Welcome

Object Oriented Programming with C++
CIS 265

Week 6 – Classes and Operator overloading

Christopher K. Burns

Schedule

Week		Content
1	1/11	*Review* Chapter 1 Intro to Computers and C++ Programming Chapter 2 Control Structures Chapter 3 Functions Chapter 4 Arrays
2	1/18	Additional Array and Function Topics Chapter 5 Pointers and Strings Lab 1 – Functions, Arrays, and Strings Homework 1 Assigned
3	1/25	Chapter 6 Classes and Data Abstraction
4	2/1	Chapter 7 Classes: Part I Lab 2 – Classes 1 Homework 1 Due
5	2/8	***MID-TERM EXAMINATION*** *(Chapters 1 through 7*)* **only portions of chapter 7 that were covered in class* *Final Project Assigned*
6	2/15	Chapter 7 Classes: Part II Chapter 8 Operator Overloading Homework 2 Assigned
7	2/22	Chapter 8 Operator Overloading Homework 2 Due Lab 3 – Classes 2
8	3/1	Chapter 9 Inheritance: Part I Homework 3 Assigned
9	3/8	Chapter 9 Inheritance: Part II Chapter 10 Polymorphism Homework 3 Due
10	3/15	Chapter 10 Polymorphism Lab 4 – Inheritance and Polymorphism
11	3/22	***FINAL PROJECTS DUE***

Agenda

Tonight's agenda

- Mid Term – Review
- Final Project Topics
- Classes II
  - Static Members
  - *this* pointer
  - Operator Overloading

Final Project

- Develop a C++ program that makes use of the techniques we have discussed up till now. Must also use *at least* one advanced technique (e.g. operator overloading, inheritance, vector…)
- Topic of the program is of your choosing.
- Additional technologies
  - OpenGL
  - Win32

Object Oriented Programming

Home Work

Homework 2

Final Project topics

Chapter 8 in text

Classes - Review

What we've covered about classes up to this point:

- Basic structure of a class
- Constructors and overloading
- Separating Interface from Implementation
- Data encapsulation
  - Prefer to declare all members variables as private
  - Implement accessor methods (get and set) functions for those variables that need to be modified by consumer of class
- Static members*
- this pointer*

Static Members

Classes and Objects

A class definition itself exists and may have methods and variables that can be accessed using class scope. This is done by declaring members as static. There is only one class, thus the static members are global within the class for all objects.

An instance of you class is an object. Your class may include instance data that is only accessible to objects. There can be many instances of a class.

Static Members Example

```
#include <iostream>
using namespace std;
const int MAX_STRING_LENGTH = 256;

class Student
{
private:
  long id;
  static long studentCount;
public:
  Student()
  {
    studentCount++;
  }
  static long getStudentCount()
  {
    return studentCount;
  }
  long getID() { return id; }
  void setID( long inID ) { id = inID; }
};

long Student::studentCount = 0;
```

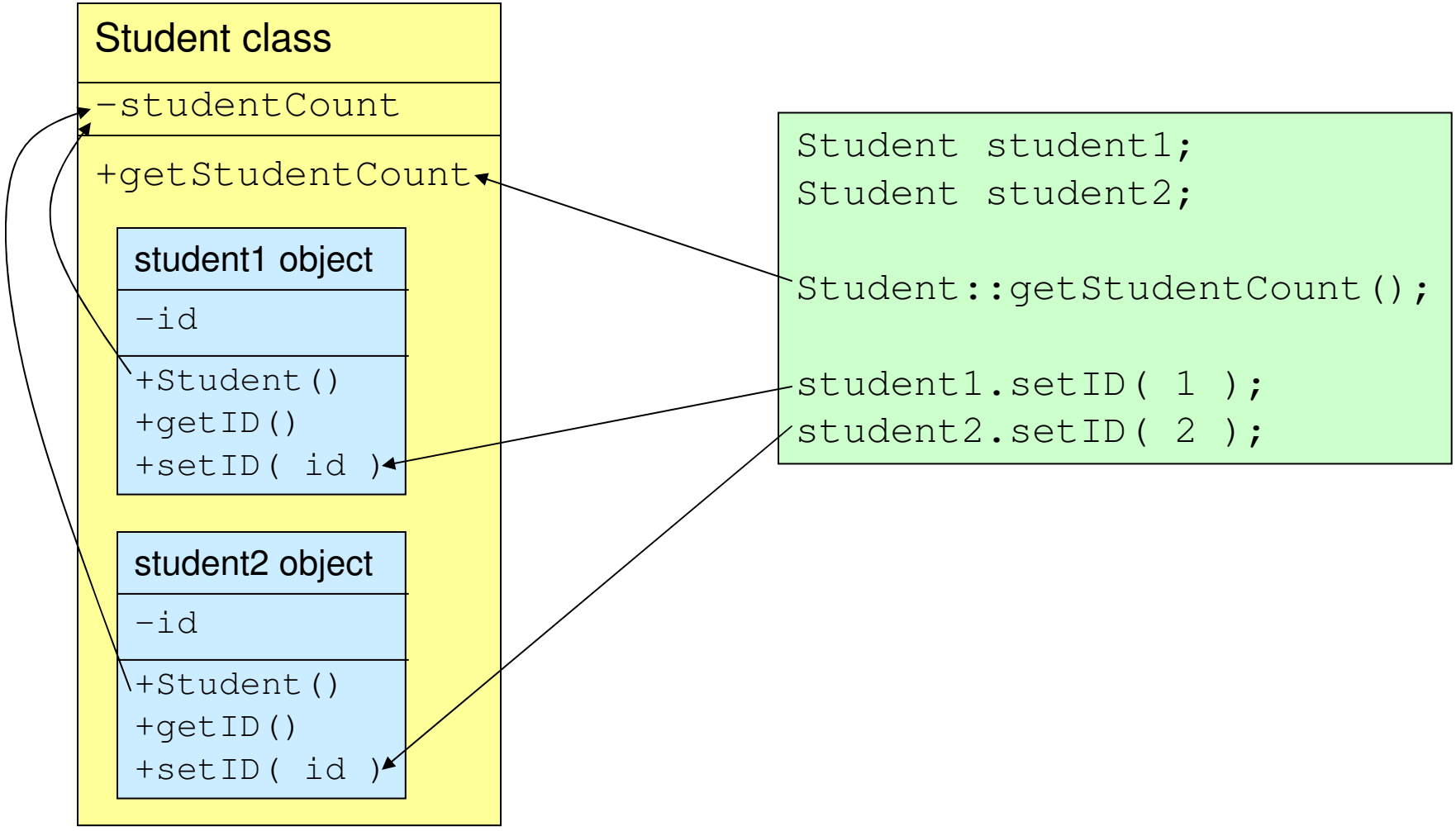
static members

```
void main()
{
  Student student1;
  Student student2;
  Student student3;
  cout << "There are " <<
    Student::getStudentCount();
  cout << " students" << endl;
}
```

class scope

```
There are 3 students
```

Static Members



Identity Crisis

All these instances… who am I?

A class instance (object), can refer to itself using a special pointer called *this*

The *this* pointer points to the objects itself

It allows the class instance to know who it is, as well as being able to tell others who it is

The *this* pointer

There are many uses for the *this* pointer, three of which are:

1. Avoiding name collisions
2. Cascading method calls
3. Passing an instance of yourself

The *this* pointer

Example 1 – get/set – avoiding name collisions

```
class Student
{
private:
  long id;
public:
  long getID() { return id; }
  void setID( long id )
  {
    this->id = id;
  }
};
```

The *this* pointer

Example 2 – cascading method calls

```
class Student
{
private:
  long id;
  bool enrolled;
public:
  long getID() { return id; }
  Student& setID( long id )
    { this->id = id; return *this; }
  bool isEnrolled() { return enrolled; }
  Student& setEnrolled( bool enrolled )
    { this->enrolled = enrolled; return *this; }
};

void main()
{
  Student student;
  student.setEnrolled( true ).setID( 1 );
}
```

The *this* pointer

Example 3 – passing your instance around

```
#include <iostream>
using namespace std;

// constants
const int MAX_STRING_LENGTH = 256;
const int MAX_STUDENTS = 20;

// class Student interface
class Student
{
private:
  static long studentCount;
  static Student* studentList[ MAX_STUDENTS ];

  char name[ MAX_STRING_LENGTH ];
  bool enrolled;
  long id;

public:
  Student( char*, bool );

  const char* getName();
  void setName( char* );
  bool isEnrolled();
  void setEnrolled( bool );
  long getID();

  static Student* getStudentFromID( long id );
  static long getStudentCount();
};
```

```
// class Student implmentation
Student::Student( char* name, bool enrolled )
{
  setName( name );
  setEnrolled( enrolled );
  id = studentCount;
  studentList[ Student::studentCount ] = this;
  studentCount++;
}

const char* Student::getName()
{
  return name;
}

void Student::setName( char* name )
{
  strcpy( this->name, name );
}

bool Student::isEnrolled()
{
  return enrolled;
}

void Student::setEnrolled( bool enrolled )
{
  this->enrolled = enrolled;
}
```

The *this* pointer

Example 3 – passing your instance around – cont'd

```
long Student::getID()
{
  return id;
}

// class Student static method implmentation
Student* Student::getStudentFromID( long id )
{
  if ( ( id <= studentCount ) && ( id >= 0 ) )
  {
    return studentList[ id ];
  }
  else
  {
    return 0;
  }
}

long Student::getStudentCount()
{
  return studentCount;
}

// class Student static member initialization
long Student::studentCount = 0;
Student* Student::studentList[ MAX_STUDENTS ];
```

```
// main routine to demonstrate Student
void main()
{
  // get students
  do
  {
    char name[ MAX_STRING_LENGTH ];
    bool enrolled;
    cout << "Student name:";
    cin >> name;
    if ( stricmp( name, "end" ) == 0 )
      break;
    cout << "   enrolled?:";
    cin >> enrolled;
    new Student( name, enrolled );
  }
  while( true );
```

The *this* pointer

Example 3 – passing your instance around – cont'd

```
  // print list of students
  for( int i = 0; i < Student::getStudentCount(); i++ )
  {
    cout << "Student Record" << endl;
    cout << " name:    " << Student::getStudentFromID(i)->getName() << endl;
    cout << " id:      " << Student::getStudentFromID(i)->getID() << endl;
    cout << " enrolled:" << Student::getStudentFromID(i)->isEnrolled() << endl;
  }
}
```

Output

```
Student name:Eric
   enrolled?:1
Student name:Kenny
   enrolled?:0
Student name:end
Student Record
 name:    Eric
 id:      0
 enrolled:1
Student Record
 name:    Kenny
 id:      1
 enrolled:0
```

Operator Overloading

Scenario: Although you enjoy the standard string functions strcpy, strcat, and strcmp, you'd like to create a string class that encapsulates these.

Your string class might then have methods for assignment, appending strings, and checking for equivalency.

Operator Overloading

The interface for your class would likely be:

```
class MyString
{
private:
  char string[ MAX_STRING ];
public:
  MyString();
  MyString( const char* );
  const char* getString();
  MyString& setString( const char* );
  MyString& appendString( const char* );
  bool isEqualTo( const char* );
};
```

Operator Overloading

You could then write code like:

```
void main()
{
 MyString string1( "Hello" );
 MyString string2( "Good bye" );
 MyString string3;

 string3.setString( string1.getString() );
 string3.appendString(" and ").appendString(string2.getString() );
 if ( string1.isEqualTo( string2.getString() ) )
 {
   cout << "string1 == string2" << endl;
 }
 cout << string3.getString() << endl;
}
```

Operator Overloading

That is good, but wouldn't it be better if you could instead write code like:

```
void main()
{
 MyString string1( "Hello" );
 MyString string2( "Good bye" );
 MyString string3;

 string3 = string1;
 string3 += " and ";
 string3 += string2;
 if ( string1 == string2 )
 {
   cout << "string1 == string2" << endl;
 }
 cout << string3 << endl;
}
```

Operator Overloading

This can be done in C++ by “overloading” operators.

Our string class can overload the =, ==, +, and += operators

Operator overloading gives you the ability to have your classes work with the standard C++ operators

This makes your class more intuitive to use

Operator Overloading

Why can't C++ do this for me?

In some ways it tries to. The "=" operator is already overloaded for your class. When "=" is encountered, the compiler performs member assignments. But the results can sometimes be unpredictable

Often your class is doing something special with these operators that only you as the class writer understand. Examples would be having "+" perform a special mathematical operation or appending strings.

Operator Overloading

Using the standard function

```
strcmp(string1.getString(), string2.getString())
```

Wrapping the standard function in a method

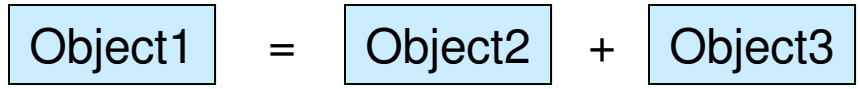
```
string1.isEqualTo( string2.getString )
```

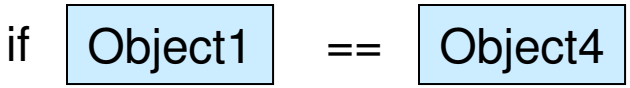
Using operator overloading

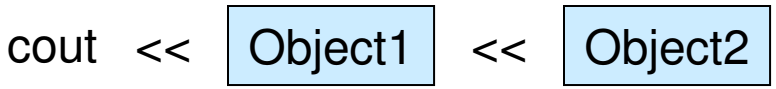
```
string1 == string2
```

Operator Overloading

You can overload operators on any class you create, thus allowing objects of your class to be used:

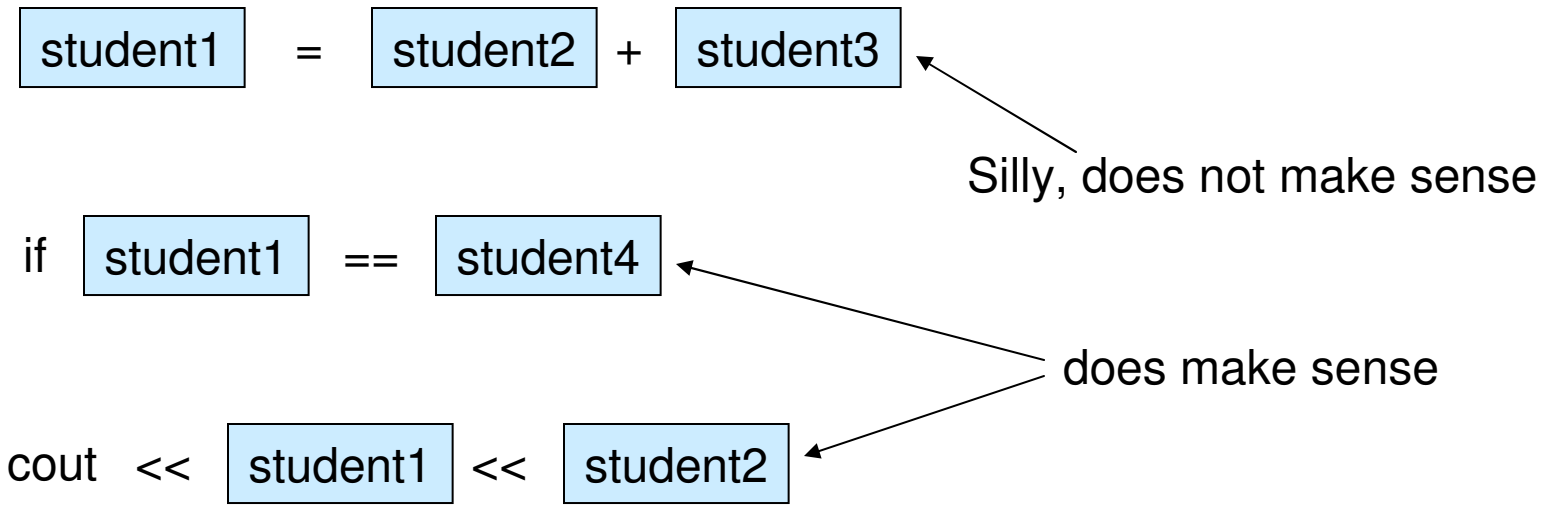






Operator Overloading

Use operator overloading where it makes sense



Operator Overloading

Operators that can be overloaded

```
+       -       *       /       %       ^       &       |
~       !       =       <       >       +=      -=      *=
/=      %=      ^=      &=      |=      <<      >>      >>=
<<=     ==      !=      <=      >=      &&      ||      ++
--      ->*     ,       ->      []      ()      new     delete
new[]           delete[]
```

Operators that can not be overloaded

```
.       .*      ::      ?:
```

Operator Overloading

Overloading the = operator:

```
ClassA = Something
```

The = operator will take a parameter of type Something

The = operator must return a reference to ClassA

```
class SomeClass
{
public:
  Integer& operator= ( const SomeType setMe )
  {
    value = setMe;
    return *this; // return a reference to myself
  }
};
```

Operator Overloading

Example: overloading the = operator

```
class Integer
{
private:
  int value;
public:
  Integer& operator= ( const int setMe )
  {
    value = setMe;
    return *this;
  }
};

void main()
{
  Integer i;
  i = 1;
}
```

Operator Overloading

Example: overloading the overloaded = operator

```
class Integer
{
private:
  int value;
public:
  Integer& operator= ( const int setMe )
  {
    value = setMe;
    return *this;
  }
 Integer& operator= ( const Integer setMe )
  {
    value = setMe.value;
    return *this;
  }
};
```

Operator Overloading

Overloading the + operator:

```
ClassA = ClassB + Something
```

The + operator will take a parameter of type Something

The + operator must return a value (not a reference) to allow the operators to cascade

```
class SomeClass
{
public:
  Integer operator+ ( const SomeType setMe )
  {
    Integer temp;
    temp.value = value + addMe.value;
    return temp;
  }
};
```

Operator Overloading

Example: overloading the + operator

```
class Integer
{
private:
  int value;
public:
  Integer operator+ ( const Integer& addMe )
  {
    Integer temp;
    temp.value = value + addMe.value;
    return temp;
  }
  Integer operator+ ( const int addMe )
  {
    Integer temp;
    temp.value = value + addMe;
    return temp;
  }
};
```

Operator Overloading

Example: overloading the += operator

```
class Integer
{
private:
  int value;
public:
 Integer& operator+= ( const int addMe )
  {
    value = value + addMe;
    return *this;
  }

  Integer& operator+= ( const Integer addMe )
  {
    value = value + addMe.value;
    return *this;
  }
};
```

Operator Overloading

Example: overloading the == operator

```
class Integer
{
private:
  int value;
public:
 bool operator== ( const Integer checkMe )
  {
    if ( this->value == checkMe.value )
    {
      return true;
    }
    else
    {
      return false;
    }
  }
};
```

Operator Overloading

What is really happening when operator overloading is used:

So far, our overloaded operators have really just been member functions of our class Integer.

```
i += 10;
// can also be written as:
i.operator +=( 10 );
```

What is really occurring in `i += 10` is that the 10 is being passed as a parameter to the overloaded member function called `i.operator +=`

Operator Overloading

Our overloaded function could also be written as a non-member function; basically just as a regular old function.

Our non-member function that overloads the `+=` operator would then have to have access to the private member `Integer.value`. This could be done either by making value `public`, or adding and accessors.

Operator Overloading

```
class Integer
{
private:
  int value;
public:
  void setValue( int value ) { this->value = value; }
  int getValue() { return value; }
};

const Integer operator+=( Integer& setMe, const int addMe )
{
  setMe.setValue( setMe.getValue() + addMe );
  return setMe;
}
void main()
{
  Integer i;
  i.setValue( 1 );
  operator+= ( i, 10 ); // now, operator+= takes two parameters
  i += 10;
  cout << i.getValue();
}
```

Look, it's not a member of class Integer

Operator Overloading

That wasn't so bad, but notice that when the operator is a non-member function it has to be passed a reference to the object it is operating on.

If the operator takes an object of the class as its first parameter, then the operator can be implemented as a member function. Two operators for which this is not the case are << and >>.

Most of the time, overloaded assignment operators are best left as member functions.

Overloaded comparison operators are often easier to implement as non-member functions.

Operator Overloading

We had to add two accessor methods, getValue and setValue in order to implement the non-member += operator.

The reason was a non-member function (a function that is not part of the class) cannot access private variables.

That's cumbersome…

Hmmm, but what if the non-member function could be really nice to the class and they could become friends; then they could trust each other and…

Friends

Classes can have two types of friends:

- classes and functions

Friends of a class can access that classes private members

why are we talking about this in the middle of discussing operators?

Operator Overloading

WARNING

There is a bug in the Visual Studio 6.0 that causes an error to be thrown when private member accessed in a friend function.

On your home systems download Service Pack 5 to fix this:
http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/

A work around is to implement the friend method within the class interface (no external implementation for this friend ;))

The code shown in these next examples should compile in the non-service packed and service packed versions of Visual Studio 6.0

Operator Overloading

Overloading the stream operators:

The stream operators << and >> act a little differently from the other standard operators in that their first parameter is not a user defined object but the cin/cout object.

```
cout << i;
// is really the same as
operator<< ( cout, i )
```

This means that the overloaded << and >> operators cannot be member functions of `i`

Thus, they are usually implemented as friend functions

Operator Overloading

Overloading the >> and << operators:

```
cout << objectA
cin >> objectA
```

The operator must return a reference to the appropriate stream type

```
friend ostream& operator<< ( ostream& out, ClassA classA )
{
  return out << classA.value;
}
friend istream& operator>> ( istream& in, ClassA& classA )
{
  return in >> classA.value;
}
```

Operator Overloading

overloading the << and >> operators

```
class Integer
{
private:
  int value;
public:
  friend ostream& operator<< ( ostream& out, Integer printMe )
  {
    out << "Integer is " << printMe.value;
    return out;
  }
  friend istream& operator>> ( istream& in, Integer& getMe )
  {
    char temp[256];
    in >> temp;
    getMe.value = atoi( temp );
    return in;
  }
};
void main()
{
  Integer i;
  cout << "Enter number        :";
  cin >> i;
  cout << i << endl;
}
```

Operator Overloading

…back to our string class. Our string class with overloading:

```
class MyString
{
private:
  char string[ MAX_STRING ];
public:
  // constructors
  MyString();
  MyString( const char* );
  // overloaded operators
  const MyString& operator= ( const MyString myString );
  const MyString& operator= ( const char* string );
  MyString operator+ ( const MyString& myString );
  MyString operator+ ( const char* string );
  const MyString& operator+= ( const MyString myString );
  const MyString& operator+= ( const char* string );
  bool operator== ( const MyString myString );
  friend ostream& operator<< ( ostream& out, const MyString& myString) {
    return out << myString.string; }
  friend istream& operator>> ( istream& in, MyString& myString ) {
    return in >> myString.string; }
};
```

The Standard Library

The C++ Standard Library,

formerly known as STL (Standard Template Library)

Provides many useful structures and containers

Two of these that we are going to look at are the:

standard string

vector

Standard String

The C++ standard string is a class. It includes numerous methods for

- inserting strings,
- finding substrings,
- testing for empty,
- Overloaded operators such as +, ==, [] so appending, comparing, and manipulating strings is easy