Operator Overloading

Operator Overloading

- C++ has many operators that work on the built-in types (=, +, *, ->, [], <<, ...)
- C++ allows us to define how the built-in operators work on new classes that we create
- String: +, +=, [], <<
- Most, but not all, built-in operators may be defined for user-defined types

Operator Overloading

- Why do they call it "operator overloading"?
  - Because the same operator has different meanings depending on what type it is applied to
  - There are multiple implementations of the same operator (sometimes even on the same class)
- Operator overloading allows us to fully integrate new types into the C++ language (i.e., they look just like built-in types)

Strings Without Operator Overloading

```
class String {
   …
};

void main() {
   String byu("BYU");
   String gatech;
   gatech.Append("Georgia Tech");
   String msg = gatech.Concat(" threw an interception.");
   msg.Write(cout);
   msg = byu.Concat(" kicked a field goal.");
   msg.Write(cout);
   for (int x=0; x < msg.Length(); ++x) {
      cout << msg.GetChar(x);
   }
}
```

Strings With Operator Overloading

```
class String {
   …
};

void main() {
   String byu("BYU");
   String gatech;
   gatech += "Georgia Tech";
   String msg = gatech + " threw an interception.";
   cout << msg;
   msg = byu + " kicked a field goal.";
   cout << msg;
   for (int x=0; x < msg.Length(); ++x) {
      cout << msg[x];
   }
}
```

Internal String representation

```
class String {
private:
  // string data stored internally
  // as a heap-allocated C-string
  char * str;

public:
  String() {
    Init("");
  }

  String(const char * s) {
    Init(s);
  }

  String(const String & s) {
    Init(s.str);
  }

  ~String() {
    Free();
  }
```

```
private:
  void Init(const char * s) {
    str = new char[strlen(s) + 1];
    strcpy(str, s);
  }

  void Free() {
    delete [] str;
    str = 0;
  }

  …
};
```

operator =

```
class String {

public:
   String & operator =(const char * s) {
      Free();
      Init(s);
      return *this;
   }

   String & operator =(const String & s) {
      if (&s != this) {
         Free();
         Init(s.str);
      }
      return *this;
   }

};
```

operator []

```
class String {

public:
   char GetChar(int index) const {
      return str[index];
   }

   void SetChar(int index, char c) {
      str[index] = c;
   }

};
```

operator []

```
class String {

public:
   char GetChar(int index) const {
      return str[index];
   }

   void SetChar(int index, char c) {
      str[index] = c;
   }

   char & operator [](int index) {
     return str[index];
   }

};
```

operator [] Example

```
void UpperCase(String & s) {

   for (int x=0; x < s.Length(); ++x) {
      s[x] = toupper(s[x]);
   }

}
```

L-value

R-value

operator +=

```
class String {
public:
   void Append(const char * s) {
      char * newStr = new char[strlen(str) + strlen(s) + 1];
      strcpy(newStr, str);
      strcat(newStr, s);
      Free();
      str = newStr;
   }

   void Append(const String & s) {
      Append(s.str);
   }

};
```

operator +=

```
class String {
public:
   void Append(const char * s) {
      char * newStr = new char[strlen(str) + strlen(s) + 1];
      strcpy(newStr, str);
      strcat(newStr, s);
      Free();
      str = newStr;
   }

   void Append(const String & s) {
      Append(s.str);
   }

   void operator +=(const char * s) {
      Append(s);
   }

   void operator +=(const String & s) {
     Append(s);
   }

};
```

operator +

```
class String {

public:
   String Concat(const char * s) const {
      String result(str);
      result.Append(s);
      return result;
   }

   String Concat(const String & s) const {
      return Concat(s.str);
   }

};
```

operator +

```
class String {

public:
   String Concat(const char * s) const {
      String result(str);
      result.Append(s);
      return result;
   }

   String Concat(const String & s) const {
      return Concat(s.str);
   }

   String operator +(const char * s) const {
      return Concat(s);
   }

   String operator +(const String & s) const {
      return Concat(s);
   }

};
```

Conversion Operators

- Constructors convert other types to our new type
  - const char * -> String
- Conversion operators convert our new type to other types
  - String -> const char *

operator const char *

```
class String {

public:
   const char * CString() const {
      return str;
   }

};
```

operator const char *

```
class String {

public:
   const char * CString() const {
      return str;
   }

   operator const char *() const {
      return str;
   }

};
```

Overloading Operators On Types You Didn't Create

- What about this?

```
const String ONLY_A_TEST = ", but only a test";
String msg = "this is a test" + ONLY_A_TEST;
```

- Or this?

```
cout << msg << endl;
```

- How can we overload operators on const char * and ostream?

operator +

```
class String {

public:
   String operator +(const char * s) const {
      return Concat(s);
   }

   String operator +(const String & s) const {
      return Concat(s);
   }

};

// STAND-ALONE FUNCTION

String operator +(const char * s1, const String & s2) {
  return String(s1) + s2;
}
```

operator <<

```
class String {

public:
   void Write(ostream & s) const {
      s << str;
   }

};

// STAND-ALONE FUNCTION

ostream & operator <<(ostream & os, const String & s) {
  s.Write(os);
  return os;
}
```