CIS 130 - Intro to Programming - Fall 2005
Homework Assignment #8

Homework #8:
HW #8 PART 1 is due by **12:00 NOON** on **Wednesday, November 2, 2005;**
HW #8 PART 2 is due by **12:00 NOON** on **Friday, November 4, 2005**

Week "10" Lab Exercise - due at end of your registered lab section
on either 10-28 or 10-31

WEEK "10" LAB EXERCISE

For this lab exercise, we are going to "walk through" developing a complete C++ program.

0. Make a new directory for this lab exercise; from your home directory on cs-server, type:

```
mkdir 130lab10
```

Now, copy over some files you'll need for this lab by typing this command on cs-server:

```
cp  ~st10/130lab10_public/* 130lab10
```

Finally, change over to this new directory, so you'll be ready to proceed.

```
cd 130lab10
```

1. One reason to write self-contained "pure" functions is that they can be easier to re-use.

You've just copied over versions of **ring_area** and **disk_area**; they are in the files **ring_area.cpp, ring_area.h, disk_area.cpp**, and **disk_area.h**

Consider, for a moment, the file **disk_area.h**:

```
/*-----------------------------------------------
 header file for function disk_area
   created by st10 at Thu Oct 27 14:42:57 PDT 2005
-----------------------------------------------*/
#ifndef disk_area_H
#define disk_area_H

const double PI = 3.14159;

double disk_area(double radius);

#endif
```

* The **#ifndef**, **#define**, and **#endif** are more precompiler directives (like **#include**). These are discussed in Section 5 of the HtDP reading packet, but, as a reminder, what they do is as follows:

#ifndef and **#endif** surround a region of code that is to be included if the name after **#ifndef** has not yet been defined; if it has been defined, that region of code is to be ignored. (Get it? #ifndef --> if not defined...)

#define defines a name (in precompiler terms...). In this case, it is very important that it define the same name as is found after **#ifndef**.

SO: Consider many functions being "gathered" into a program. What if two of them happened to use the same function? Each would #include it, and you might end up with the function being declared twice; C++ doesn't like that, and will consider such "double declaration" to be an error.

#ifndef, **#define**, and **#endif** prevent such double declaration. When the "first" function #include's this file, the name after **#ifndef** will indeed not be defined, and that region of code will be included. That will also cause the **#define** to define that name.

When the "second" function #include's this file, the name after **#ifndef** will now be defined. So, the "second" copy of this code is NOT included, and double-definition is avoided, but all functions still get access to the function declarations needed.

Note: for this to work, each **.h** file must #define a different name! By selecting the #define name to be based on the function name, then, you should not have any name collisions.

* NOTE: Really, this .h file is pretty "cookbook" --- that's how I was able to so easily write a tool to generate it automatically, after all.

  On the public course web page, you'll now find a **.h** file **template** --- you can copy it, and then make the modifications needed for your particular **.h** file.

* You also now have another version of **disk_area.cpp**

```
/*-------------------------------------------------
created by st10 at Thu Oct 27 14:42:57 PDT 2005
-------------------------------------------------*/
#include <cmath>
#include "disk_area.h"
using namespace std;

/*-------------------------------------------------
 Contract: disk_area : double -> double
 Purpose: compute and return the area of a disk whose
         radius is <radius>

 Examples: disk_area(1) == 3.14159
           disk_area(5) == 78.53975
-------------------------------------------------*/
double disk_area(double radius)
{
    return PI * pow(radius, 2);
```

```
}
```

The only "new" things of note here are to notice:

* the **#include** of the **cmath** standard library, necessary because we are calling its **pow** function;

* the **#include** of **disk_area.h**. Normally, a function does NOT need to #include its own .h; the exception is if that is where named constants are defined. **PI** is defined there, in this case, so **disk_area.cpp** does not to #include **disk_area.h**

* notice the **using namespace std;** --- that will be our default after all of the #includes for a file.

* To **compile** disk_area.cpp, make sure that you are in the directory where that file is, and type (at the cs-server prompt):

```
g++ -c disk_area.cpp
```

* You also have copies of **ring_area.cpp** and **ring_area.h**:

ring_area.h:

```
/*-------------------------------------------------
 header file for function ring_area
   created by st10 at Thu Oct 27 21:22:21 PDT 2005
-------------------------------------------------*/
#ifndef ring_area_H
#define ring_area_H

double ring_area (double outer, double inner);

#endif
```

* See how similar this is to **disk_area.h**? But, be sure that you notice where it differs, too.

ring_area.cpp:

```
/*-------------------------------------------------
created by st10 at Thu Oct 27 21:22:21 PDT 2005
-------------------------------------------------*/
#include "disk_area.h"
using namespace std;

/*-------------------------------------------------
 Contract: ring_area : double double -> double
 Purpose: Compute and return the area of a ring whose outer
          edge has radius <outer>, and whose "hole" has radius
          <inner>.
```

```
 Examples: ring_area(10, 5) = 235.619
           ring_area(5, 3) == 50.265
-----------------------------------------------*/
double ring_area (double outer, double inner)
{
    return disk_area(outer) - disk_area(inner);
}
```

  * This particular function calls another, **disk_area**, and so it #include's **"disk_area.h"**.

* And, to compile **ring_area.cpp**, type this at the cs-server prompt:

  ```
  g++ -c ring_area.cpp
  ```

* Finally --- here are two example **main** functions that make different use of **ring_area**.

  The first is a "testing main" --- it is written to automate your function's examples. You have copied it over into a file named **test_ring_area.cpp**:

```
// Contract: main: void -> int
// Purpose: test function ring_area's examples
//
// Examples: If the tests succeed, two 1's (for true)
//           should be written to the screen.
//
// By: Sharon Tuttle
// Last modified: 10-28-05

#include <cmath>
#include <iostream>
#include "ring_area.h"
using namespace std;

int main()
{
    cout << "Testing ring_area: " << endl;
    cout << "   (1 == pass, 0 == fail)" << endl;
    cout << "-------------------------" << endl;

    cout << ( abs(ring_area(10, 5) - 235.619) < .001) << endl;
    cout << ( abs(ring_area(5, 3) - 50.265) < .001) << endl;

    return EXIT_SUCCESS;
}
```

  * This is a quite standard "testing" main function;

  * It #includes the **cmath** standard library because it calls **abs** (absolute value), and it #includes the **iostream** standard library because it uses **cout** and **cin**.

* It #includes **ring_area.h** because it calls **ring_area**.

* Notice how the tests were rewritten (compared to how they were shown in the Examples section in ring_area.cpp); you'll remember that we discussed this approach for wanting to see if two double values are "close enough".

* And, notice that our main returns **EXIT_SUCCESS**...

* To just compile this, at the cs-server prompt type:

```
g++ -c test_ring_area.cpp
```

And, to link and load the whole program (and create an executable), type at the cs-server prompt:

```
g++ -o test_ring_area test_ring_area.cpp ring_area.cpp disk_area.cpp
```

or, if you prefer,

```
g++ test_ring_area.cpp ring_area.cpp disk_area.cpp -o test_ring_area
```

Note that you need to type either the **.cpp** files or the **.o** files for EVERY function involved in the program --- main calls ring_area, and ring area calls disk_area, so all 3 of test_ring_area.cpp, ring_area.cpp, and disk_area.cpp have to appear in the above link-load-and-create-executable statement.

I've included a quickie reminder of these compiling commands on the public course web page, in the "References" section.

* Now, to RUN your program **test_ring_area**, simply type at the cs-server prompt:

```
test_ring_area
```

... and you should see its results.

* And, what if you wanted to create a file containing these results? You can simply **redirect** the output to go to a file instead of to the screen by using **>** and a file name after the command; for example,

```
test_ring_area > test_ring_area.out
```

...would create a file named **test_ring_area.out** that shows the results of running **test_ring_area** (in this case, what it prints to the screen when it is run).

2. Let's try one more main function: this one will create an interactive user interface for ring_area. That is, it will ask the user for outer and inner ring radii, call ring_area to determine the area of that ring, and then print the resulting area to the screen for the user.

Do you see that ring_area and disk_area are ready to go? Their files **ring_area.cpp**, **ring_area.h**, **disk_area.cpp**, and **disk_area.h** exist and have been compiled.

We just need the new main function. You'll see it in your copied file **asking_ring_area.cpp**:

```
// Contract: main: void -> int
// Purpose: ask user for inner and outer radii of a ring,
//          and print the area of that ring to the screen
//
// Examples: if the user types in an outer radius of 10
//           and an inner radius of 5, then the following
//           should be printed to the screen:
// Area of this ring is: 157.0795
//
// By: Sharon Tuttle
// Last modified: 10-28-05

#include <iostream>
#include "ring_area.h"
using namespace std;

int main()
{
    // declarations

    double outr_rad;
    double innr_rad;
    double area;

    // ask user for outer and inner ring radii

    cout << "enter an outer-ring radius: ";
    cin >> outr_rad;

    cout << "enter an inner-ring radius: ";
    cin >> innr_rad;

    // compute and print that ring's area

    area = ring_area(outr_rad, innr_rad);

    cout << endl
         << "Area of this ring is: "
         << area
         << endl;

    return EXIT_SUCCESS;
}
```

* Note how **cout** and **cin** statements are being used to create a very simple interactive user interface to the ring_area function;

* And, we have #include'd "ring_area.h", because it calls function ring_area;

* And, the main function returns EXIT_SUCCESS.

* To just compile this, at the cs-server prompt type:

```
g++ -c asking_ring_area.cpp
```

And, to link and load the whole program (and create an executable), type at the cs-server prompt:

```
g++ -o asking_ring_area asking_ring_area.cpp ring_area.cpp disk_area.cpp
```

or, if you prefer,

```
g++ asking_ring_area.cpp ring_area.cpp disk_area.cpp -o asking_ring_area
```

* Now, to RUN your program **asking_ring_area**, simply type at the cs-server prompt:

```
asking_ring_area
```

... and you should see its results.

* Note that interactive programs like this one do not lend themselves well to redirected output files; the prompts get written to the file, but the program still stops to wait for input, so it can be very confusing! 8-)

To complete the lab exercise, I'll come by to each of you and ask you to run **test_ring_area** and **asking_ring_area**. As long as you are able to run both for me by the end of your scheduled lab session, then you will have completed this lab exercise.

HOMEWORK #8

You are to work **individually** on this assignment.

PART 1: (30% of the HW #8 grade)

Complete problem #1 below, and submit the files it mentions (using **~st10/130submit**) by **12:00 noon** on **Wednesday, November 2nd** to receive **any** credit for Part 1 of HW #8.

PART 2: (70% of HW #8 grade)

Complete problems #2 - #5 below. When you are ready, you must submit the files specified in each problem (using **~st10/130submit**) by **12:00 noon** on **Friday, November 4th** to receive **any** credit for Part 2 of HW #8.

1. You saw a "testing main" function for **ring_area.cpp** as part of the lab exercise above (in **test_ring_area.cpp**)

Write a "testing main" function for **disk_area**; put it in **test_disk_area.cpp**.

When it has compiled and runs, create an output file to submit by typing the following command at cs-server:

```
test_disk_area > test_disk_area.out
```

For this problem, you should submit (using **~st10/130submit**) the files **test_disk_area.cpp** and **test_disk_area.out**.

2. And, you saw you **asking_ring_area.cpp** included a main function that implemented, essentially, a simple interactive interface for the function **ring_area**.

Write a main function in a file names **asking_disk_area.cpp** that implements a simple interactive interface for the function **disk_area**. Compile and test it (and debug it if necessary) until you are satisfied with it.

Since, as mentioned in lab, interactive programs do not lend themselves to redirected output files, you only need to submit **asking_disk_area.cpp** for this problem.

3. Let's branch this out one step further.

We want a program that users can use to find out either a ring's area or a disk's area, their choice.

It should first ask the user to type **1** if they want a disk area, and **2** if they want a ring area. Then, if the user has typed **1**, the program should ask for the disk radius, and then compute and print to the screen that disk's area. If the user has typed **2**, then the program should ask for the inner and outer radii of the ring, and then compute and print to the screen that ring's area. And, if the user typed neither **1** nor **2**, the program should simply complain to the screen that an illegal answer was given.

Obviously, your program will make use of the functions **ring_area** and **disk_area**. Type the main function in a file named **disk_or_ring.cpp**. Compile/test/debug it until you are satisfied with it.

Since this is interactive, no output file will be required; and, since ring_area and disk_area were provided to you, you need not submit those for this problem, either. You need to submit just **disk_or_ring.cpp** for this particular problem.

4. main functions are not the only ones that do interactive input/output (i/o).

NOTE that it is considered POOR STYLE for a "pure" function such as those we have been writing before this week to do interactive i/o; these functions are expected to quietly take their arguments, compute a result, and return it to the caller. They can be easily called from other programs without messing up those programs' user interfaces, causing unexpected output to the screen, etc.

But, some functions are written **for** such side-effects; their whole point is to have an effect, not necessarily return a value.

Write a function **show_sig** that takes no parameters, and returns no result; it simply prints to the screen **at least three lines** that include your name and whatever else you'd like to include. (For example, how might you "sign" an e-mail message? You might include your name, that you are a student at Humboldt State University, and perhaps a message of the day.)

I'll leave it up to your what your "sig" should include, as long as it meets the following minimum requirements:

* it should include your name;
* you should print something non-blank on at least three different lines;
* you should try to make it attractive.

NOTE1: the return type for a function that returns NO value is **void**. And, a **void** function --- one that returns no type --- does not need to include a **return** statement.

NOTE2: in the contract for a function that takes no arguments, you should use **void** for the type of its non-existent arguments. And, in the contract for a function that does not return anything, the return type given should be **void** (just like the return type for the function, when it is written!)

You will submit **show_sig.cpp** and **show_sig.h** for this problem.

But --- how will you **test** it? **expr_play** will not work well for this! You'll need to write a testing main for this one, and that leads us to...

5. Write a testing main function in a file named **test_show_sig.cpp** to test show_sig.

Since it prints to the screen, you cannot compare its result to an expected result; you CAN print a message to the screen before calling it, however, saying what SHOULD be about to happen (what the expected behavior should be). You are thus expected to do so.

NOTE1: if a function does not return any result, it can be typed by itself on a line when it is called; such a function call should be followed by a semicolon, and that call then becomes a C++ statement.

NOTE2: if a function takes no arguments, its name STILL must be followed by a set of parentheses when it is called! That set of parentheses will simply be empty.

Put these two notes together, and you now know that **show_sig** can be called in your testing main function by typing the statement:

```
show_sig();
```

This one can have its output redirected to a file, so when you are done with compiling/testing/debuggin and are happy with your main in **test_show_sig.cpp**, create an output file by typing:

```
test_show_sig > test_show_sig.out
```

And submit **test_show_sig.cpp** and **test_show_sig.out** for this problem.