Topic 9

Problem Solving: Solving a Simpler Problem First

When you face a complex task:

1. simplify the problem
2. solve the simpler problem first
3. Apply what you learned to build the complete, complex task solution

This methodology helps you get started, and your success on the simpler problem will help motivate you to solve the harder one.

Solving a Simpler Problem

<u>The Task</u>:

Arrange pictures, aligned at the top edges, separated with small gaps. Start a new row once the arrangement reaches a certain width.

Solve these simpler problems to build towards a final solution:

1. Draw one picture.
2. Draw two pictures next to each other.
3. Draw two pictures with a gap between them.
4. Draw all pictures in a long row.
5. Draw a row of pictures until you run out of room, then put one more picture in the next row.

See the textbook Section 4.9 for details!

Topic 10

Random Numbers and Simulations

A *simulation program* models an activity in the real world (or in an imaginary one)

Commonly used for:

1. predicting climate change
2. analyzing traffic
3. picking stocks
4. many other applications in science and business.

.

Simulations and the `rand` Function

Since things in the real world happen at random times and with random magnitudes within a certain range, we need to generate random numbers for simulations.

`<cstdlib>` has a random number function: **`rand()`**

- **`rand()`** `returns` a random `int` between 0 and **`RAND_MAX`**
  - implementation-dependent constant, typically the largest valid `int`
- Call **`rand()`** again, and you get a different number.

The `rand` Function is "Pseudorandom"

`rand` calculates a value from a long sequence of numbers that don't repeat for a long time.

But they do eventually repeat, thus the name "*pseudorandom numbers".*

Furthermore, if you run the program again, you get the *exact same sequence of pseudorandom numbers.*

To prevent the repeated sequences, use the following nested function call at the top of your program once, to "seed" the **`rand`** function with a unique value:

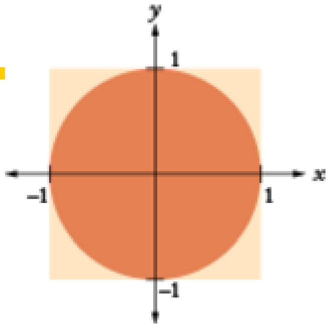
```
srand(time(0)); //time of day is the seed
// time() requires #include <ctime> header
```

Simulation Example: Die Tosses

- Usually we want random numbers in a certain range (1 to 6 for dice)
- To get a random number value between a & b, use
  - `int r = rand() % (b - a + 1) + a;`
- Complete die-toss program: sec10/dice.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main() {
   srand(time(0)); //seed the generator
   for (int i = 1; i <= 10; i++) // 10 tosses
   {
         int d1 = rand() % 6 + 1;
         int d2 = rand() % 6 + 1;
         cout << d1 << " " << d2 << endl;
   }
   cout << endl;
   return 0;
}
```

Simulations: the Monte Carlo Method for π



- Named after the famous casino in Monte Carlo
- For finding approximate solutions to problems that cannot be precisely solved
  - by doing a bunch of trials and tallying averages
- Example for approximating the value of pi:
  - Simulate shooting a dart into a square surrounding a circle of radius 1.
    - generate random x and y coordinates between –1 and 1
    - The dart hits inside the circle if $x^2 + y^2 \leq 1$
    - Because our shots are entirely random, the ratio of hits / tries is approximately equal to the ratio of the areas of the circle and the square, that is, $\pi/4$.
    - Therefore, our estimate for π is $4 \times$ hits/tries.

Code for Monte Carlo Simulation of π

```
// sec10/montecarlo.cpp
// #includes not shown, insert here if you run this code
int main()
{
    const int TRIES = 10000; //can increase TRIES for more accuracy
    srand(time(0));
    int hits = 0;
    for (int i = 1; i <= TRIES; i++)
    {
        double r = rand() * 1.0 / RAND_MAX; // Between 0 and 1
        double x = -1 + 2 * r; // x in range -1 to 1
        r = rand() * 1.0 / RAND_MAX; //rand value for y
        double y = -1 + 2 * r;
        if (x * x + y * y <= 1) //hit inside circle
            { hits++; }
    }
    double pi_estimate = 4.0 * hits / TRIES;
    cout << "Estimate for pi: " << pi_estimate << endl;
    return 0;
}
```

Topic 11

1. The `while` loop
2. Problem solving: hand-tracing
3. The `for` loop
4. The `do` loop
5. Processing input
6. Problem solving: storyboards
7. Common loop algorithms
8. Nested loops
9. Problem solving: solve a simpler problem first
10. Random numbers and simulations
11. Chapter summary

Chapter Summary, Part 1

- Explain the flow of execution in a loop.
  - Loops execute a block of code repeatedly while a condition remains true.
  - An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.
- Use the technique of hand-tracing to analyze the behavior of a program.
  - a simulation of code execution in which you step through instructions and track the values of the variables.
  - helps you understand how an unfamiliar algorithm works.
  - can show errors in code or pseudocode. Use for loops for implementing counting loops.
- Choose between the `while` loop and the `do` loop.
  - The `while` loop is for loops that only should run if the condition is true at the beginning
  - The `do` loop is appropriate when the loop body must be executed at least once, such as prompting the user to enter correct input.
- The `for` loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.

Chapter Summary, Part 2

- Implement loops that read sequences of input data.
  - A sentinel value denotes the end of a data set, but it is not part of the data.
  - You can use a Boolean variable to control a loop. Set the variable before entering the loop, then set it to the opposite to leave the loop.
  - Use input redirection to read input from a file. Use output redirection to capture program output in a file.
- Use the technique of storyboarding for planning user interactions.
  - A storyboard consists of annotated sketches for each step in an action sequence.
  - Developing a storyboard helps you understand the inputs and outputs required for a program.
- Know the most common loop algorithms.
  - To compute an average, keep a total and a count of all values.
  - To count values that fulfill a condition, check all values and increment a counter for each match.
  - To find a match, exit the loop when the match is found.
  - To find the largest value, update the largest value seen so far whenever you see a larger one.
  - To compare adjacent inputs, store the preceding input in a variable.

Chapter Summary, Part 3

- Use nested loops to implement multiple levels of iteration.
  - When the body of a loop contains another loop, the loops are nested.
  - A typical use of nested loops is printing a table with rows and columns.

- Design programs that carry out complex tasks.
  - To solve a complex problem, first solve a simpler task.
  - Make a plan consisting of a series of tasks, each a simple extension of the previous one, and ending with the original problem.

- Apply loops to the implementation of simulations.
  - In a simulation, you use the computer to model an activity.
  - You can introduce randomness by calling the random number generator `rand()` after seeding the generator by calling `srand(time(0))`