Topic 10

1. Functions as black boxes
2. Implementing functions
3. Parameter passing
4. Return values
5. Functions without return values
6. Reusable functions
7. Stepwise refinement
8. Variable scope and globals
9. Reference parameters
10. Recursive functions

Recursive Functions

- A recursive function is a function that calls itself.
- Recursion may provide a simpler implementation than a function that iterates (loops) to calculate an answer
  - By calling itself (and the new copy calling ***itself***), multiple iterations are automatically created and handled by the computer hardware's function-call-stack mechanism

- For example, to print a text triangle:

```
[]
[][]
[][][]
[][][][]
```

- Using a function we'll define as:
  - `void print_triangle(int side_length)`

Recursive Function Example

```
[]
[][]
[][][]
[][][][]
```

- The function call would be:

```
print_triangle(4);
```

- Pseudocode of a recursive version, for an arbitrary side length:

If side length < 1, return.
Else, call print_triangle with side length = side length -1.
Then print a line consisting of side length [] symbols

Recursive Function C++ Code

```
void print_triangle(int side_length)
{
   if (side_length < 1) { return; }
   print_triangle(side_length - 1);
   for (int i = 0; i < side_length; i++)
   {
      cout << "[]";
   }
   cout << endl;
}
```

A recursive function works by calling itself with successively simpler input values

Recursive Function Rules

Two requirements ensure that the recursion is successful:

1. Every recursive call must simplify the task in some way.
2. There must be special cases to handle the simplest tasks directly.

The print_triangle function calls itself again with smaller and smaller side lengths. Eventually the side length must reach 0, and the function stops calling itself.

Tracing the Calls to the Recursive Function

- The call print_triangle(4) calls print_triangle(3).
  - The call print_triangle(3) calls print_triangle(2).
    - The call print_triangle(2) calls print_triangle(1).
      - The call print_triangle(1) calls print_triangle(0).
        - » The call print_triangle(0) returns, doing nothing.
      - The call print_triangle(1) prints [].
    - The call print_triangle(2) prints [][].
  - The call print_triangle(3) prints [][][].
- The call print_triangle(4) prints [][][][].

How to Think Recursively

- Just focus on reducing the problem to a simpler one with a call to the same function with smaller inputs
  - And include the exit case with no additional call, when the input reaches the limit, so that the recursion eventually stops
- Example: summing the digits of an `int` input such as 1729:

1. Break the input into parts that can themselves be inputs to the problem
   - Save & remove the last digit and re-call with the remaining digits as input
     - Save it with `%10` and remove it with `/10`.
2. Combine the solutions with simpler inputs into a solution of the original problem.
   - Total that saved digit plus the return from the call
   - Return the total
3. Find solutions to the simplest inputs (the stopping points).
   - Terminate recursion when input=0
4. Combine the simple cases and the reduction step.

How to Think Recursively: the Code and a Trace

```
int digit_sum(int n)
{
   // Special case for terminating the recursion
   if (n == 0) { return 0; }
   // General case
   return digit_sum(n / 10) + n % 10;
}
```

- The call `digit_sum(1729)` calls `digit_sum(172)`.
  - The call `digit_sum(172)` calls `digit_sum(17)`.
    - The call `digit_sum(17)` calls `digit_sum(1)`.
      - The call `digit_sum(1)` calls `digit_sum(0)`.
        - » The call `digit_sum(0)` returns 0.
      - The call `digit_sum(1)` returns 1.
    - The call `digit_sum(17)` returns 1+7 = 8
  - The call `digit_sum(172)` returns 8+2 = 10
- The call `digit_sum(1729)` resumes and returns 10+9 = 19.

CHAPTER SUMMARY (Part 1)

- Understand functions, arguments, and return values.
  - A function is a named sequence of instructions.
  - Arguments are supplied when a function is called.
  - The return value is the result that the function computes.
- Be able to implement functions.
  - When defining a function, you provide a name for the function, a variable for each argument, and a type for the result.
  - Function comments explain the purpose of the function, the meaning of the parameter variables and return value, as well as any special requirements.
- Describe the process of parameter passing.
  - Parameter variables hold the argument values supplied in the function call.
- Describe the process of returning a value from a function.
  - The return statement terminates a function call and yields the function result.

CHAPTER SUMMARY (Part 2)

- Design and implement functions without return values.
  - Use a return type of void to indicate that a function does not return a value.
- Develop functions that can be reused for multiple problems.
  - Eliminate replicated code or pseudocode by defining a function.
  - Design your functions to be reusable. Supply parameter variables with values that can vary when the function is reused.
- Apply the design principle of stepwise refinement.
  - Decompose complex tasks into simpler ones.
  - When you discover that you need a function, write a description of the parameter variables and return values.
  - A function may require simpler functions to carry out its work.

CHAPTER SUMMARY (Part 3)

- Determine the scope of variables in a program.
  - The scope of a variable is the part of the program in which it is visible.
  - A variable in a nested block shadows a variable with the same name in an outer block.
  - A local variable is defined inside a function. A global variable is defined outside a function.
  - Avoid global variables in your programs.
- Describe how reference parameters work.
  - Modifying a value parameter has no effect on the caller.
  - A reference parameter refers to a variable that is supplied in a function call.
  - Modifying a reference parameter updates the variable that was supplied in the call.
- Understand recursive function calls and implement recursive functions.
  - A recursive computation solves a problem by using the solution of the same problem with simpler inputs.
  - For a recursion to terminate, there must be special cases for the simplest inputs.
  - The key to finding a recursive solution is reducing the input to a simpler input for the same problem.
  - When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.