CS1010E Lecture 4

Nested Control Structures

Henry Chia (hchia@comp.nus.edu.sg)

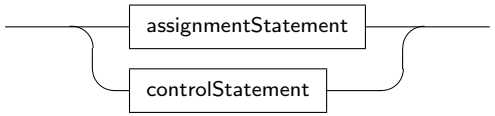
Semester 1 2016 / 2017

Lecture Outline

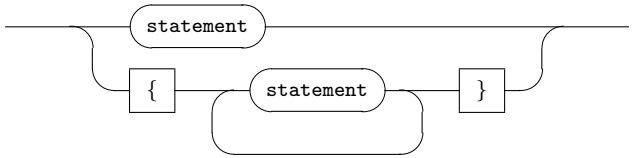
- Nesting control statements
  - Nested loops
- Pattern printing exercises
- Kolb experiential learning model
- Coding standard
- Testing and Debugging
  - Incremental testing
  - Positive versus negative testing
  - Black-box versus white-box testing
  - Automating testing

Nested Control Statements

statement



stmtBlock



- Nested loops: a repetition statement can be nested within the statement block of another repetition statement

Example: Multiplication Table

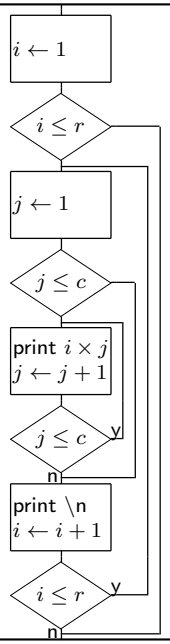
- Consider the following 4×5 multiplication table

$\times$	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20

- Write a program to generate a $r \times c$ multiplication table
  - Use one loop to generate a series of values in a row
  - Wrap the above loop around another loop to generate a table of values

Nested Loops

```c
#include <stdio.h>

int main(void) {
   int r = 0, c = 0;
   int i = 0, j = 0;

   printf("Enter r c: ");
   scanf("%d%d", &r, &c);

   for (i = 1; i <= r; i++) {
      for (j = 1; j <= c; j++) {
         printf("%d ", i * j);
      }
      printf("\n");
   }

   return 0;
}
```



Timeline Tracing – Nested Loops

- Trace of variables i and j for $r = 2$ and $c = 5$

```c
for (i = 1; i <= r; i++) {
   for (j = 1; j <= c; j++) {
      printf("%d ", i * j);
   }
   printf("\n");
```

i	1						2						3
j	⋮	1	2	3	4	5	6	⋮	1	2	3	4	5	6	⋮
s	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

$\rightarrow t$

- Related task: compute the value of $\sum_{i=1}^{r} \sum_{j=1}^{c} i \times j$

Example: Triangular Multiplication Table

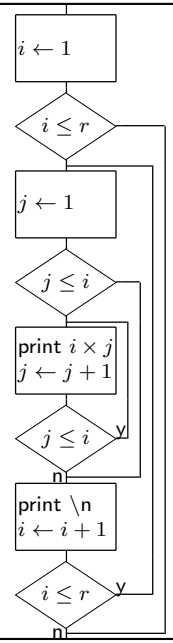
- Consider the following 5×5 multiplication table where only the lower-triangular portion of the table is shown

×	1	2	3	4	5
1	1				
2	2	4			
3	3	6	9		
4	4	8	12	16	
5	5	10	15	20	25

- Task: Generate a $r \times r$ triangular multiplication table
- Related task: Sum of all values in the above table

Example: Triangular Multiplication Table

```c
#include <stdio.h>

int main(void) {
   int r = 0;
   int i = 0, j = 0;

   printf("Enter r: ");
   scanf("%d", &r);

   for (i = 1; i <= r; i++) {
      for (j = 1; j <= i; j++) {
         printf("%d ", i * j);
      }
      printf("\n");
   }

   return 0;
}
```



Pattern Printing

- Pattern output is a good way to test one's competence with nested loop constructs
- Write a program that takes in an integer n and prints a lower left triangle of n rows
- Example, when $n = 5$

```
Enter n: 5
*
**
***
****
*****
```


Pattern Printing

```c
#include <stdio.h>

int main(void) {
   int n = 0;
   int r = 0, c = 0;

   printf("Enter n: ");
   scanf("%d", &n);

   for (r = 1; r <= n; r++) {
      for (c = 1; c <= r; c++) {
         printf("*");
      }
      printf("\n");
   }

   return 0;
}
```


Pattern Printing

- How about this?

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

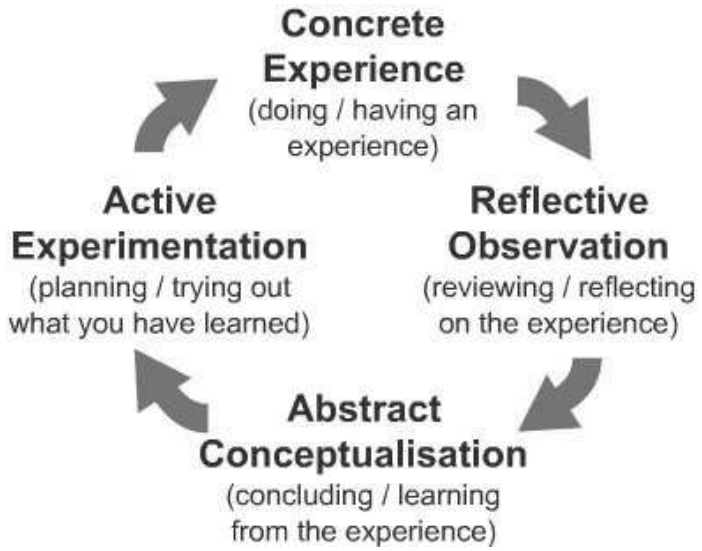
- Develop the solution incrementally

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


Pattern Printing

```c
#include <stdio.h>

int main(void) {
   int n = 0, r = 0, c = 0;
   int numSpaces,numBlanks;

   printf("Enter n: ");
   scanf("%d", &n);

   for (r = n; r >= 1; r--) {
      numSpaces = r;
      numBlanks = n - numSpaces;

      for (c = 1; c <= numBlanks; c++) {
         printf(" ");
      }

      for (c = 1; c <= 2 * numSpaces - 1; c++) {
         printf("*");
      }

      printf("\n");
   }

   return 0;
}
```

Kolb Experiential Learning Model



Kolb Experiential Learning Model

- Given a complex programming task, start with a simpler and more manageable problem
  - **Concrete experience** — The novice programmer writes a program to *physically experience* how programming constructs used within the program work in the context of the problem
  - **Reflective observation** — This physical experience forms the basis for observation and reflection, so that he has the opportunity to consider what works and fails
  - **Abstract conceptualization** — The learner thinks about ways to improve on the next attempt at solving the same problem, or how to make use of the current solution for the next higher level problem
  - **Active experimentation** — Every new attempt to write programs to solve a problem incrementally is informed by a cyclical pattern of previous experience, though and reflection

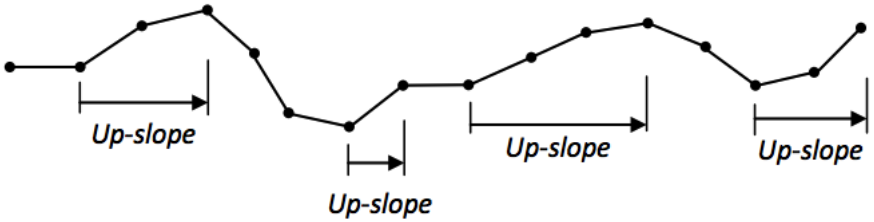
Kolb Experiential Learning Model

- In order to gain genuine knowledge from an experience,
  - the learner must be willing to be actively involved in the experience;
  - the learner must be able to reflect on the experience;
  - the learner must be able to apply analytical stills in conceptualizing the experience; and
  - the learner must be able to utilize decision-making and problem-solving skills in order to transform the new ideas gained from the experience into action

Coding Standard

- Naming:
  - Identifier: camelCasing
  - Constant: ALL_CAPS_UNDERSCORE_SEPARATED
- Look and feel:
  - Blank lines to separate groups of related statements
  - Single space added before and after assignment, as well as each operator
  - Place opening braces on the same line; closing braces on a new line
  - Use constants in place of “magic numbers”
  - Comment on “what it does” and not “how it is done”
  - Comments indented at the same level as the code

Exercise



- An up-slope is defined as a contiguous group of heights of *increasing* values
- Find the number of up-slopes given a list of non-negative values representing heights terminated by a negative value

  30 30 40 44 30 20 18 28 32 36 37 35 27 29 36 -1

Testing and Debugging

- Incremental testing
  - Testing as the program is incrementally developed
  - As opposed to "big bang" testing
  - Regression testing
    - Rerun test cases that have previously been passed whenever a change is made to the code
- Debugging
  - Tracing the program fragment by hand
  - Using `printf` statements to "watch" variable changes
  - Using a debugger (not required in this course)

Positive versus Negative Testing

- Positive testing
  - Testing of functionality that is expected to work
  - Test cases:
    - 0 0 -1
- Negative testing
  - Testing cases that is expected to fail and how they are handled
  - Test cases:
    - -1
    - 0 -1

```c
#include <stdio.h>
#define INVALID -1

int main(void) {
   int prevHeight, currHeight;
   int numUpslopes = INVALID;

   scanf("%d", &prevHeight);
   if (prevHeight >= 0) {

      scanf("%d", &currHeight);
      if (currHeight >= 0) {
         numUpslopes = 0;
      }
   }

   if (numUpslopes == INVALID) {
      printf("INVALID\n");
   } else {
      printf("VALID\n");
   }

   return 0;
}
```


Counting Up-slopes

```c
numUpslopes = 0;
upslope = false;

while (currHeight >= 0) {

   if (prevHeight >= currHeight)
      upslope = false;
   else {
      if (!upslope) {
         numUpslopes++;
         upslope = true;
      }
   }

   prevHeight = currHeight;
   scanf("%d", &currHeight);
}
```

Black-Box Testing

- Develop test cases by treating functional code as a blackbox
- Awareness of the code can bias the test cases we create
- Use specification to test the code
  - Test cases consisting of two height values:
    - No up-slope: `0 0 -1`
    - No up-slope: `1 0 -1`
    - One up-slope: `0 1 -1`
  - Test cases consisting of three height values:
    - One up-slope: `0 0 1 -1`
    - One up-slope: `0 1 0 -1`
    - One up-slope: `0 1 1 -1`
    - One up-slope: `1 0 1 -1`
    - No up-slope: `0 0 0 -1`
    - No up-slope: `1 0 0 -1`
    - No up-slope: `1 1 0 -1`

White-Box Testing

- Path coverage — create test cases so that all logical paths of the program can be examined

```
if (prevHeight >= currHeight)
   upslope = false;
else {
   if (!upslope) {
      numUpslopes++;
      upslope = true;
   }
}
```

- Path #1 tests (not up-slope)
  - `..0 0..`
  - `..1 0..`
- Path #2 tests (begin up-slope)
  - `..0 0 1..`
  - `..1 0 1..`
- Path #3 tests (still up-slope)
  - `..0 1 2..`

Automating Testing

- Prepare a test case by writing two files (say using `vim`)
  - Test input file
    - Write the input into a file, e.g. `test1.in`
  - Test output file
    - Write the output into a file, e.g. `test1.out`
- Use the `diff` command to compare the actual output of the program with the expected output from the test output file

  `$ ./a.out < test1.in | diff - test1.out`
- The above command redirects the test input file into the executable, and the output of the executable is compared with the test output file

Summary

- Appreciate how loops can be nested
- Appreciate how control statements can be nested within other control statements
- Application of timeline tracing to nested loops
- Adopt an experiential learning model in programming
- Adopt a coding standard to enhance readability of programs
- Able to generate suitable test cases to test program functionality, and debug if necessary
- Use of test automation to simplify the process of testing