Welcome

Object Oriented Programming with C++
CIS 265

Week 3 – C with Classes

Christopher K. Burns

Syllabus – 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

- Review of last week
  - Arrays, Pointers and Strings
  - Homework 1 and Lab 1 questions
- References
- Function overloading
- C++ Classes

Object Oriented Programming

Home Work

Homework #1 – Due next week

Lab #1 – Due next week

Read Chapter 6

Pointers and References

Pointers allow for pointer math and the actual address contained in the pointer can be changed.

C++ introduced a different type of pointer called a reference.

A reference is a pointer in spirit but it acts just like the variable it is pointing to.

A reference can be thought of as an alias for a variable.

Pointers and References

```
int i = 10;

int &iRef = i;                  // iRef is a reference to i

int *iPtr = &i;                 // iPtr is a pointer to I

cout << “i = “ << i << “ iRef = “ << iRef << “ *iPtr = “ << *iPtr << endl

i = 10   iRef = 10   *iPtr = 10
```

Passing References

```
void swapRef( int &a, int &b )
{
  int temp = a;
  a = b;
  b = temp;
}

main()
{
  int i = 10;
  int j = 20;
  cout << "i = " << i << endl;
  cout << "j = " << j << endl;
  cout << "swapping i and j" << endl;
  swapRef( i, j );
  cout << "i = " << i << endl;
  cout << "j = " << j << endl;
}
```

Passing Pointers

```
void swap( int* a, int* b )
{
  int temp = *a;
  *a = *b;
  *b = temp;
}

main()
{
  int i = 10;
  int j = 20;
  cout << "i = " << i << endl;
  cout << "j = " << j << endl;
  cout << "swapping i and j" << endl;
  swap( &i, &j );
  cout << "i = " << i << endl;
  cout << "j = " << j << endl;
}
```

Ad-hoc Polymorphism

- Function Overloading
  - You can have multiple functions with the same function name.
  - The functions are differentiated by the parameters that are passed to them.
  - The following functions all have the same name, so it is easy for the user of your function. Depending on the parameters passed a different actual version of your function is called.
    - printMe(char*)
    - printMe(int)
    - printMe(float)
    - printMe(char*, char*)

Ad-hoc Polymorphism

- Function Overloading
  - In C++ the name of your function is:

    functionName + parameterNames

    (this is what is referred to a C++ name mangling)
  - The C++ name of the function does not include the return value since it is optional for the caller to use the return value.
  - Thus in C++:

int printMe(char*) has the same name as bool printMe(char*)

Ad-hoc Polymorphism

- Function Overloading
  - The C++ name of the function does not include the return value since it is optional for the caller to use the return value.
  - Thus in C++:

    int printMe(char*)

    has the same name as

    bool printMe(char*)
  - If you tried to compile you would get the error:

```
error C2556: int printMe(char*) overloaded
function differs only by return type from bool
printMe(char*)
```

Function Overloading

- Function Overloading
  - You are writing a function to divide two numbers.
    - If the two numbers are floats, just divide as normal.
    - If the two numbers are integer, divide them but return the result rounded using the floor() function.

Example – divideUs

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

const int MAX_STRING_LEN = 256;

float divideUs( float num1, float num2 )
{
  cout << "divideUs - float version called" << endl;
  float returnValue = num1 / num2;
  cout << "         - " << num1 << " / " << num2 << " = " << returnValue << endl;
  return returnValue;
}

int divideUs( int num1, int num2 )
{
  cout << "divideUs - integer version called" << endl;
  int returnValue = (int)floor( (float)num1 / (float)num2 );
  cout << "         - " << num1 << " / " << num2 << " = " << returnValue << endl;
  return returnValue;
}
```

Example – divideUs

Calling divideUs:

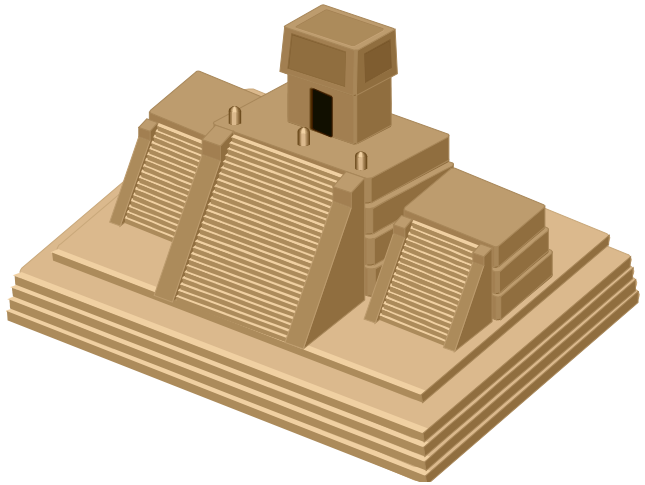
```
main()
{
  char aString1[ MAX_STRING_LEN ];
  char aString2[ MAX_STRING_LEN ];

  cout << "Enter two numbers seperated by a space: ";
  cin  >> aString1 >> aString2;

  float floatNum1 = (float)atof( aString1 );
  float floatNum2 = (float)atof( aString2 );
  float floatResult = divideUs( floatNum1, floatNum2 );

  int intNum1 = (int)floatNum1;
  int intNum2 = (int)floatNum2;
  int intResult = divideUs( intNum1, intNum2 );
}
```

Classes

Before looking at classes, lets look at the `struct` data type in C

What is a struct?

- A struct is a special data type that can contain multiple variables
- It is usually used to organize related variables together



Classes

What is a struct?

- A struct data type might be created for a car
- The car struct would contain data related to a car
- Some of these fields of the car might be:
  - Current Odometer
  - Blue Book Value
  - Last oil change date
  - Life time service cost
  - Original Price
  - Owner's name

Declaring a Struct

```
struct Car
{
  float odometerReading;
  float blueBookValue;
  long  lastOilChangeDate;
  float lifeTimeServiceCost;
  float originalPrice;
  char  ownerName[MAX_STRING_LEN];
};
```

Using a Struct

Creating a struct instance

- Create an instance of a struct on stack
  - Car myCar;
- Create an instance of a struct on heap
  - Car *myCar = new Car;
- Delete instance of a struct from the heap
  - delete Car;

Variables allocated on the stack are automatically de-allocated when they go out of scope.

Variables allocated in the heap need to be explicitly de-allocated.

Heap vs. Stack

When a computer program executes, it must allocate memory for variables.

The run-time memory of a program is made of three subdivisions:

- Code
- Stack
- Heap

Typically memory can be allocated on either the stack or in the heap.

In most high level languages, the details are taken care of for you by the compiler, but you need to be aware of the implications

Heap vs. Stack

Stack:

- Variables allocated locally are pushed onto this stack
- When functions are called their parameters are pushed onto this stack, as well as any variables they allocate
- When function exits, its variables are popped off the stack
- Stack has important implications for variable scoping
- On the x86 platform, stack variables can be accessed quickly
- Typically there is a size limit to the stack, so do not want large variables to be allocated here.

Heap vs. Stack

Heap:

- Variables allocated in heap remain there even after the function that allocated it exit
- Variables allocated here must be explicitly de-allocated, else memory “leaks”
- Some run-times include a Garbage collector to automatically de-allocate space.
- Generally, the heap can be much larger than the stack. Allocate large arrays and large classes here
- Memory fragmentation can be an issue. Some run-times include a heap manager that can take care of this

Heap vs. Stack

So what does all this mean?

- Variables declared locally are removed from memory when they leave their block of code
- When passing variables to functions, they are allocated on the stack. Pass large variables as pointers to avoid stack bloating.
- If you have a large array or class, declare it on the heap using either malloc() or new
- If you use malloc() or new, you are responsible for releasing the memory with free() or delete

Using a Struct

Structs are variables, thus we will have to access them either as variables or pointers

The fields of a struct are accessed differently if the struct is a variable or a pointer

The members of a struct can be accessed by either the '.' or '->' operators.

Using a Struct

If you are accessing an instance of the struct, use '.'

```
myCar.originalPrice = "75000.00";
```

If you have a pointer to the instance of the struct, use '->'

```
myCarPtr->originalPrice = "75000.00";
```

Struct

Passing a struct to a function

```
void printMileage( Car car )
{
  cout << "Meleage is " << car.odometerReading << endl;
}

main()
{
  Car myCar;
  myCar.odometerReading = 20000;
  printMileage( myCar );
}
```

Struct

Passing a struct as a pointer to a function

(because some structs can get quite large, it is preferable to pass them as pointers)

```
void printMileage( Car* car )
{
  cout << "Meleage is " << car->odometerReading << endl;
}

main()
{
  Car myCar;
  myCar.odometerReading = 20000;
  printMileage( &myCar );
}
```

Struct

What if a struct could contain not only variables, but functions that are useful to those variables.

Useful Functions in our Struct might be:

- check if it is time for an oil change
- determine if the car is a lemon

Struct

In C++ Structs can contain functions. When a function is contained it is usually called a method of that entity.

Useful Functions in our Struct might be:

- check if it is time for an oil change
- determine if the car is a lemon

Struct

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

const int MAX_STRING_LEN = 128;

struct Car
{
  float odometerReading;
  float lifeTimeServiceCost;
  float originalPrice;
  char  ownerName[MAX_STRING_LEN];
  // a function in our struct:
  bool isCarALemon()
  {
    return ( lifeTimeServiceCost > originalPrice );
  }
};

void main()
{
  Car aCar;
  char inputString[MAX_STRING_LEN];

  cout << "Owner's name: ";
  cin >> aCar.ownerName;
  cout << "How much did the car cost " << aCar.ownerName << " ?\n$";
  cin >> inputString;
  aCar.originalPrice = atoi( inputString );
  cout << "How much has " << aCar.ownerName << " paid for vehicle service?\n$";
  cin >> inputString;
  aCar.lifeTimeServiceCost = atoi( inputString );
  cout << aCar.ownerName << "'s car is " <<
    ( aCar.isCarALemon() ? "a lemon" : "okay" ) << endl;
}
```

Object Oriented Development

What is a class?

- A class is a collection of attributes and behaviors.
- A well designed class has a single purpose.
- An instance of a class is an object.

Classes

A `class` is (essentially) a `struct`

They are the essentially the same data type, but "typically" a struct is a data record, while a class is an encapsulated entity that can be interacted with as an object.

Members of a struct are public by default, while members of a class are private by default.

Because you already know how to create a struct, you know the basics of how to create a class.

Classes

Classes can have public and private members

Design a class so that it only ***public***ly provides access to members that are needed to use the class.

Members of a class that are not needed to use the class should be ***private***.

Members of a class are accessed just like a struct using '.' or '->'

Struct becomes a Class

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

const int MAX_STRING_LEN = 128;
//struct Car
class Car
{
public:
  float odometerReading;
  float lifeTimeServiceCost;
  float originalPrice;
  char  ownerName[MAX_STRING_LEN];
  // a function in our struct:
  bool isCarALemon()
  {
    return ( lifeTimeServiceCost > originalPrice );
  }
};

void main()
{
  Car aCar;
  char inputString[MAX_STRING_LEN];

  cout << "Owner's name: ";
  cin >> aCar.ownerName;
  cout << "How much did the car cost " << aCar.ownerName << " ?\n$";
  cin >> inputString;
  aCar.originalPrice = atoi( inputString );
  cout << "How much has " << aCar.ownerName << " paid for vehicle service?\n$";
  cin >> inputString;
  aCar.lifeTimeServiceCost = atoi( inputString );
  cout << aCar.ownerName << "'s car is " <<
    ( aCar.isCarALemon() ? "a lemon" : "okay" ) << endl;
}
```

Classes

Programmers like to keep their class definitions clean so that they are easy to read.

This includes implementing functions outside of the class definition.

This is done by prototyping the function in the class, and implementing it outside.

The outside implementation is done by “scoping” (the :: operator) the function name within the class name:

```
class ClassName
{
  public:
    int someIntVariable;
    bool methodName( int anIntValue ); // this is just a
                                       //  regular protoype
};

// implement our function outside the class definition
bool ClassName::methodName( int anIntValue )
{
   // do something meaningful
   return true;
}
```

Classes

```
class Car
{
  public:
    float odometerReading;
    float lifeTimeServiceCost;
    float originalPrice;
    bool isCarALemon(); // this is now just a
  prototype
};

bool Car::isCarALemon()
{
   return ( lifeTimeServiceCost > originalPrice );
}
```

Multiple Files in VC++

Although you can define a class, implement it, and use it in one file, this is not generally a good idea.

Simple class can be done this way, but with more complicated class it can lead to a source file that is difficult to maintain and will cause great frustration.

C++ programmers usually define their class in a header file, implement it in a source file, and reference from where they are invoking it.

MyClass.h

(header file)

Defines MyClass

MyClass.cpp

(source file)

#include “MyClass.h”

Implements MyClass

MyProgram.cpp

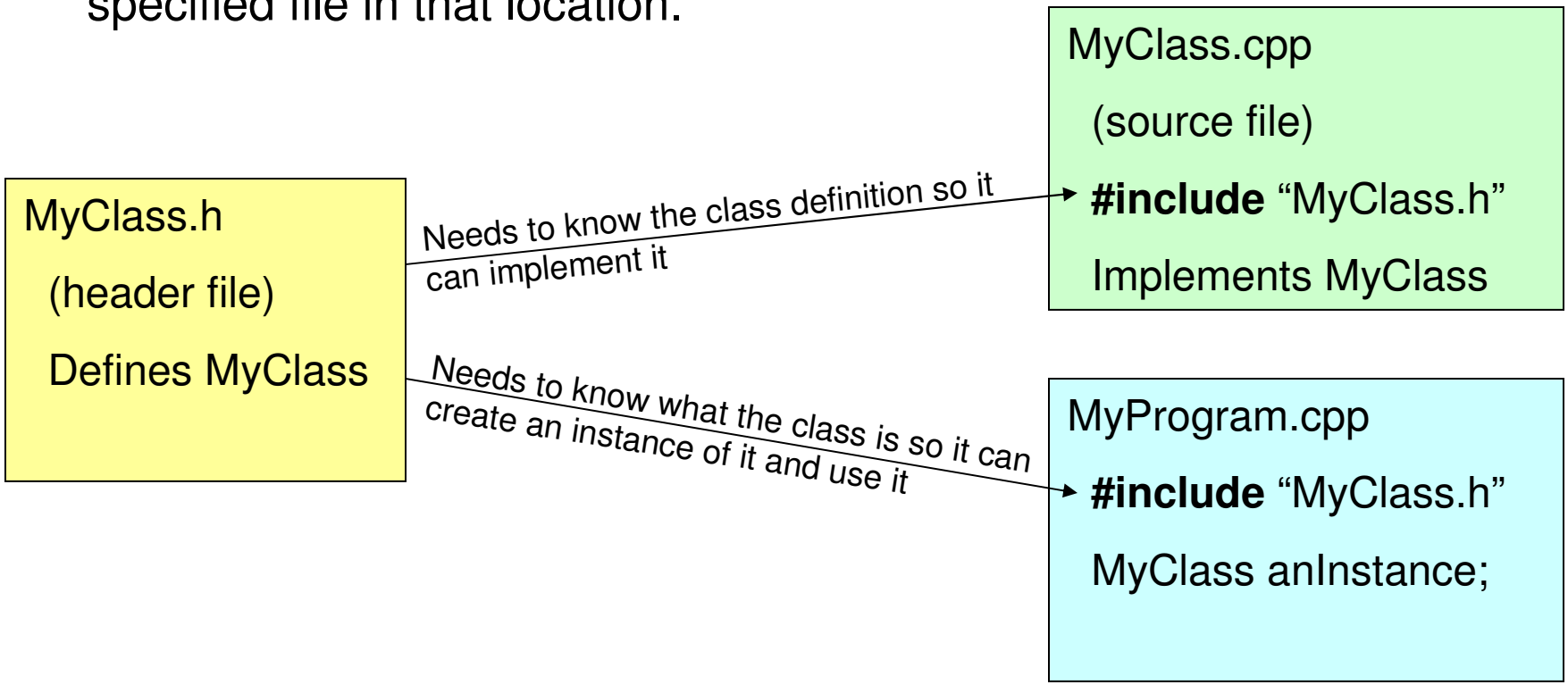
#include “MyClass.h”

Car aCar;

Multiple Files in VC++

#include

The #include directive tells the compiler to include the text from the specified file in that location.



Multiple Files in VC++

CarInfo.h

```
const int MAX_STRING_LEN = 128;
class CarInfo
{
public:
  float odometerReading;
  float lifeTimeServiceCost;
  float originalPrice;
  char  ownerName[MAX_STRING_LEN];
  bool isCarALemon();
};
```

CarInfo.cpp

```
#include "CarInfo.h"
bool CarInfo::isCarALemon()
{
  return ( lifeTimeServiceCost > originalPrice );
}
```

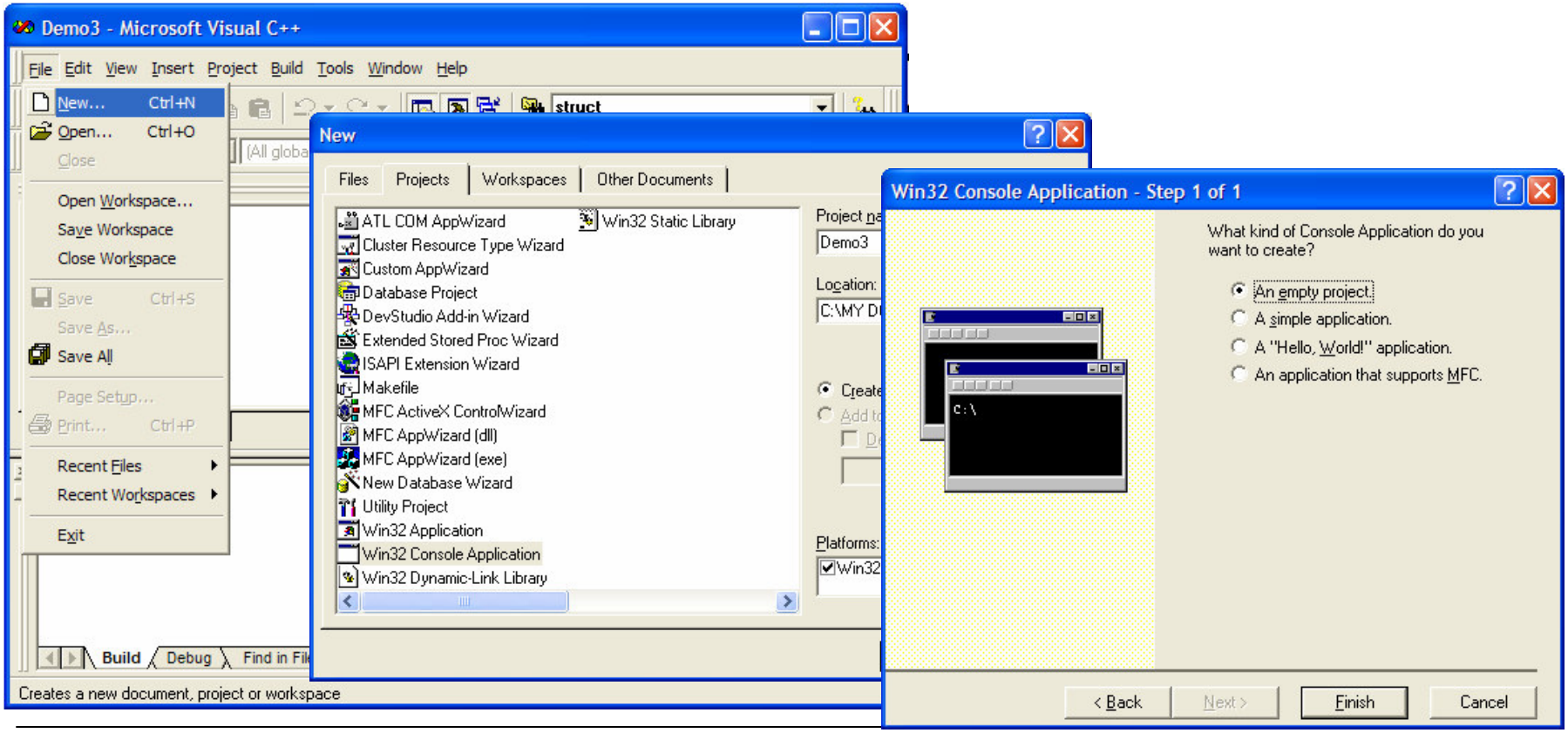
MyProgram.cpp

```
#include <iostream>
#include "CarInfo.h"
using namespace std;
void main()
{
  CarInfo aCar;
  char inputString[MAX_STRING_LEN];
  cout << "Owner's name: ";
  cin >> aCar.ownerName;
  cout << "How much did the car cost " <<
            aCar.ownerName << " ?\n$";
  cin >> inputString;
  aCar.originalPrice = atoi( inputString );
  cout << "How much has " << aCar.ownerName <<
            " paid for vehicle service?\n$";
  cin >> inputString;
  aCar.lifeTimeServiceCost = atoi( inputString );
  cout << aCar.ownerName << "'s car is " <<
    ( aCar.isCarALemon() ? "a lemon" : "okay" ) << endl;
}
```

Multiple Files in VC++

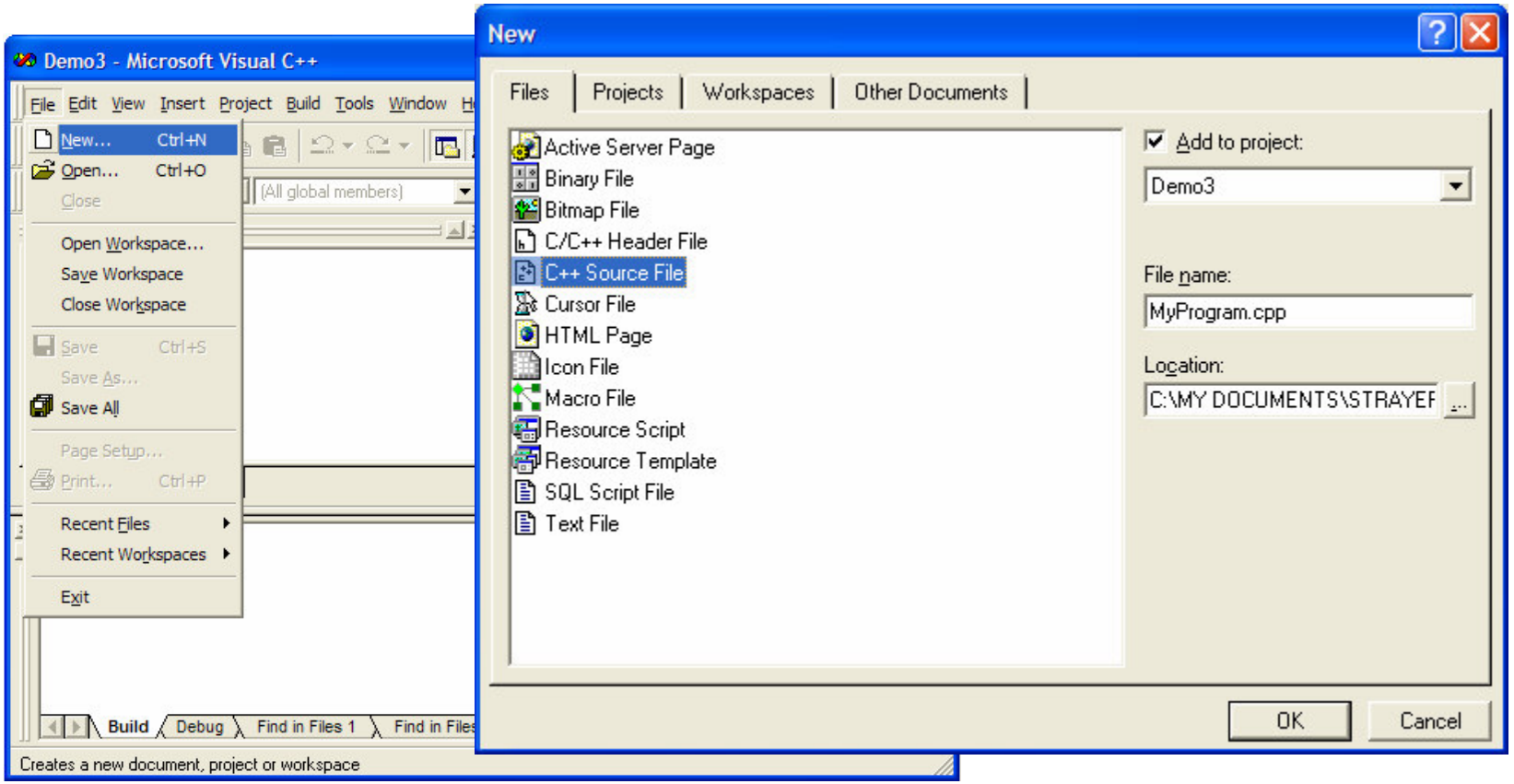
Actually creating these multiple files in VC++ 6.0

Create a new project as usual:



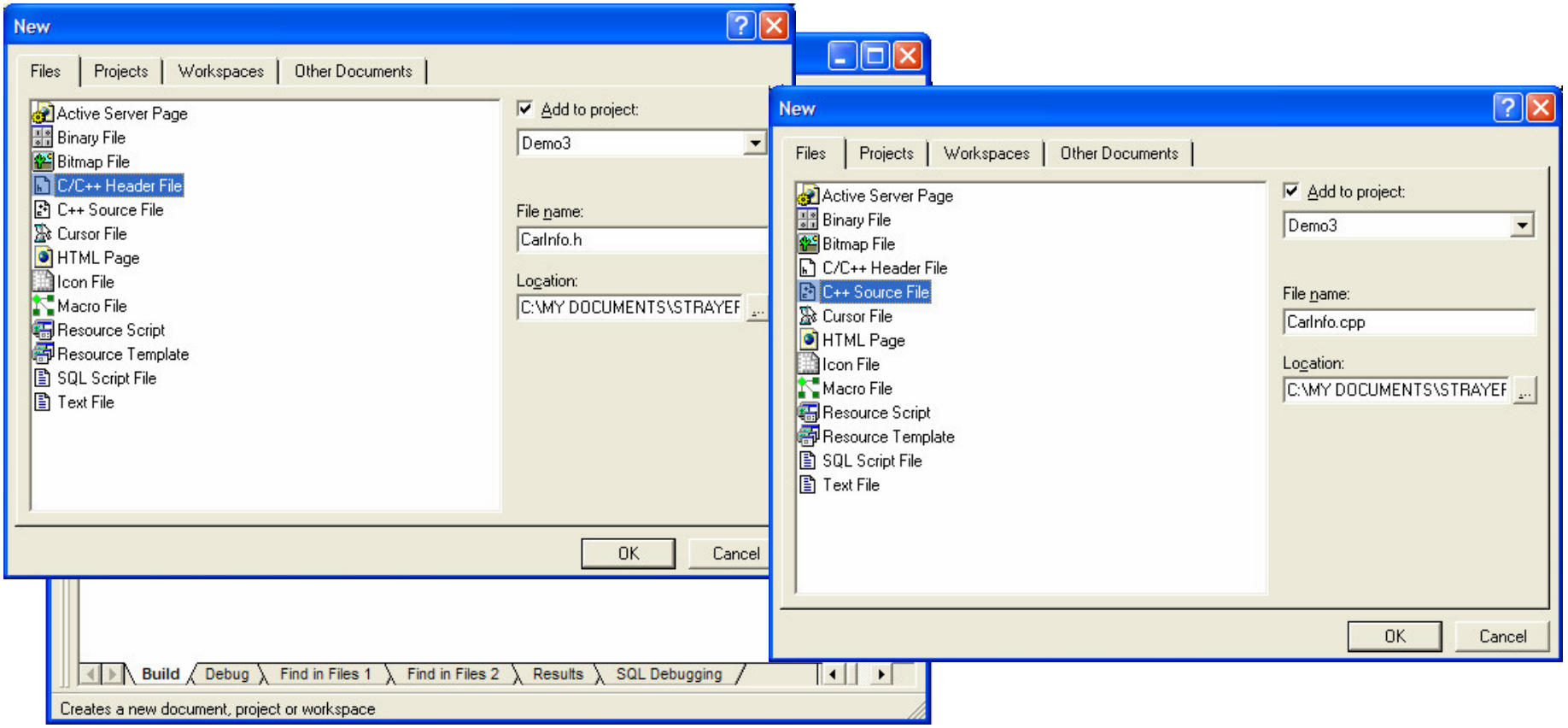
Multiple Files in VC++

Add a source file, as usual:



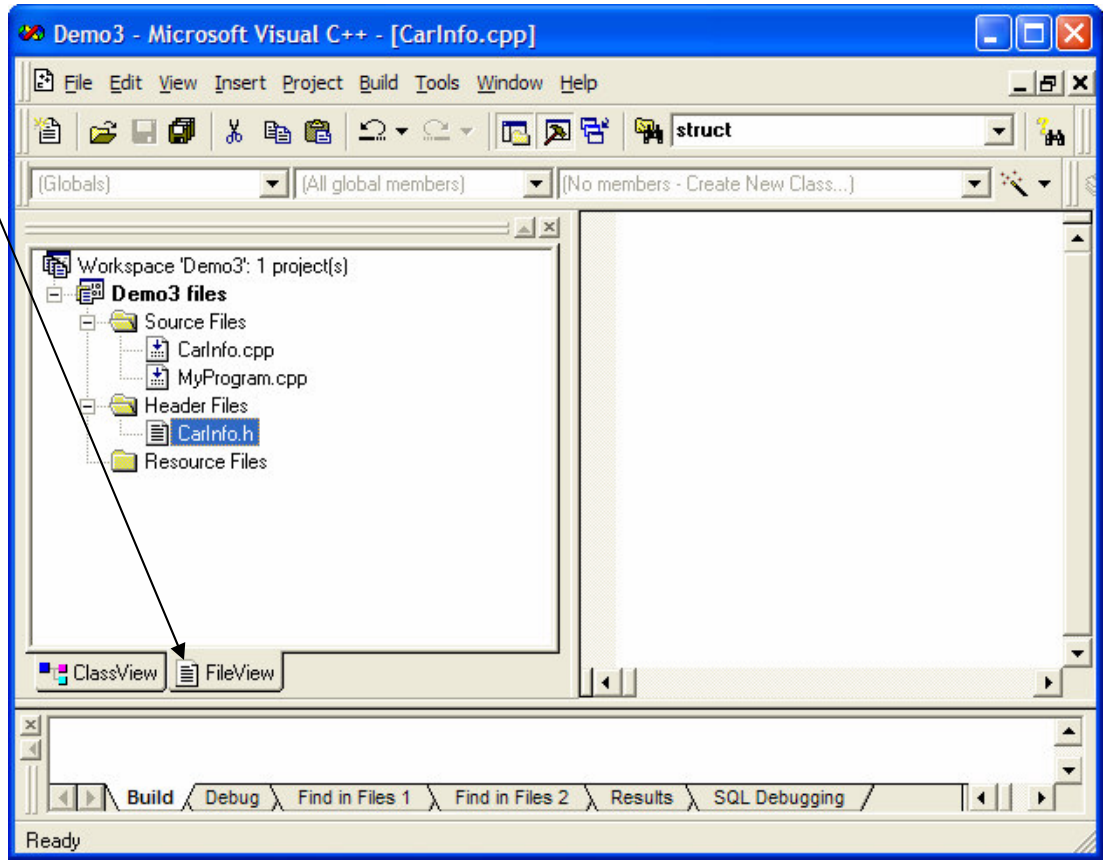
Multiple Files in VC++

Then add a header file and a matching source file:



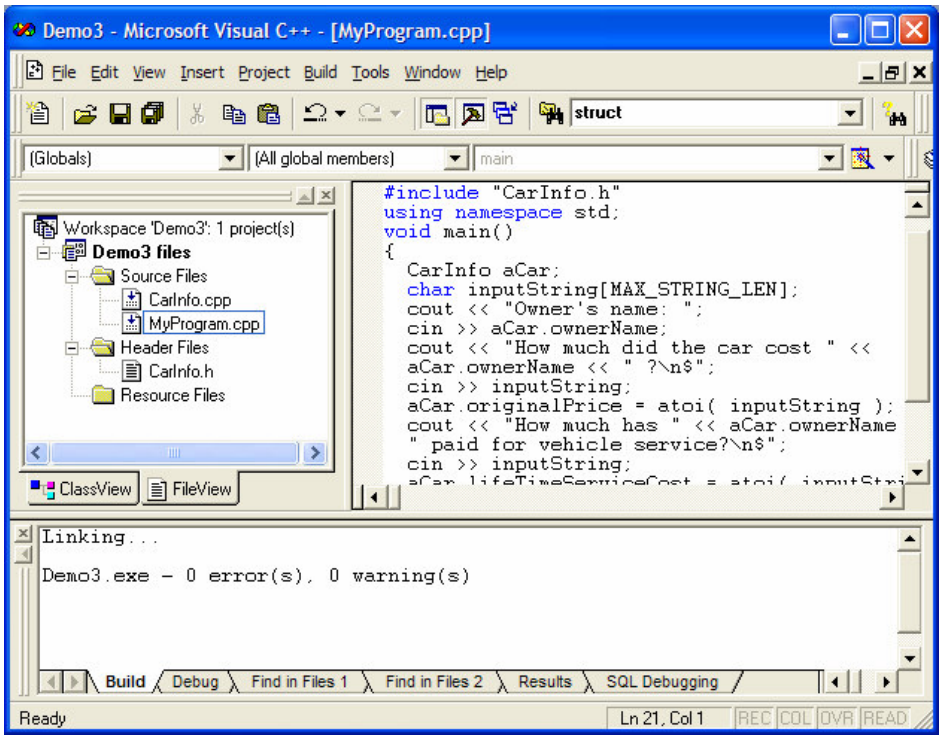
Multiple Files in VC++

The file view for your project should look like:



Multiple Files in VC++

Add code to your files (class definition in class header file, class implementation in class source file, and main program in main source file)



Multiple Files in VC++

The Class View for your project also shows the attributes and methods of your class(es)