided in the linked list

++ Operators: Prefix & Postfix

```
string & list::operator ++ () {  //prefix
  if (!ptr || !(ptr->next)) {
  //consider what other alternatives there are
    string * temp = new string; //just in case
    return *temp;
  }
  ptr = ptr->next;
  return ptr->obj;
}
string operator ++ (int){  //postfix
  string temp;
  if (!ptr) {
    temp = "\0";    //what does this do?
    return temp;    //and this?
  }
  temp = ptr->obj;  //and this?
  ptr = ptr->next;  //and this?
  return temp;      //and this?
}
```