# COMSC-165 Lab 2, Basic C/C++ Control Structures

---

In this lab session, you will practice practical control structures, using three exercises out of the Deitel textbook. Each exercise allows you to write a programming solution "from scratch", without a lot of guidance. All that's required is that you apply the programming tools and conventions presented so far in this course. We have not covered functions yet, so no functions other than `main`.

As you complete this assignment, post the required file(s) to the COMSC server. For assignments that involve more than one file, you may post them all at once, or one-at-a-time as you complete them, as you prefer. For assignments that involve more than one separate "lab" (like lab 1a, 1b, and 1c), the individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your FA2014 work" link on the class website to post your file for this lab, using the **lab2** folder provided.

---

**LAB 2a:** Typewriter Graphics `[ Square.cpp ]`
Write **Square.cpp** to solve exercise 3.25 on page 112 of Deitel (reproduced below):

**3.25** *(Square of Asterisks)* Write a program that reads in the size of the side of a square then prints a hollow square of that size out of asterisks and blanks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 5, it should print

```
*****
*   *
*   *
*   *
*****
```

"Read" means to use console input. "Print" means output to the console screen. Be sure to use understandable prompts, and remember to use the "string buffer method" to read the "size". Include the 1-20 range in your prompt, so that the user knows what to expect. Check the size value, and if it's outside the 1-20 expected range, print no square.

As an enhancement, use a sentinel-controlled loop to run the program over and over again until the user wants to quit. Quit when the user inputs an uppercase or lowercase Q. Quit by breaking out of the loop -- do not `return` from inside the loop.

If the user enters a value outside the 1-20 range, including anything that `atoi` resolves to zero, skip to the next cycle. Note that since string buffers and `atoi` are being used, any non-numeric entry for size will be interpreted as zero.

For example, the user enters 5 for size -- output a square of size 5, and prompt for a new size. Or the user enters XXX for size -- that resolves to zero, so output nothing and prompt for a new size.

*Post **Square.cpp** to the COMSC server for credit.*

---

**LAB 2b:** Palindromes `[ Palindrome.cpp ]`
Write **Palindrome.cpp** to solve exercise 3.26 on page 112 of Deitel (reproduced below):

**3.26** *(Palindromes)* A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether it's a palindrome. [*Hint:* Use the division and modulus operators to separate the number into its individual digits.]

"Read in" means to use console input. The output should say in your own words if the inputted value is a palindrome or not a palindrome. If a blank line is entered, skip any output.

Note that this program works with *numbers*, not words. Also, numbers with fewer than 5 digits are automtically "padded" with leading zeros. So if you enter 0, that gets treated as 00000, and it's a palindrome. But 1 gets treated as 00001, and it's not. 100 is, because that's 00100. Be sure to test this program with a wide variety of input values, to confirm that it works right.

As an enhancement, use a sentinel-controlled loop to repeat the palindrome determination until the user enters a lowercase or

uppercase Q to quit. For example, the user enters 12421 -- print (e.g.) "that's a palindrome", and prompt for a new input. Continue until a Q is inputted, and break out of the loop to end the program -- do not `return` from inside the loop. Remember to tell about "Q to exit" in your input prompts.

*Post **Palindrome.cpp** to the COMSC server for credit.*

---

**LAB 2c:** Tabular Formatting `[ Growth.cpp ]`
Write **Growth.cpp** to solve exercise 3.36 on page 114 of Deitel (reproduced below):

**3.36** *(World Population Growth)* World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable cropland and other limited resources. There is evidence that growth has been slowing in recent years and that world population could peak some time this century, then start to decline.

For this exercise, research world population growth issues online. *Be sure to investigate various viewpoints.* Get estimates for the current world population and its growth rate (the percentage by which it's likely to increase this year). Write a program that calculates world population growth each year for the next 75 years, *using the simplifying assumption that the current growth rate will stay constant*. Print the results in a table. The first column should display the year from year 1 to year 75. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the year in which the population would be double what it is today, if this year's growth rate were to persist.

There is no input for this program -- only console output, in a nice tabular format.

Use this for left or right justification, in conjunction with `setw`. Remember to get `ios` and `cout` from the `iostream` library, and `setw` from `iomanip`:

```
cout.setf(ios::left, ios::adjustfield); or the manipulator left
cout.setf(ios::right, ios::adjustfield); or the manipulator right
```

Ignore the last line of the exercise, about determining the year of doubling.

Remember that the `int` limit is 2 billion, and you'll need numbers bigger than that. Here's a hint -- use the `long long` data type! Also note that whole numbers are "literal `int`s" -- 7000000000 is *too big* and some compilers will choke on it. Whenever you need a whole number larger than 2 billion, append an L, like this: 7000000000L, so it's interpreted as a long.

*Post **Growth.cpp** to the COMSC server for credit.*

---

**GRADING POLICY**
Lab work will be returned for you to fix and resubmit for any of the following reasons, or for not following stated specifications, unless directed otherwise by your instructor, Point penalties will be assessed per the syllabus for each time a lab is returned for redo for any listed reason below or specification above. Points will be awarded for an assignment upon final, successful completion of all of its labs. Note that you will NOT receive a list of corrections to make, because to do so would remove any incentive for students to do quality work. Instead, grading of your work will stop upon discovery of the first mistake. Grading will not continue until the mistake is corrected. So to avoid multiple cycles of correction and resubmission, and the cumulative point penalties, make sure that your work is as correct as you can make it before submitting it.

Basic checklist, even if your compiler "forgives" doing otherwise:

1. Spell and case filenames *exactly* as specified. To avoid errors, be sure to configure your Windows computer to NOT hide file name extensions -- refer to the Burns Intro to Programming text, chapter 2.
2. Submit files to the specified folder. Do NOT submit archive files, like ZIP, JAR, or RAR.
3. Include identifying information as the first lines in the file AND as the first lines of your output in main.
4. Do NOT use the statement `system("PAUSE");` or any other system-specific code.
5. Do NOT try to access the value stored in an uninitialized variable.
6. CPP and H files must be free of careless typing mistakes that would prevent compilation.
7. Put an `endl` as the last item in the last-executed `cout` statement before the closing curly brace of `main`.
8. Do NOT forget to use the `cin.ignore` statements for non-string console input, unless specifically directed otherwise.

9. Do NOT forget to use the `fin.ignore` statements for non-string file input, unless specifically directed otherwise.
10. For file I/O, NEVER include a path designation. ALWAYS use the "working folder" for files.
11. Use the proper `#includes` and `usings` as listed in the C++ Library PDF.
12. Do NOT #include C-libraries such as `math.h` -- include their C++ equivalents instead, like `cmath`.
13. Do NOT #include libraries that you do not use.
14. Do NOT #include "**stdafx.h**"; do NOT use `pragma`.
15. Do NOT use C/C++ constructs or library items that are not specifically taught in this class, such as `goto`, `scanf` or `fscanf`, `fixed`, `strcmp_c`, `stricmp`, etc.
16. Do NOT use `void main()` -- use `int main()` always.
17. Do NOT use `return 0;` as the last statement in `int main()`, because it is not required by ANSI Standard C++.
18. If you use output statements in your program for debugging purposes, be sure to delete them before submitting your work.
19. Do *not* use "global variables".

Advanced checklist, even if your compiler "forgives" doing otherwise:

1. `NULL` is ONLY a pointer value -- do NOT use it for integer or other numeric values, even if your compiler allows it.
2. Do NOT declare a static array with a variable size specification, such as `int n =...; int a[n];`, even if your compiler allows it.
3. NEVER #include a CPP file, even if the textbook includes examples of such.
4. For console input and text file input with `>>`, use `.ignore` as explained in the Burns Intro to Programming text, chapter 5, unless specifically directed otherwise.
5. NEVER use `cin >>` to read directly into a numeric variable, like `int` or `double` or `float`. Read as a string (C or C++) and convert to a number using `atoi` or `atof`. Include `cstdlib`!
6. Do NOT use the `inline` keyword. To make class member functions inline, write their definitions inside the class definition.
7. Once the `const` keyword is introduced in this course, use it where applicable. Note that class "getter" functions are not really getter functions unless they have a trailing `const` keyword to flag them as such.
8. Make sure that all value-returning functions return a valid value under any logical circumstance.
9. Exit functions with `return` statements, and NOT `exit` -- do NOT use `exit` under any circumstances.
10. In every H and CPP file, ALWAYS #include libraries and H files for ALL functions, classes, and identifiers used in that file. Exception: use a class "forward declaration" instead of #including the class' H file when only pointers and reference variables appear.
11. Include the `cstring` library for char-array-based string ("C string") *functions*, and the `string` library for STL-based-strings ("C++ strings").
12. When using the random number generator seed function `srand`, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
13. In COMSC 210, use `using namespace std;`. In 165 and 200, use `using std::` statements, properly placed under the #includes of the libraries to which they pertain.
14. `std::string` and `std::getline` belong to the `string` library. In 165 and 200, use `using std::` statements for these. Note that `cin.getline` is a different thing, and only requires `using std::cin;`.
15. Do NOT use `using std::` for anything in the C libraries. Use it for items from C++ libraries, only.
16. Test drivers MUST at least include ifndef testing, object copy testing with assignment after declaration, and const object testing with assignment upon declaration.
17. Test drivers must NOT include console or file input, unless directed otherwise.
18. Do NOT allow memory leaks, unless specifically instructed otherwise.