Operator Overloading

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 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

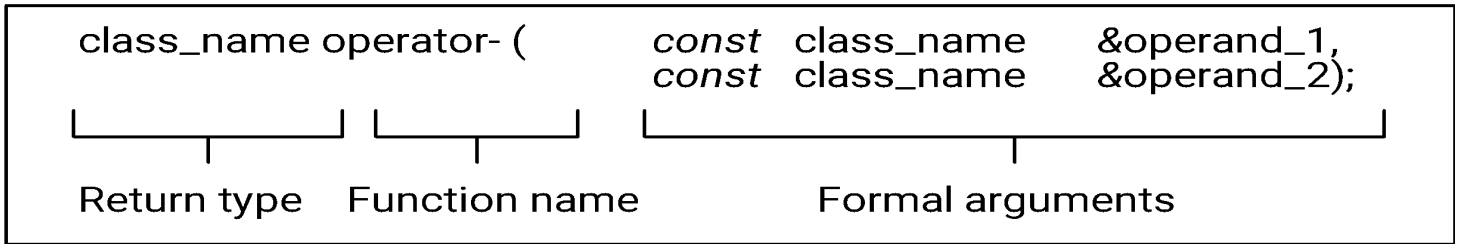
- It is similar to overloading functions
  - except the function name is replaced by the keyword operator followed by the operator's symbol
  - the return type represents the type of the residual value resulting from the operation
    - rvalue? -lvalue?
    - allowing for "chaining" of operations
  - 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.
- 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.
- Ii is recommend that unary operators always be overloaded as members, since the first argument must be an object of a 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. (Normally 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

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

String Class Example

- Let's build a complete class using operator overloading to demonstrate the rules and guidelines discussed
- 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 & 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

- 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

- 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;

ostream & operator << (ostream &, const string &);

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

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);
}
```

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+1];
    strcpy(temp, str);
    strcat(temp, s2.str);
   delete [] str;
    str = temp;    //copy over the pointer
    return *this;  //just copying an address
}
```

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 *);
```

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);
}
```

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 }
```