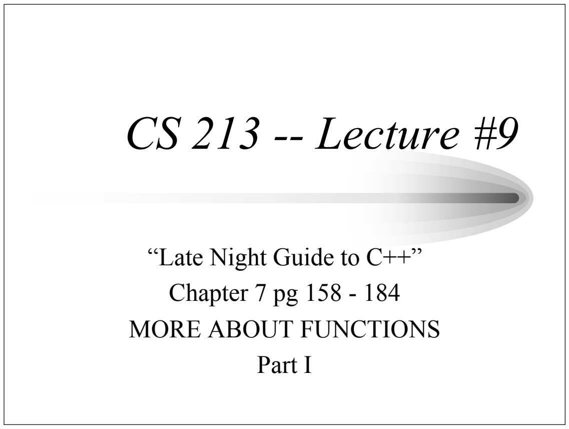
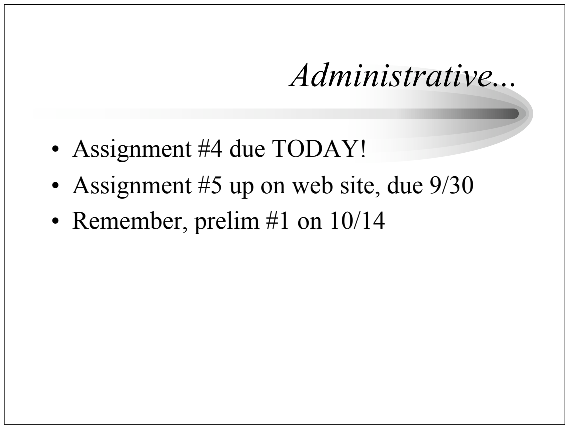


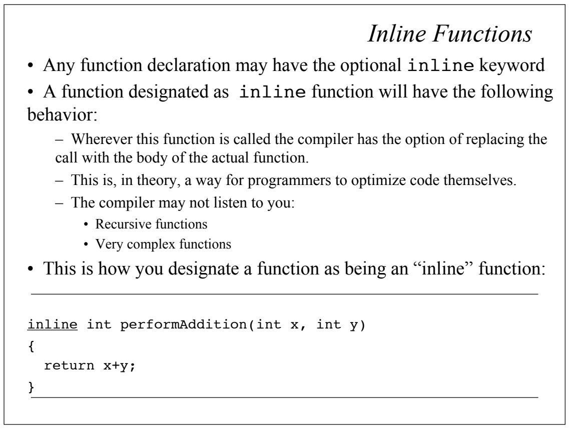
CS 213 -- Lecture #9

"Late Night Guide to C++"
Chapter 7 pg 158 - 184
MORE ABOUT FUNCTIONS
Part I



Administrative...

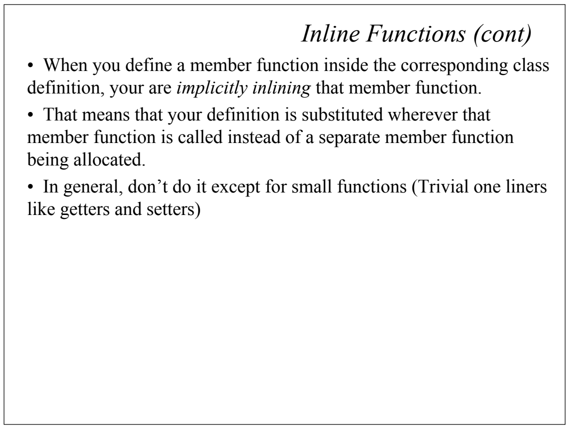
- Assignment #4 due TODAY!
- Assignment #5 up on web site, due 9/30
- Remember, prelim #1 on 10/14



Inline Functions

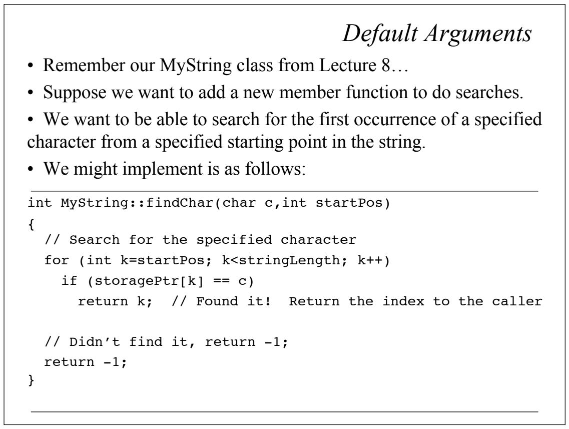
- Any function declaration may have the optional `inline` keyword
- A function designated as `inline` function will have the following behavior:
  - Wherever this function is called the compiler has the option of replacing the call with the body of the actual function.
  - This is, in theory, a way for programmers to optimize code themselves.
  - The compiler may not listen to you:
    - Recursive functions
    - Very complex functions
- This is how you designate a function as being an "inline" function:

```
inline int performAddition(int x, int y)
{
  return x+y;
}
```



Inline Functions (cont)

- When you define a member function inside the corresponding class definition, your are *implicitly inlining* that member function.
- That means that your definition is substituted wherever that member function is called instead of a separate member function being allocated.
- In general, don't do it except for small functions (Trivial one liners like getters and setters)

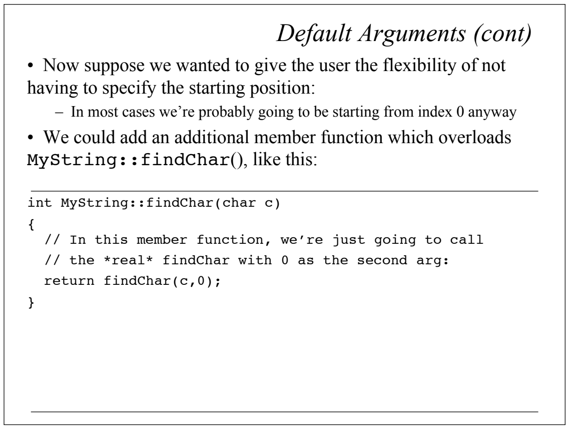


Default Arguments

- Remember our MyString class from Lecture 8…
- Suppose we want to add a new member function to do searches.
- We want to be able to search for the first occurrence of a specified character from a specified starting point in the string.
- We might implement is as follows:

```
int MyString::findChar(char c,int startPos)
{
  // Search for the specified character
  for (int k=startPos; k<stringLength; k++)
    if (storagePtr[k] == c)
      return k;  // Found it!  Return the index to the caller

  // Didn't find it, return -1;
  return -1;
}
```



Default Arguments (cont)

- Now suppose we wanted to give the user the flexibility of not having to specify the starting position:
  - In most cases we're probably going to be starting from index 0 anyway
- We could add an additional member function which overloads `MyString::findChar()`, like this:

```
int MyString::findChar(char c)
{
  // In this member function, we're just going to call
  // the *real* findChar with 0 as the second arg:
  return findChar(c,0);
}
```

Default Arguments (cont)

- This gives us the following member functions in MyString:

```
class MyString
{
public:
  MyString();
  MyString(string);
  virtual ~MyString();
  string MakeString();
  int MakeInt();
  bool setValue(string);
  int readString();
  int readString(int);
  int findChar(char);
  int findChar(char,int);
…
}
```

Default Arguments (cont)

- What we’re really doing with findChar() is providing two definitions, one of which uses a *default value* for one of the parameters.
- However instead of going to the trouble of having two member functions defined, C++ gives us a way to specify a default value for a parameter right in the declaration (but not the definition):

```
class MyString
{
  …
  int findChar(char c,int startPos=0);
  …
}
int MyString::findChar(char c,int startPos);
{
  …
}
```

Default Arguments (cont)

- To have the compiler use the default value for a given argument I simply omit that argument when calling the function.
- This means that default arguments *must* come at the end of a function declaration.
- In other words, you cannot have an argument with a default value specified appear before a regular argument (with no default specified)

```
int main()
{
  MyString aStr;
  aStr.setValue(“This is a test”);

  int pos = aStr.findChar(‘s’);
  cout << “the first s is at position: “ << pos << endl;
  cout << “the next on is at “ << aStr.findChar(‘s’,pos+1)
       << endl;
}
```

Demonstration

`MyString`:

Using Default Arguments

Default Arguments (cont)

- Remember, parameters with default values need to appear at the *end* of your parameter list.
- Once you choose to take the default value when calling a function with default values in it, all subsequent parameters must take the default as well.

```
int findChar(char c,int startPos = 0);
int findChar(char c,int startPos = 0, int stopPos);  // ???
int findChar(char c,int stopPos, int startPos = 0);  // ???
int findChar(char c,int startPos = 0, int stopPos = -1);
```

- Sometimes default arguments can be awkward:
  - No clean way to specify a default value for `stopPos` (length of string)
  - We could omit the default value, but then we’d need to put `stopPos` before `startPos`!
  - We resort to “flag passing” (-1 to mean we want to search until the end of the string)

Default Arguments (cont)

- Let’s take a look at how that might be implemented:

```
int MyString::findChar(char c,int startPos,char stopPos)
{
  // Determine what our stopping point is.  Introducing
  // some really weird notation...
  int stopAt = (stopPos == -1) ? stringLength-1 : stopPos;

  // Search for the specified character
  for (int k=startPos; k<=stopAt; k++)
    if (storagePtr[k] == c)
      return k;  // Found it!  Return the index to the caller

  // Didn’t find it, return -1;
  return -1;
}
```

Default Arguments (cont)

- According to LNG, a default value “can be any expression, it needn’t be a constant. Note, though, that any variables involved are statically bound, so be careful when using default argument values with virtual functions.”
- Stroustrup makes no reference to being able to specify a default value with a variable.
- I’ve never seen it used.
- It would have been useful in our `MyString::findChar()` member function to pass `stringLength` as the default value for the `stopPos` argument, but...
- You cannot use member variables (unless they are *static*, but we haven’t covered static members yet)
- You can use global variables
- Let’s look at MyString again with our `findChar` modification.

Demonstration

`MyString`:

With New and Improved findChar()

Returning References

- Remember, a reference is *like* a pointer (pointers are used to implement references), so when you return a reference to an object it’s like returning a pointer.
- Remember, too, that functions which return pointers usually allocate the memory they return a pointer to. Otherwise the potential for dangling pointers exists.
- Since you can’t dynamically allocate memory directly to a reference (like you can for a pointer) you are more likely to return pointers than references when performing this type of work.
- So when *is* it a good idea to return a reference?
- Consider the following code:

```
char MyString::element(int k)
{
  if ((k >= 0) && (k < stringLength))
    return storagePtr[k];
}
```

Returning References (cont)

- MyString::element() can be used to access an individual character in our string.
- So, it’s a nice shorthand.
- BUT, if we change our definition so that it returns a reference...

```
char &MyString::element(int k)
{
  if ((k >= 0) && (k < stringLength))
    return storagePtr[k];
}
```

- There is an interesting side effect. We can now put a call to this function on the *left side* of an assignment operator.
- That is, we can now make assignments to a given element of our string using a call to this function...

Demonstration

`MyString`:

MyString::element as an l-value

Function Overloading

- We’ve touched on overloading in past lectures.
- To review, you can provide multiple definitions of the same function (or member function) with different parameter lists.
- Depending on how the function is called, the compiler will call the version of your function that matches the arguments passed.
- Consider the following:

```
void MyRead(char &aChar)
{
  MyString aStr(“”);
  aStr.readString();
  aChar = aStr.element(0);
}
```

- This provides a convenient way to read a character (although we could have done it just as easily with MyString directly...

Function Overloading (cont)

- But consider making the following additions:

```
void MyRead(int &anInt)
{
  MyString aStr(“”);
  aStr.readString(10);
  anInt = aStr.MakeInt();
}
void MyRead(string &aString)
{
  MyString aStr(“”);
  aStr.readString(80);
  aString = aStr.MakeString();
}
void MyRead(bool &aBoolean);  // you get the idea
void MyRead(float &aFloat);
...
```

Function Overloading (cont)

- Then, all I’d have to do to read in different types would be to call `MyRead` with a reference to the appropriate type.
- Consider the following:

```
int main()
{
  char aChar;
  int anInt;
  string aString;

  cout << “Enter a character: “;  MyRead(aChar);
  cout << “Enter an integer:  “;  MyRead(anInt);
  cout << “Enter a string:    “;  MyRead(aString);
  cout << “You entered: “ << aChar << “, “ << anInt <<
          “, and “ << aString << endl;
}
```

Operator Overloading (cont)

- In addition to overloading functions, you can also overload operators.
- The following operators may be overloaded (from LNG pg 170-171)

```
Unary Operators:
++  --  ~  !  -  +  &  *  new  new[]  delete  delete[]

Binary Operators:
->  *  /  %  +  -  <<  >>  <  <=  >  >=  ==  !=  &
^  |  &&  ||

Assignment Operators:
=  *=  /=  %=  +=  -=  <<=  >>=  &=  |=  ^=
```

- You cannot alter precedence, only extend the definition as they apply to the particular class you are overloading them from.

Unary Operator Overloading (cont)

- Just for fun, let’s overload the unary ~ to mean string length in `MyString`, and + to return a C++ string class instance.
- To overload, we use the following definition:

```
int MyString::operator~()
{
  return stringLength;
}

string MyString::operator+()
{
  return MakeString();
}
```

- Let’s check it out...

Demonstration

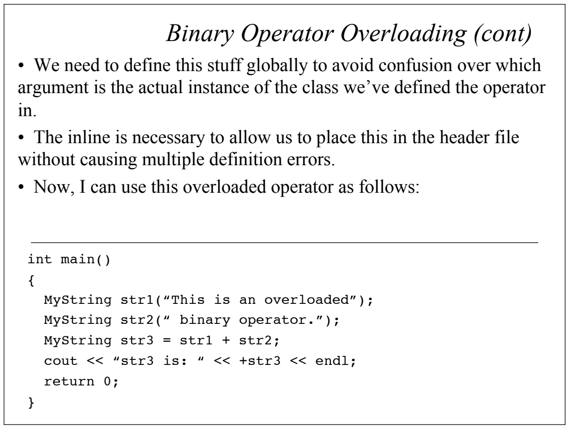
`MyString`:
Unary Operator Overloads

Binary Operator Overloading

- I can see it now, you’re all thinking “COOL, what else can we overload”.
- OK, ok, you don’t have to twist my arm. How about overloading the binary + to do concatenation?

```
inline MyString operator+(const MyString &str1,
                          const MyString &str2)
{
  // OK, we’ll cheat.  Let’s just take two string
  // variables and concatenate them that way...
  MyString temp( (+str1) + (+str2) );
  return temp;
}
```

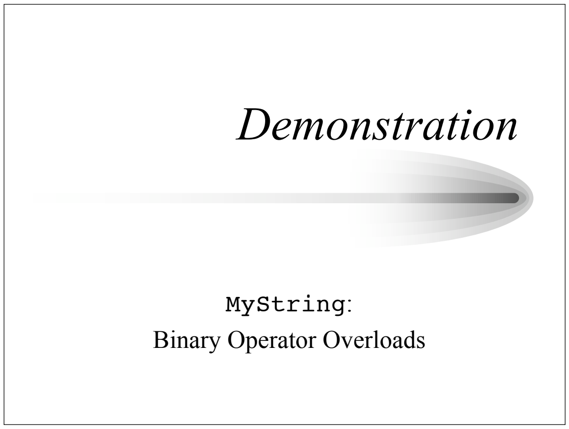
- But why inline? Any why is this defined globally?



Binary Operator Overloading (cont)

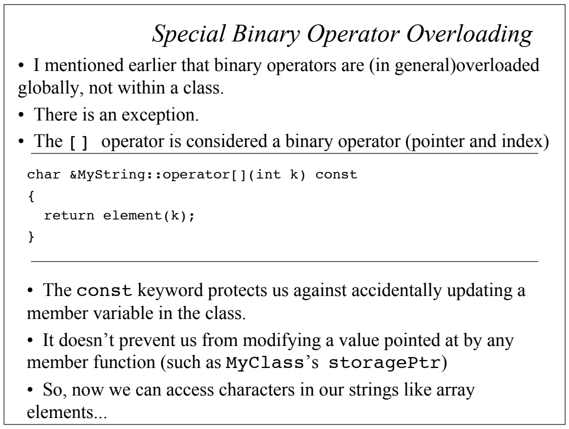
- We need to define this stuff globally to avoid confusion over which argument is the actual instance of the class we've defined the operator in.
- The inline is necessary to allow us to place this in the header file without causing multiple definition errors.
- Now, I can use this overloaded operator as follows:

```
int main()
{
  MyString str1("This is an overloaded");
  MyString str2(" binary operator.");
  MyString str3 = str1 + str2;
  cout << "str3 is: " << +str3 << endl;
  return 0;
}
```



Demonstration

`MyString`:
Binary Operator Overloads

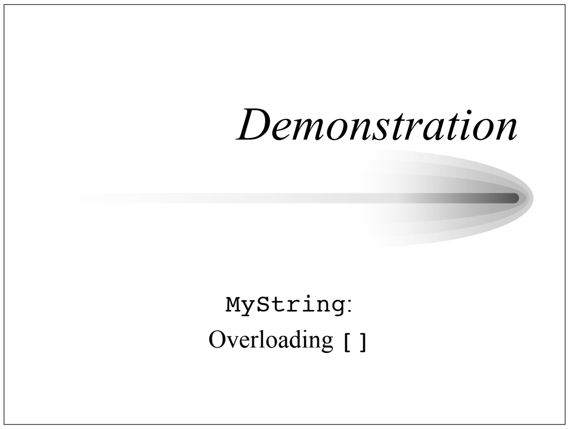


Special Binary Operator Overloading

- I mentioned earlier that binary operators are (in general)overloaded globally, not within a class.
- There is an exception.
- The `[]` operator is considered a binary operator (pointer and index)

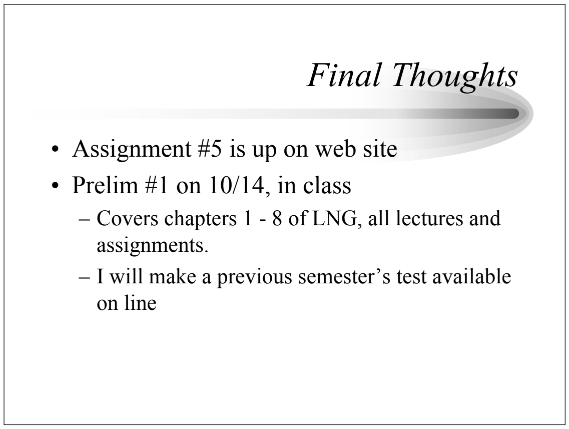
```
char &MyString::operator[](int k) const
{
  return element(k);
}
```

- The `const` keyword protects us against accidentally updating a member variable in the class.
- It doesn't prevent us from modifying a value pointed at by any member function (such as `MyClass`'s `storagePtr`)
- So, now we can access characters in our strings like array elements...



Demonstration

`MyString`:
Overloading `[]`



Final Thoughts

- Assignment #5 is up on web site
- Prelim #1 on 10/14, in class
  - Covers chapters 1 - 8 of LNG, all lectures and assignments.
  - I will make a previous semester's test available on line