

COMP370
LABORATORY SCHEDULE AND EXPERIMENTS

INSTRUCTOR: Tom Wheeler (Office in Room 208) 941-0430 x5211
twheeler@kc.devry.edu (DeVry e-mail address)
<http://www.kc.devry.edu/homepages/twheeler>

The work in this laboratory reinforces the concepts covered in *COMP370, Software Design with OOP/C++*. Each project teaches a new feature of the C/C++ language.

<u>PROJECT</u>	<u>DESCRIPTION</u>	<u>WEEK # DUE</u>
1	FAMILIARIZATION	3
2	POWER CALCULATIONS	5
3	DEFINING A CLASS	7
4	CONSTRUCTORS AND DESTRUCTORS	9
5	AN OBJECT-ORIENTED DATABASE	11
6	DERIVING A NEW C++ CLASS	13
7	MFC PROJECT (TO BE ANNOUNCED)	15

LABORATORY REPORT CONTENTS

Every person will turn in a complete lab report for each experiment performed in COMP370. The content must be as follows:

a) Cover Page -- Must be computer-generated. Contains:

YOUR NAME
COMP370
LAB NUMBER AND TITLE
TODAY'S DATE
DATE DUE (WEEK #)

FOR: PROFESSOR WHEELER

OPERATIONAL SIGN OFF _____
FINAL SIGN OFF _____

"THIS IS THE ORIGINAL WORK OF (your name)"
(YOUR SIGNATURE)

HINT:

Make sure to have a completed cover page when you start each lab. No sign-off can be given without a proper cover page.

ABOUT SIGN-OFFS:

A sign-off will be given only during the scheduled laboratory period. The sign-off verifies that your program works correctly. No credit is given to any report lacking a sign-off. You must present your program listing and demonstrate correct operation to obtain a sign-off. Your name must be at the top of all program listings.

- b) **End-User Documentation** -- This should be a complete description of how to operate your program from an end-user's point of view. More information on this topic is given in the section "End-User Documentation."
- **Sample Run** -- This is a printout of a "typical" run of your program. To get a "screen dump" of your program, do the following:
 1. Let it run in an MS-DOS window. Run the program.
 2. Click on the *system* menu (MS-DOS icon at top-left of the blue window frame).
 3. Choose *Edit->Mark*, and highlight the text you wish to capture. Press the *RETURN* key to copy the highlighted text to the clipboard.
 4. Paste the highlighted text into the desired word-processing application.

Note: If you can't see the system menu, make sure your application is not running in full-screen mode. Press ALT-ENTER to switch back to a window if needed.

- c) **Program Listing** -- A printed listing of your program. Your name must be at the top of all program listings. The program will be properly documented and commented throughout. See "Program Documentation."

TURNING IN WORK

Work may only be turned in directly to the instructor, or his appointed representative, during the laboratory period. Do not turn in papers to room 208.

GRADING

Each report in COMP370 is worth 100 points. 7 reports are required, which produces a raw total of 700 points possible. This is converted into a percentage (your total point score is divided by 7) for inclusion into the total course grade. (See the COMP370 syllabus for details).

Therefore, the labs contribute 100 points of the 500 points possible in COMP370.

Laboratory Report Grading breakdown:

End-user documentation	40 points
Overall neatness of entire report	30 points
Accuracy and readability of listing	30 points

	100 points total

End-user documentation will be graded based on clarity, grammar, completeness, and neatness.

The program listing is evaluated for both accuracy and adherence to certain stylistic conventions including (but not limited to):

- Properly written *comment block* or "header" for each function.
- Consistent and readable use of *indentation*.
- *Comments* for each major variable function in program.
- Use of *structured-programming* techniques (where appropriate), and the adherence to *naming conventions* for variables and other objects.

Course Policies

I. Lab Partners: There are *no* lab partners allowed in COMP370.

II. Handing Work in: Work should be given directly to the instructor or his authorized assistant. Under no circumstances should work be turned in to any other persons (including the office) without advance permission from the instructor.

III. Late Work: Late laboratory reports are not accepted in COMP370L. Reports are due *during or before the end* of the laboratory of the week number indicated on the schedule on page 1 of this document. *The laboratory period ends at xx:50 UTC of the second hour of the assigned period.* (UTC=Coordinated Universal Time, Standard World Time).

IV. Lab Success Hints: The successful student will have worked through the majority of the source code before entering lab. Laboratory time should be used to proper advantage; this is the main time that the instructor will be available to assist in troubleshooting and debugging, and it is the only time that sign-offs will be available. Please plan your activities accordingly.

V. Plagiarism: *Copying the work of another, and claiming it to be your own is plagiarism.* This includes (but is not limited to) copying others homework, copying from a lab manual or textbook, or collusion. The minimum penalty for cheating in any form is a grade of zero for the element involved; in some cases, failure of the course and/or expulsion from the Institute will also result. All cases of misconduct will be documented and forwarded to Student Services for disciplinary consideration. The DeVry Student Handbook contains complete information on this topic.

Please do not turn in any work that is not your own! If in doubt, ask the instructor. Here are some ways to avoid any problems:

- Don't share your computer files (text, C/C++ source/object, etc) with anyone else.
- Don't share a diskette (or other media) with another student; it's too easy to get files mixed up.
- Don't copy answers from a neighbor. If you don't understand how to do it, ask!
- Decline any request from fellow students for a copy of your work. Anybody needing this level of help should ask the instructor.

MISCELLANEOUS INFORMATION

Emergency Procedures: There are plaques located in the lab discussing emergency procedures. The instructor will remain in charge of your class group in an emergency.

Food and Drink: Are not allowed in the laboratory at any time, even in closed containers. Violators will be expelled from the laboratory.

GOOD DATA PROCESSING PROCEDURES

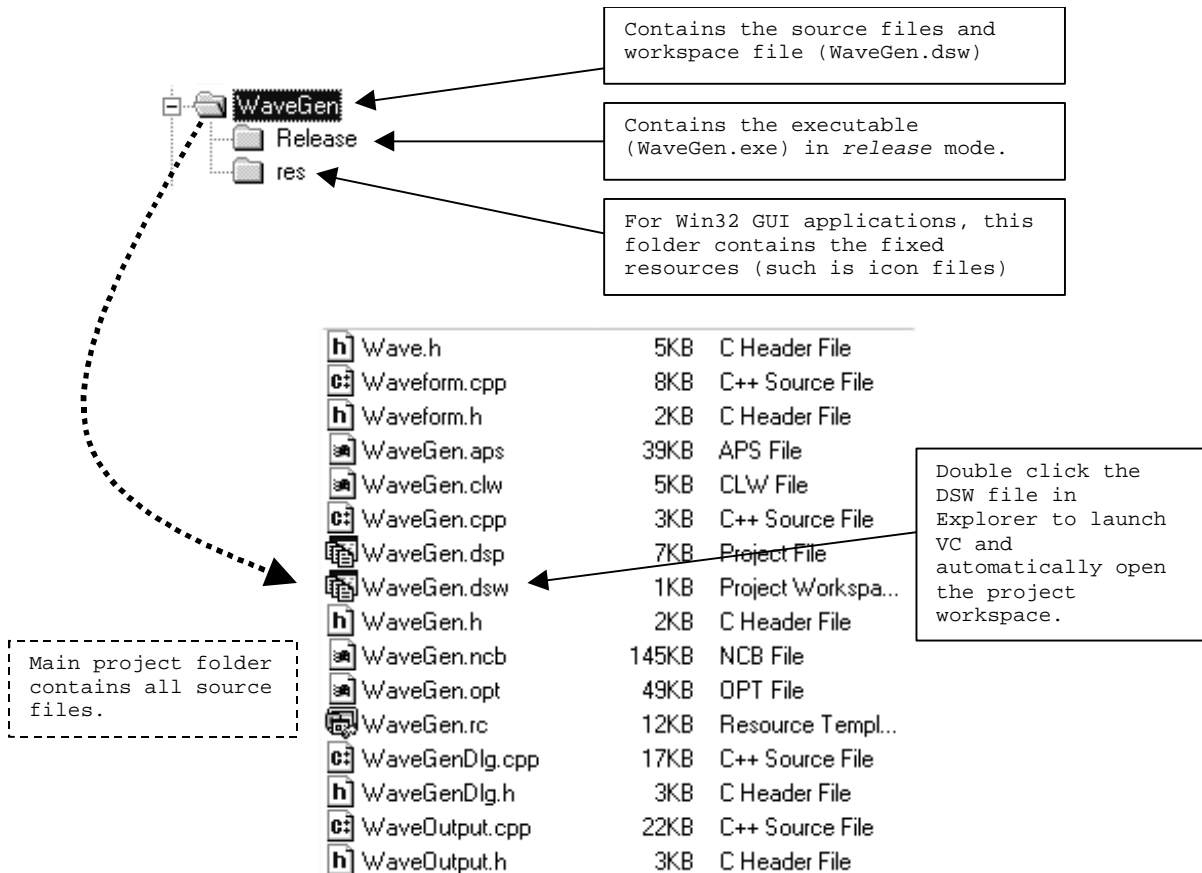
Computers will be used extensively in this lab. The student can expect to spend many hours creating and updating C programs; loss of this data can be disastrous! The following tips will help to minimize the chance of losing a project:

- Make frequent backups. These backups should be in at least two different physical locations.
- Always keep schoolwork on two different diskettes. Both of these disks will contain identical information. If a computer damages one diskette, the data can still be recovered from the other during the lab period.
- Don't save your data to the hard disk on the workstation, except in an emergency. The hard disks on lab workstations are periodically "cleaned" of any extra information as part of a housekeeping program.
- Keep the work for each class on a separate disk.
- Write your name, course, section, and professor's name on each disk. This will make it easier for others to return your work to you should you accidentally leave a disk behind. It happens to all of us!
- If you're using a computer at home, an *anti-virus* program is strongly recommended.
- Remember that at some time in the future, your C program will be dangerous to the general health and well being of your computer. An errant C program can easily wipe a hard disk clean, and in some cases, can damage the hardware. Be guided accordingly. If you execute a C program and the screen blanks unexpectedly, and you hear any whistling, arcing, or other suspicious sounds from the computer or video display monitor, immediately turn off or reset the machine!

VISUAL C TIPS

Visual C++ is intended to be used by a single user who will be storing all his or her source files directly on the hard disk. Attempting to compile a project directly from a floppy diskette is not advisable.

Visual C++ programs are written as *projects*. A project contains all the source and header files you have created, plus some supporting information used by the compiler. For example, the project "WaveGen" is arranged like this:



The main project folder, *WaveGen* contains all the source (*.CPP) and header (*.H) files for your project.

When you tell VC to build (F7) your project, it will create a new subfolder called either *Release* or *Debug* (depending on your *Project...* settings). The *Release* folder contains the object file(s) for your program, a rather large PCH (precompiled header) file, and the executable version of your program. The *Debug* folder contains all of this, plus specific information for the symbolic debugger. It is not unusual for 10 MB of data to accumulate in these folders, however, you can safely delete these folders when it is time to save your project back to a floppy disk.

Working with a Project in Lab

When you begin working with a VC project in lab, copy the entire project folder from your floppy or ZIP disk to drive C (hard drive) and work with the project as you would on your own machine.

When you build a VC project, VC will fill the *Release* (or *Debug*) folder with files much too large to be held on a floppy. Eventually, you will need to save your source code and project information onto some type of obsolete media (such as a floppy disk, 1/2" tape, audio cassette, etc...). To do so:

1. Delete the *Release* and *Debug* sub-folders. (If you wish to preserve the executable program file, copy it before doing this.)
2. Using *Windows Explorer*, copy the project main folder back onto the floppy disk.
3. Delete the project folder you created on the hard disk to prevent others from obtaining your work.

All but the largest of projects can be preserved on a floppy disk using this method.

About VC Projects

Visual C++ is a project-oriented environment. It can not compile individual C++ source code files unless they are part of a project. For large and complicated programs, this is a great advantage. It allows the programmer to concentrate on the details of what he or she is doing, because VC will keep an eye on everything in the project at build time, ensuring that everything is put together right.

You can create a project in several ways. First, you can do it from the *File* menu (choose New, Project and follow the prompts.) Once you have created the project, you will have to inform VC to add the desired C++ source/header files. Second, you can force VC to open a C++ source file by dragging and dropping the C++ source file on top of the Visual C++ program icon. When you attempt to build (F7) with the open C++ source file, VC will issue a warning that no project exists, and will ask you if you want a default workspace created. Answer "yes" and a workspace will be built, and the source file will be compiled. This last method is not very clean, but it works in a pinch.

Source (.CPP) and Header (.H) Files

When you create a multi-file project, certain things can get you in trouble very quickly. It is best to let Class Wizard create new files whenever possible. In a multi-file project, be careful to avoid any generation of *code* or *data* objects within a header file. Header files should contain class and constant definitions, and nothing else. Failure to observe this may lead to pattern baldness!

END-USER DOCUMENTATION

End-user documentation is the set of information that enables a relatively uninformed second party to properly operate your program. There are several points which should be addressed, including:

- a) Purpose of the Program. This should be a short summary of what results a user can expect from running your program.
- b) Using the Program. Should have several basic pieces of information, including but not limited to:
 - What Operating System and Computer are required? (Are there any special hardware or software requirements?)
 - How is the program started? (What command is used, or what icon is clicked?)
 - What inputs are required by the program? (What information must be provided by the user to operate the software, and in what order?)
 - What outputs are generated by the program? (How will it look on the screen?)
 - How does the program respond to erroneous user input? (How fault-tolerant is the program?)

For reports in COMP370, the first item need not be addressed, as all students will be working under the same computer platform. Most laboratory projects will yield documentation that is less than one page in length. Keep documentation simple and concise. **Your documentation must "walk" the user through a session with the software, showing all on-screen prompts, user inputs, and program responses.**

Actual applications programs tend to be involved efforts, and as such, documentation becomes a much larger (and much more important) task. When writing documentation, always think of the uninformed end-user.

End-User Documentation "Dos and Don'ts"

DO

- Walk users through the operation of the program, showing prompts and responses as they'll actually appear.
- Include appropriate figures and illustrations to show how your program applies to a problem.
- Use a spelling and/or grammar checker to proof your work.

DON'T

- Assume that the end-user knows how your program should operate. (Even technical users can be intimidated by a new piece of software.)
- Use a spell-checker as a substitute for proof reading. ("Eye awl weighs use my spell check ere.")

PROGRAM DOCUMENTATION

Program documentation is information that is directed at a technical person who may be required to maintain your code. Later, someone (possibly you!) will most likely have to make corrections or additions to your program. The quality of program documentation (and the basic structuring of your code) largely determines how easy (or difficult) this task will be.

Documentation is generally done in at two forms. For involved projects, the **flowchart** is an excellent method for capturing the train of events inside a program. The flowchart gives a conceptual, "bird's eye" view of what is happening. Flowcharts should avoid specific details such as "increment the R register" unless such detail is necessary to understanding the overall action of the program. A more suitable comment might be "Add \$1.00 to account balance," since this directly states what the desired effect is. In other words, flowcharts are more concerned with *what* is happening rather than *how* the machine mechanically achieves it.

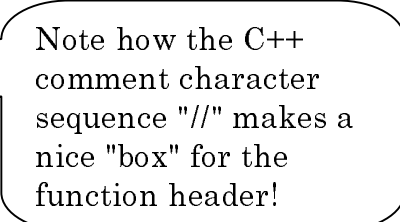
The second form of documentation is the **program listing**. The program listing is a detailed explanation of the mechanics behind the execution of a task. It should be *appropriately* and *thoroughly* commented. In "C" programs, blank lines should be inserted to separate parts of a task that are logically separate, and indentation should be used where appropriate. Consider the two pieces of code below:

```
// This poorly-styled code actually works!
int my_function(x,y)
int x,y;
{int alpha,z;alpha=0;
  for(z=1;z<10;z++) {alpha=alpha+particle(x,y*y,z);y--;}
return(alpha);}
```

This code will compile and execute just fine, but it's difficult for humans to understand and maintain. The purpose of this code (and how it fits into the context of the whole program) is impossible to determine by mere inspection. It would be easier to understand if it were written like this:

```
////////////////////////////////////
// Function: int my_function(int x,int y)
//
// Inputs: integers x and y.  x is the RMS mass of the particle and y
//         is its current velocity in Kilometers/uS.
// Returns: an integer with an approximation of the deceleration of the
//         particle in Kilometer/uS^2
//
// Errors: There are no error conditions possible for
//
////////////////////////////////////

int my_function(int x,int y)
{
int alpha,z;
```



Note how the C++ comment character sequence "//" makes a nice "box" for the function header!

```

alpha=0;    // initial acceleration is assumed to be zero.

/* Compute the approximate acceleration using BIG AL's iterative method.
   Reference: CAR AND DRIVER magazine, July 1990, pp 34-35. */

for(z=1;z<10;z++)
{
    alpha=alpha+particle(x,y*y,z);
}
return(alpha);
}

```

Consistent use of indents improves readability.

It's OK to mix C and C++ comment styles, as long as you're consistent. Note how *each algorithm is explained to the reader*.

Note that every instruction in a program listing isn't necessarily commented. Machine instructions, in and of themselves, are generally self-evident because of the use of mnemonics. The documenting comments therefore serve to illuminate individual ideas or concepts that are important for understanding *how* the machine is accomplishing the task at hand.

Every function you define should have a *header* comment. This important field describes what the function does, what inputs it uses, and what outputs it produces. It should be written as if you were explaining how to use the function to another programmer.

Appropriate placing of comments, program elements, and use of indentation all help to make source code more easily understood.

PROJECT DESCRIPTIONS

For a program to be accepted, it must meet the minimum specifications for each project. Extra functionality, as a function of your own creativity, is encouraged, as long as it does not detract from the original purpose of the project (or make it difficult to evaluate or analyze.)

In most cases, the *inputs* and *outputs* of the program will be specified. Your program should have the exact inputs and outputs that we specify, otherwise it becomes difficult for us to check your program's operation.

PROJECT 1: FAMILIARIZATION

Please read pages 1-33 of *Beginning Visual C++ 6* (available in the Library) before starting. Bring the textbook to lab. You may need to refer to it as you perform the steps in this experiment.

Let's get familiar with Visual C++. The best way to do this is to build a working Win32 console application:

1. Launch Visual C.
2. From the *File* menu, choose *New* and create an empty Win32 console application. (Make sure to select the PROJECT tab in the dialog box.) Call your project MYPROJECT.
3. From the *Project* menu, select *Add to project* and *New...* , and select "C++ Source File." Call the new file MYFIRST.CPP. Make sure the "Add to Project" box is checked.
4. Again from the *Project* menu, add a new component to your project, a "C/C++ Header File." Call this file MYFIRST.H.
5. From the *Project* menu, select *Add to project* and *New...* , and select "C++ Source File." Call the new file MYMAIN.CPP.

6. Add the following code to MYFIRST.H:

```
//  
// Myfirst.h -- Definition of the CTriangle class  
//  
class CTriangle  
{  
public:  
  
// constructor  
CTriangle( double Base = 3, double Height = 4, double Hyp = 5 );  
  
// destructor  
~CTriangle();  
  
// Public methods  
double GetArea();  
void SetBase(double Base);  
void SetHeight(double Height);  
void SetHyp(double Hyp);  
double GetBase();  
double GetHeight();  
double GetHyp();  
  
int GetNumberOfTriangles();  
  
private:  
  
double m_Base, m_Height, m_Hyp;  
  
static int m_nNumberExisting;  
};
```

7. Add the following code to MYFIRST.CPP:

```
#include <math.h>
#include "myfirst.h"
//
// Myfirst.h -- implementation of the CTriangle class
//
int CTriangle::m_nNumberExisting = 0;

double CTriangle::GetArea()
{
return( m_Base * m_Height / 2. );
}

void CTriangle::SetBase(double Base)
{
m_Base = Base;
}

void CTriangle::SetHeight(double Height)
{
m_Height = Height;
}

void CTriangle::SetHyp(double Hyp)
{
m_Hyp = Hyp;
}

double CTriangle::GetBase()
{
if (m_Hyp >= m_Height)
    m_Base = sqrt( m_Hyp * m_Hyp - m_Height * m_Height );
return(m_Base);
}

double CTriangle::GetHeight()
{
if (m_Hyp >= m_Base)
    m_Height = sqrt( m_Hyp * m_Hyp - m_Base * m_Base );

return(m_Height);
}

double CTriangle::GetHyp()
{
m_Hyp = sqrt( m_Base * m_Base + m_Height * m_Height );

return(m_Hyp);
}

int CTriangle::GetNumberOfTriangles()
{
return(CTriangle::m_nNumberExisting);
}

// Constructor

CTriangle::CTriangle( double Base , double Height , double Hyp )
{
CTriangle::m_nNumberExisting++;

m_Base = Base;
m_Height = Height;
m_Hyp = Hyp;
}

// Destructor
CTriangle::~CTriangle()
{
m_nNumberExisting--;
}
```

8. Add the following code to MYMAIN.CPP:

```
#include <stdio.h>
#include "myfirst.h"

void main(void)
{
    CTriangle Triangle1;

    printf("%d Triangles exist\n", Triangle1.GetNumberOfTriangles() );

    printf("Triangle 1 area is %lf Units^2\n\n",
           Triangle1.GetArea() );

    printf("Triangle 1 base is %lf\n", Triangle1.GetBase() );
    printf("Triangle 1 height is %lf\n", Triangle1.GetHeight() );
    printf("Triangle 1 hypotenuse is %lf\n", Triangle1.GetHyp() );

    CTriangle Triangle2(10., 11., 12. ); // impossible triangle
    printf("%d Triangles now exist\n", Triangle1.GetNumberOfTriangles() );

    printf("Triangle 2 base is %lf\n", Triangle2.GetBase() );
    printf("Triangle 2 height is %lf\n", Triangle2.GetHeight() );
    printf("Triangle 2 hypotenuse is %lf\n", Triangle2.GetHyp() );

    printf("Triangle 2 area is %lf Units^2\n\n",
           Triangle2.GetArea() );

    printf("Changing triangle2 base to 5 units.\n");

    Triangle2.SetBase(5.);

    printf("Triangle 2 area is now %lf Units^2\n\n",
           Triangle2.GetArea() );

    printf("Triangle 2 hypotenuse is now %lf\n", Triangle2.GetHyp() );

    CTriangle* pTriangle3 = new CTriangle(1.,2.);

    printf("Triangle 3 hypotenuse is %lf\n",
           pTriangle3->GetHyp() );

    printf("%d Triangles now exist\n", Triangle1.GetNumberOfTriangles() );

    printf("Deleting Triangle3\n");
    delete pTriangle3;

    printf("%d Triangles now exist\n", Triangle1.GetNumberOfTriangles() );
}
}
```

9. Build the project (F7) and execute it (CTRL-F5). Congratulations! You've just run your first OOP program under Visual C!

PROJECT 2: POWER CALCULATIONS

This project is our last taste of conventional (POP) C programming. It is intended to refresh your existing C programming skills. Since we're *nerds*, let's try an electrical problem!

The *average power* dissipated in an AC circuit can be calculated as follows:

$$P_{av} = \frac{1}{n} \sum_{i=1}^n \frac{V_n^2}{R_L}$$

Where V_n is instantaneous voltage ("sample") at each point on the waveform, n is the number of points on the waveform, and i is an index (counter) variable.

Don't panic. The above formula tells us to calculate average power by following this sequence of steps:

- Calculate the instantaneous power (V^2 / R) at each point on the waveform.
- Accumulate the sum of the instantaneous powers (add all the powers calculated in the first step above.)
- Divide by the number of samples, n , to get the average.

Your program will perform this calculation on a sine waveform. It will ask the user for a peak voltage, starting angle, ending angle, and load resistance (R_L). The program will calculate the voltage across the resistor in 1-degree steps (this result can be optionally displayed), and will report the average power in the resistor. A typical run looks like this:

```
Comp 370 Average Power Calculator
```

```
What is the peak voltage? 120  
What is the starting angle in degrees? 0  
What is the ending angle in degrees? 90  
What is the load resistance in Ohms? 50
```

```
The average power in the resistor is 144 Watts.
```

Note: You can use an array to store the sine wave data, but it is not required. This program must use modular design techniques -- don't just write a `main()` function!

PROJECT 3: DEFINING A CLASS

In order to demonstrate the action of objects in classes, you will create and define a class called *CResistor*. Objects of type *CResistor* will be an abstraction of physical resistors in electronic circuits, and they will have similar behavior. Your class must function correctly within the framework (main) shown below. You will use *main.cpp* as a foundation for this program.

```
#include <stdio.h>
#include "Resistor.h" // the header for your CResistor class

void main(void)
{
    CResistor r1,r2,rtotal;
    double value;

    //
    // Demonstration of finding series resistance
    //

    printf("What is the value of R1 in Ohms?");
    scanf("%lf", &value);
    r1.SetValue(value);

    printf("What is the value of R2 in Ohms?");
    scanf("%lf", &value);
    r2.SetValue(value);

    printf("What current will flow in the equivalent resistance (Amps)?");
    scanf("%lf", &value);

    rtotal = r1 + r2;          // overloading of the + operator will be required
    rtotal.SetCurrent(value); // Set current in new total resistance

    printf("The total series resistance would be %lf\n", rtotal.GetValue() );
    printf("There would be %lf volts dropped under this condition.\n\n",
           rtotal.GetVoltage() );

    rtotal = r1 || r2;        // overloading of the || operator means parallel
    rtotal.SetCurrent(value); // must set current again since assignment
                             // overwrites previous object data (default
                             // copy constructor is used.)

    printf("The total parallel resistance would be %lf\n", rtotal.GetValue() );
    printf("There would be %lf volts dropped under this condition.\n\n",
           rtotal.GetVoltage() );

}
```

Required methods for CResistor and their explanation:

Method Prototype	Method Explanation of Function
<code>void SetValue(double Value);</code>	This function sets the value of the resistor object.
<code>double GetValue(void);</code>	This function returns the current value held in the resistor object.
<code>void SetCurrent(double Current);</code>	Sets the current within the resistor object, which will be needed for voltage computation.
<code>double GetVoltage(void);</code>	Returns the voltage that is being dropped across the resistor, which is based upon the Ohm's law calculation of current and resistance.
<code>CResistor operator+(CResistor& ValR);</code>	Returns a new CResistor object that contains the total series resistance of two CResistor objects.
<code>CResistor Operator (CResistor& ValR);</code>	Returns a new CResistor object that contains the total parallel resistance of two CResistor objects.

Program Inputs:

The values as requested by the *main.cpp* framework.

Program Outputs:

The correctly calculated outputs from *main.cpp*.

PROJECT 4: CONSTRUCTORS AND DESTRUCTORS

Constructors and *destructors* are important components of most classes. A constructor is a method (function) that is called when an object of a class is created. This allows for controlled initialization of object data, which eliminates any questions as to the initial state of an object, regardless of whether it is locally or globally stored. An object's destructor is called whenever the object is about to go out of scope (which means that the object is about to be destroyed).

One of the primary actions that takes place in a destructor is the release of any memory that has been dynamically allocated by the object. Because this activity is made to take place automatically (at destruction time), the programmer's burden is eased, and the likelihood of memory leaks is greatly diminished.

In this project, you will create a general-purpose calculator for computing the total resistance of a series-parallel circuit made of CResistor objects. You will also exercise your knowledge of dynamic memory allocation by using `new` and `delete` to create and destroy intermediate CResistor objects. The program will be menu-driven and will appear as follows:

```
COMP 370 Resistance Calculator by (your name)
```

```
The total resistance is 0 Ohms.  
The voltage drop is 0 Volts.  
The current is 0 A.
```

```
What do you want to do?
```

1. Add a series resistor to the circuit
2. Add a parallel resistor to the circuit
3. Subtract resistance from the circuit
4. Change the current in the equivalent resistance
5. Exit the program

```
Choose?
```

How the program will work

The program will begin by creating a local CResistor object called *RTotal*, which will have a current of 0 Amps and a resistance of 0 Ohms by virtue of its default constructor. This object will persist throughout the entire run of the program and will serve to hold the total resistance, current, and voltage values.

When the user chooses option *1*, the program will ask for the resistance of the resistor to be added in series. A CResistor object will be created using the new operator (using the class constructor to initialize its value with the user's choice), and this object will be added to and stored in *RTotal*. The temporary CResistor object will then be destroyed. The program will then loop back to the menu and display the result.

Options *2* and *3* will operate in an identical manner to option *1*. A temporary CResistor object must be instantiated in each case to store and process the user's input.

Option *4* will simply ask for the new value of current, and will use the `SetCurrent()` method of the *RTotal* object to modify the current flowing in the equivalent resistance. After completion of this task, the program will loop back to the menu to display the result.

The following additions to the CResistor class will be necessary:

Method Prototype	Method Explanation of Function
<code>CResistor(double Value = 0, double Current = 0)</code>	Class constructor with two possible default arguments. Note that the "default" constructor configuration (no arguments passed) sets both the resistance and current to zero.
<code>~CResistor()</code>	Class destructor. <u>Print a message stating that a CResistor object was destroyed.</u>
<code>CResistor operator-(CResistor& ValR);</code>	Returns a new CResistor object that contains the difference in resistance of two CResistor objects.

PROJECT 5: AN OBJECT-ORIENTED DATABASE

Many programming projects involve the use of several classes that are designed to work together. This project introduces a class called CDatabase which is designed to encapsulate a data storage and retrieval system. CDatabase objects are complete functioning databases with the following capabilities:

- Storage of a variable number of document records.
- Record search for a particular text string.
- Deletion of selected records.
- Printing of selected records to the display.

CDatabase objects operate in a very straightforward manner. When a CDatabase object is instantiated, it allocates an array of CDatabaseEntry objects. The CDatabase object manages the individual CDatabaseEntry objects as needed, and when specific actions are needed (such as printing), control is passed to the proper CDatabaseEntry object.

For example, if the caller wishes to start up a CDatabase, add a record, and then print that record's contents (the first record in the database), the following sequence of instructions would work:

```
int nResult;          // result flag
CDatabase aDB(100); // create empty database with 100 elements

// Add a chip called "74LS14" to the database with a cost of
// $0.59 and a part number of 12345

aDB.AddEntry( "74LS14", 0.59 , 12345 );
//
// Check to see if the database accepted the entry
//
if (nResult < 0)
    printf("\n7The database did not accept this item.");

// print the first element in the database (the one we just
// created)

aDB.PrintEntry( 0 ); // 0 would be the first element,
                    // 99 would be the last (100 elements)
```

The logic to exercise the CDatabase class is provided for you in the file "main.cpp" which is a completely functional main program. The code for the CDatabase class is contained in the files "Database.h" and "Database.cpp" which are also supplied in completely functional form.

The project also demonstrates the use of *helper* classes. Helper classes are used to facilitate specific functions. The class *CInput* is a helper class that encapsulates most of the basic functionality for enhanced user input. *CInput* replaces *scanf()* and *gets()*, providing a much more stable (and simple) input mechanism. You can learn more about *CInput()* by studying the code in *main.cpp*, and of course, the source files for the *CInput* class.

What is my task in this project?

Your task is to create the CDatabaseEntry class, which CDatabase needs to function. All the other classes have been completely defined and written, and a functioning main program has been provided to test the works. The definition of the CDatabaseEntry class is as follows:

```
class CDatabaseEntry
{
public:
    CDatabaseEntry();
    virtual ~CDatabaseEntry();

    char* GetItemName();
    void SetItemName(char* szName);

    double GetItemCost();
    void SetItemCost(double dCost);

    int GetPartNumber();
    void SetPartNumber(int nPart);

    void PrintContents();

protected:

    char m_szItemName[128];
    double m_dItemCost;
    int m_nPartNumber;

};
```

Notice that each CDatabaseEntry object represents just *one* entry in the complete database. The CDatabaseEntry objects do not have to worry about managing the database; they must only take care of themselves.

Description of the CDatabaseEntry Methods

CAUTION: Your declaration of the CDatabaseEntry class must be in a file named "DatabaseEntry.h", and your implementation of the class must be in a file named "DatabaseEntry.cpp". The methods you write must correspond exactly to the prototypes and functional conventions described below, or your class won't interface properly with the database manager (CDatabase).

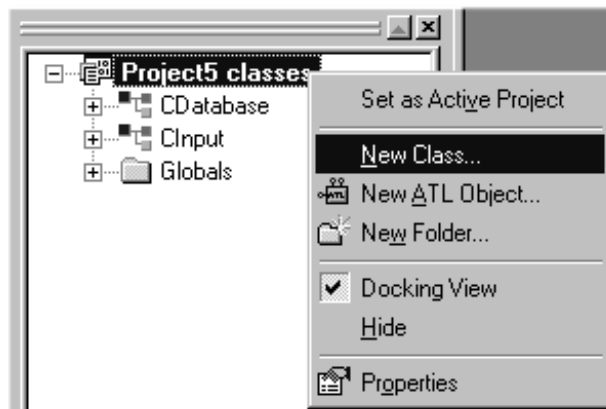
Method Prototype	Method Explanation of Function
<code>CDatabaseEntry();</code>	Class constructor. Set members <i>m_dItemCost</i> and <i>m_nPartNumber</i> to zero; place the words "Empty record" into the string <i>m_szItemName</i> .
<code>~CDatabaseEntry();</code>	Class destructor.
<code>char* GetItemName();</code>	Returns a pointer to the internal string <i>m_szItemName</i> .
<code>void SetItemName(char* szName)</code>	Copies the caller's string <i>szName</i> into the internal member <i>m_szItemName</i> (use <i>strcpy()</i>).
<code>double GetItemCost();</code>	Returns the value in <i>m_dItemCost</i> .
<code>void SetItemCost(double dCost);</code>	Sets the internal member <i>m_dItemCost</i> equal to the value passed in <i>dCost</i> .
<code>int GetPartNumber();</code>	Returns the value in <i>m_nPartNumber</i> .
<code>void SetPartNumber(int nPart);</code>	Sets the internal member <i>m_nPartNumber</i> equal to the value passed in <i>nPart</i> .
<code>void PrintContents();</code>	Prints the contents of the object to the screen in a neat format, such as: Item Name : 74LS14 Part Number: 12345 Item Cost : \$0.59 <i>Make sure to print a newline character before and after printing the record contents.</i>

How to Proceed with the Project:

1. Create a blank Visual C project of type Win32 Console.
2. Move the provided source files "main.cpp", "database.cpp", "database.h", "input.cpp", and "input.h" to the same Windows folder as your VC project.
3. At the *Project, Add to Project, Files...* menu of Visual C, add all of the files of step 2 to your project, then press the *Save All* (multiple disc) button on the toolbar.
4. Click on the *Class View* tab for the workspace.
5. Click on the [+] sign next to the top most item in the class view window to see the classes in your project. It should look something like this:



6. If you can't get the display above, don't proceed further until you find the problem, or get help.
7. Right click on the root item ("*MyDatabase classes*") and select the choice "*Add new class.*" You should see something like this:



8. Call the new class you're adding *CDatabaseEntry*. Visual C will automatically create the CPP and H files.
9. Implement the class *CDatabaseEntry* as described above.

PROJECT 6: DERIVING A NEW C++ CLASS

One of the most useful features of C++ is the ability to re-use objects without the need for the tedious operations of cutting and pasting source code. However, often an existing object doesn't completely fill a program's needs. In this case, two options are possible. First, the programmer can modify the source code for the object in question, and add the required functionality. This method assumes that the programmer has access to the object's source code, and that the changes produced will not affect how the object will work in other portions of the project. Often this is an undesirable method.

The second way of providing additional (or modified) functionality is to derive a *new* class from the existing class, using the C++ property of *inheritance*. The new class can then be modified appropriately to obtain the desired behavior. The advantage of this method is that the original class definition is left unchanged, so that other programs or modules that use the original class will continue to operate with no required changes.

When a programmer derives a new class from an existing one, he or she does not need detailed knowledge of the inner workings of the parent class. The programmer need only concentrate on the details that will be *different* from those in the base (parent) class.

When a new class is derived from a base class, all the methods and variables of the parent class are inherited, with two exceptions. Constructors, destructors, and private members of the base class are not inherited. Therefore, the child (derived) class must provide its own constructor and destructor. The child class will not have access to any *private* members of the parent class. Well-designed base classes use the keyword *protected* to allow derived classes access to necessary data elements.

A child class is responsible for constructing its immediate parent. If the parent has a default constructor, the compiler will automatically call it. Under any other condition, the child must call the parent's constructor before it can proceed.

Description of the Project

In this project, you will derive a new class called *CMyDatabase* from the *CDatabase* class of project 5. *CMyDatabase* will provide the same exact functionality as *CDatabase*, except for the following details:

- *CMyDatabase* will provide a function called *Browse()* that performs the same function as the *BrowseDB()* function in the *main* program of project 5. (You may transplant this code as needed.) This will demonstrate how a derived class can have enhanced functionality when compared to a parent.
- The *PrintEntry()* method of *CDatabase* will be overridden in *CMyDatabase*. This means that *CMyDatabase* will contain a function called *PrintEntry()* that replaces the *PrintEntry()* function from the base class. This will demonstrate the ability to override parent-class methods.

Instructions

1. Build a new visual C console application called "project6" in a new folder.
2. Copy all the source and header (*.h, *.cpp) files from project 5 except main.cpp to the project6 folder.

*Caution: Do not copy anything except the source and header files. For example, do not copy the project (*.dsp) or workspace (*.dsw) files from project 5!*

3. Copy the supplied main.cpp for project 6 into the project 6 folder. This new main program will test your *CMyDatabase* class.
4. Add all the source files and header files in the project 6 folder to the project, using the *Project, Add to Project, Files...* menu of Visual C.
5. Add a new class to your project called "CMyDatabase" that is derived from "CDatabase." To do this, right-click on the root item ("project 6 classes") in the *class view* window, then select "*New Class...*" from the menu. The resulting dialog will allow you to derive the new class from an existing class; choose this option. The resulting files generated will be "MyDatabase.cpp" and "MyDatabase.h".

6. Examine the beginning of "MyDatabase.h" and you'll notice that Visual C has automatically added the following include:

```
#include "Database.h"
```

This should make sense; after all, the CMyDatabase class is derived from CDatabase, and the compiler therefore needs to know the details about CDatabase when building CMyDatabase objects. However, this isn't enough information. The CDatabase class also uses CDatabaseEntry objects, and the compiler needs to know about these as well. Therefore, the following include must be added to "MyDatabase.h" *before* the one above:

```
#include "DatabaseEntry.h"
```

7. Add the following constructor code to CMyDatabase. This code will correctly call the base class constructor:

```
CMyDatabase::CMyDatabase(int nMaxEntries):CDatabase(nMaxEntries)
{
;
}

// This constructor should be prototyped as follows in MyDatabase.h:

CMyDatabase(int nMaxEntries = 100 );
```

Important: Make sure that CMyDatabase has no default constructor. Otherwise, an error will be generated at compile time due to the ambiguous default constructor call.

8. Add the prototype for the overridden *PrintEntry()* function to the MyDatabase.h file, and the *PrintEntry()* function to MyDatabase.cpp:

```
void CMyDatabase::PrintEntry(int nItemID)
{
printf("\n-----"); // print extra banner so that
// we can tell that overridden
// method is being called

CDatabase::PrintEntry(nItemID); // call base class method here
printf("-----\n");
}
```

9. Add the prototype for the new *Browse()* function to MyDatabase.h:

```
void Browse(void);
```

10. Add the *Browse()* function to *MyDatabase.cpp*. You can do this by *transplanting* the code for *BrowseDB()* from the *main.cpp* of project 5. No argument is passed, and since the function is now contained within the object, there is no need for the "pDbase" pointer. Every place where this pointer is accessed, such as:

```
if ((nCount=pDbase->GetNumberEntries()) == 0)
```

You can replace this code with either of the following:

```
if ((nCount = this->GetNumberEntries()) == 0)
```

Or even better,

```
if ((nCount = GetNumberEntries()) == 0)
```

Once you have completed this, you will now have a database object that is "self browsing" (look at how *main.cpp* has been simplified.)

11. The project should now compile and run correctly with these changes. Note how *main.cpp* instantiates the database; it now builds a *CMyDatabase* object, instead of a *CDatabase* object. *CMyDatabase* has inherited all the functionality of *CDatabase*, so everything in *main()* uses it identically -- except where we implemented the *Browse()* function.
12. If you have time, see if you can transplant the other functionalities (such as the search and delete dialogs) into *CMyDatabase*. (This is not a primary requirement for this project, so do it at your own pace.)