1 Purpose

1. Explore scope of identifiers in C++.
2. Practice "global", "local" and "shadowing" concepts.
3. Practice writing own C++ user defined functions.
4. Practice with parameter passing in C++ user defined functions.
5. Explore logical expressions and `bool` objects.

2 Exercise 1: Understanding Scope

A *declaration* in a program is a way to associate certain attributes (for example a type) with an identifier. The declarations

```
int x, y;
char ch;
```

associate the type-attribute `int` with the identifiers `x` and `y` and the type-attribute `char` with the identifier `ch`.

When the compiler is processing the code of a program it has to keep track of all the identifiers it has seen and the attributes associated with each identifier. The *scope of a declaration* is the region of a program's code where the compiler remembers the attributes from the declaration. Remember identifiers in C++ name more than objects or variables. Identifiers also name constants, functions, classes and other entities.

C++'s *scoping rule* states that the scope of a declaration extends from the declaration itself to the end of the block where the declaration appears; that usually means from the point of the declaration up to the next right curly brace which isn't matched with a left curly brace. A basic rule of declarations is that you cannot declare an identifier twice in the same block. It is also possible in C++ to have one block *nested* inside another block and then the "basic rule of declarations" does not apply — you can redeclare an identifier inside a nested block.

Copy the file `scope.cc` from `~cs203/Labs/Lab5` to your `Lab5` directory. Study the program carefully and try to determine what will be printed when run *before* you compile and run the program.

After you have analyzed the program and written down your answers, compile and run it. Try to understand why the program gives the results it does. If you need an explanation, please ask your lab TA or instructor. Don't continue with the lab until you are confident you understand scope.

Copy and paste the contents of `scope.cc` file and its output to your hand-in file. On the hand-in file copy, circle the declarations of the objects and constants and using a bracket show the scope of each.

Place in your hand-in file responses to questions that follow. (Specify which "`k`" object you are talking about by using the line number. `Emacs` displays the current line number (the line with the small black rectangle) at the bottom as "Lline-number", for example, `L40`.)

1. Describe "global". What objects and associated line number are *global*?
2. What constants and associated line number are *global*?
3. Describe "local". What objects and associated line number are *local*?
4. What constants and associated line number are *local*?
5. Describe "parameter". What objects and associated line number are *parameters*?
6. Describe "argument". What objects and associated line number are *arguments*?
7. Describe "shadowing". What object(s) and associated line number shadow others?

3 Exercise 2: Parameter Passing

C++ allows calls to functions to pass arguments *three* ways — 1) **pass by value**, 2) **pass by reference** and 3) **pass by constant reference**. You need to understand and to be able to use all three.

Write a C++ program with a `void` function with one `int` parameter `x`. The function should output the function name and the value of `x`, similarly to the output used in `scope.cc`. Then one is added to `x`. Finally before returning, the function outputs again the function name and the value of `x`.

The main program declares an `int` object `y`, assigns it 7 and passes `y` as an argument to your function. After the call, the program outputs the value of `y`.

Type in your program and get it to compile. In your hand-in file, provide a listing of your program and a run.

3.1 Pass by Value

If you use only `(int x)` in your function header, the *value* of `y` is *copied* to the newly created parameter object `x` on the call. This is called **pass by value**. When the program hits the `return` statement, all the local objects and constants including all the parameters in the function are destroyed. Notice that the state of `y`, the argument, is unchanged.

The *lifetime* of an object or constant begins when the object or constant is created until it is destroyed. For *local* objects and constants in a function, they are created when the function is called and destroyed when the function returns. In the `scope.cc` program, the function `funny()` is called twice, therefore, the parameter `k` and local object `j` are each created and destroyed twice.

3.2 Pass by Reference

Now alter your function header and place an `&` between the `int` and `x`. Compile and run again. In your hand-in file describe what is printed? What is different?

If you use `&` as in `(int & x)` in your function header, the *address* of `y` is passed to `x` on the call. Any use of `x` in the function uses the address of the *same* cell in memory as `y`. Any change to the state of `x` is a change of state of `y`. This is called **pass by address** or **pass by reference**. Notice that the state of the argument object, in this case `y`, may change.

3.3 Pass by `const` Reference

Now alter your function header and place a `const` before the `int`. Compile and run again. In your hand-in file describe what is printed? What is different?

If you use `const` as in `(const int & x)` in your function header, the *address* of `y` is passed to `x` on the call as before with **pass by reference** but the function promises *not* to change the state of `x`. The compiler enforces this promise and gives you an error message because your program violates it.

Alter your program by removing the `x = x + 1;` line and compile and run. Does it compile now?

4 Exercise 3: Parameter Passing

Copy the file `ex3.cc` from the `~cs203/Labs/Lab5` directory into your `Lab5` directory. Read and study the code carefully. Compile and run it feeding Samba — as usual.

Remember that to compile a Samba program you type the following on *one* line:

```
g++ ex3.cc -L/home/hydra/COURSES/cs203/lib
           -lSamba -o ex3.exe
```

To run it, type the following in your **Terminal** window.

```
ex3.exe | samba
```

In this exercise, you will need to reread the discussions in Exercise 2. This exercise asks you to apply what you have just learned to a new situation. Try to visualize what is happening to the Circle object(s). Run samba in slow mode to observe what is happening. Ask yourself: "Are there two Circle objects or one? What is the **lifetime** of the object(s)?"

In your `handin.txt` file, answer the following questions under a banner labeled "Exercise 3":

1. Describe in a sentence what happens to the Circles on the screen.
2. What does the `delay()` function do?
3. Is the object `x` a "call by value" or "call by reference" parameter? Why?
4. Why does the orange circle disappear before the blue one?
5. Insert an `&` between `Circle` and `x` and recompile and run the program again. Describe in a sentence what happens on the screen.
6. Why is there only one circle now?

5 Exercise 4: C++ Logical Expressions and `bool` Objects

Two objects of certain classes, e.g., `int`, `double` and `string`, can be compared by using a comparison operator. The comparison operators in C++ are the following:

`==`	equal — Note *two* equal signs, not one!
`!=`	not equal
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal

An expression such as `(x - 3) < (y * 5)` is a *logical expression*, i.e., it has truth value (evaluates to *true* or *false*). Logical expressions are also referred to as *Boolean* expressions, after the turn-of the eighteenth century mathematician George Boole.

Compound logical expressions can be created using logical expressions along with the `&&`, `||` and `!` operators, standing for *and*, *or* and *not*, respectively.

`&&`	"and" operator, true if both logical expressions are true.
`||`	"or" operator, true if either logical expression or both is true
`!`	"not" operator, true if the logical expression is false, false if the logical expression is true.

Note that both `&&` and `||` have *two* symbols. (The single characters `&` and `|` mean something different!)

Type in the following C++ program. You'll notice that there is a new type used in the program. `bool` is the name of the type which has just two values — `true` and `false`.

```
#include <iostream>
int main(){
    bool answer;
    int x, y;

    x = 2;
    y = 3;

    answer = x < 5;   // replace "x < 5" expression here

    cout << answer << endl;

    return 0;
}
```

Study this program, compile it, and run it.

Try the following expressions, one by one, by replacing the `x < 5` in the above program with each. In your `handin.txt` file present the answer to each of the seven logical expressions and explain the result. Clearly label the part of the question you are answering. It's a good idea to try to guess the answer before the program tells you.

1. `true`

2. `false`
3. `x = 7`
4. `x == 7`
5. `(x > 1) && (y < 2)`
6. `(x > 1) || (y < 2)`
7. `!(x == 3)`

6 Hand In

Hand in the answers from the four exercises which should be in your `handin.txt` file. Make sure you have the proper banner across at the beginning of the file and appropriate banners separating each of the exercises.