Exercise 4: Iterative Statements

Nanda H Krishna

22 February 2018

1 Pattern printing

Problem description: Define a function `indent(n)` to print `n` times the pattern (|–) in a line. Construct an input file in which each line is a pair of numbers referred to as `level` and `key`. For each line print `level` number of times the pattern |– followed by the `key`.

Specification: The function `indent()` takes an integer `n` as the parameter which is `level` and prints on `stdout` as many of the pattern (|–).

Prototype:

```
void indent(int n)
```

Program design: The program consists of `indent(int n)` which prints the pattern `n` times on `stdout` and `main()` which reads the input from the file for `level` and `key`, and calls `indent()` and prints the `key` on `stdout`.

Algorithm: The algorithm to print the pattern is as follows:

```
indent(n):
  for i in range(n):
    print("|--")
```

Program:

```
#include<stdio.h>
void indent(int n) // Prints the pattern
{
  for(int i = 0; i < n; i++)
    printf("|--");
}
int main()
{
  int level, key;
  while(scanf("%d %d", &level, &key) != EOF) {
    indent(level);
    printf("%d\n", key); // Print key
  }
  return 0;
}
```

Test Input:

```
0 0
1 10
2 20
3 30
4 40
5 50
```

Output:

```
0
|--10
|--|--20
|--|--|--30
|--|--|--|--40
|--|--|--|--|--50
```

2 Length of an array

Problem description: Represent a list of numbers by an array of numbers terminated by -1 as the end of list marker. Define a function int $array_{len}$(int a[]) that takes such an array as the parameter and returns the length of the array, that is, the number of items in the array.

Specification: The function `array_len()` takes the integer array as input and returns the length to the calling place.

Prototype:

```
int array_len(int a[])
```

Program design: There are 2 functions, `array_len(int a[])` which calculates the array length and `main()` which gets the inputs and calls the function for testing.

Algorithm: The algorithm to calculate array length is as follows:

```
array_len(int a[]):
  i = 0
  while a[i] is -1:
    i++
```

Program:

```
#include<stdio.h>
int array_len(int a[])
{
  int i = 0;
  while(a[i++] != -1); // Calculates array length
  return i - 1;
}
int main()
{
  int l[101];
  for(int i = 0, a = 0; a != -1; i++) {
    scanf("%d", &a); // Gets element of array
    l[i] = a;
  }
  printf("%d", array_len(l));
  return 0;
}
```

Test Input:

```
1 2 3 4 5 6 7 8 9 0 -1
```

Output:

```
10
```

3 Reading input data

3.1 List of numbers

Problem description: Read a list of numbers using `scanf()`. Format the list of numbers as a line of numbers in the input file. Read the n items with a loop. In each iteration of the loop, let `scanf()` read one number.

Specification: The function `main()` contains a loop which contains a `scanf()` statement, to read numerical inputs one by one till no more input is given.

Program design: Function `main()` reads the numbers into the list, and prints the list for verification.

Program:

```
#include<stdio.h>
int main()
{
  int a[100], i = 0;
  while(scanf("%d", &a[i]) != EOF) { // Get element
    printf("%d ", a[i]);
    i++; // Print element
  }
  return 0;
}
```

Test Input:

```
1 2 3 4 5 6 7 8 9 0
```

Output:

```
1 2 3 4 5 6 7 8 9 0
```

3.2 List of tuples

Problem description: Read a list of k-tuples. Format a k-tuple of items as a line of `k` items, and the list as a sequence of lines, in the input file, as illustrated below. Read the list with a loop. In each iteration, let `scanf()` read `k` items from the file.

Specification: Function `main()` contains a loop with `scanf()` which reads `k` and and `k` numbers into the tuple, till no more inputs are given.

Program design: Function `main()` reads the list of tuples from the file and prints for verifictaion, on `stdout`.

Program:

```
#include<stdio.h>
int main()
{
  int k, n, i = 0;
  while(scanf("%d", &k) != EOF) { // Assumes k for the tuple given as
  // the first element of line
    i = 0;
    while(i < k) {
      scanf("%d", &n);  // Read tuple element
      printf("%d ", n); // Print the element
      i++;
    }
    printf("\n");
  }
  return 0;
}
```

Test Input:

```
1 2
2 3 5
3 4 5 6
5 1 2 3 4 5
```

Output:

```
2
3 5
4 5 6
1 2 3 4 5
```

3.3 List of lists

Problem description: Read a list of lists. Format a simple list as a line of items. Read from `stdin`.

Specification: Function `main()` contains a double-loop to read the list of lists element-wise, using the `fgets()` and `sscanf()` statemets till no more inputs are given.

Program design: Function `main()` reads the list of lists and also prints it on `stdout` for verification.

Program:

```
#include<stdio.h>
int main()
{
  char line[100]; // Max length of list is 100
  int nbytes, l = 0, i = 0;
  int a[100][100]; // Max 100 lists
  while(fgets(line, 100, stdin)) {
    char *s = &line[0];
    for(i = 0; sscanf(s,"%d%n", &a[l][i], &nbytes) && line[i] != -1; i++)
      s+=nbytes;
    l++;
  }
  for(int c = 0; c < l; c++) {
    for(i = 0; a[c][i] != -1; i++)
      printf("%d ", a[c][i]);
    printf("\n");
  }
  return 0;
}
```

Test Input:

```
10 20 30 -1
20 -1
30 40 50 60 -1
```

Output:

```
10 20 30
20
30 40 50 60
```

4 Print a sub-array

Problem description: Print a sub-array. Write a function `print_array(a, low, high)` that prints the subarray `a[low:high]`, that is, the items of array `a` from `low` to `high`, respectively called the lower bound and upper bound of the subarray. We follow the convention of upper bound excluded.

Specification: The function `print_array()` takes the array, lower bound and upper bound as inputs and prints `a[l:h]` on `stdout`.

Prototype:

```
void print_array(int a[], int l, int h)
```

Program design: The program consists of `print_array(int a[], int l, int h)` which prints the sub-array on `stdout` and `main()` which reads input and tests the function.

Algorithm: The algorithm to print a sub-array is as follows:

```
print_array(a[], l, h):
  for i in range(l, h):
    print a[i]
```

Program:

```
#include<stdio.h>
```

```
void print_array(int a[], int l, int h)
{
  for(int i = l; i < h; i++)
    printf("%d ", a[i]);
}
int main()
{
  int a[10], l, h;
  for(int i = 0; i < 10; i++) scanf("%d", &a[i]);
  scanf("%d %d", &l, &h);
  print_array(a, l, h);
  return 0;
}
```

Test Input:

```
0 1 2 3 4 5 6 7 8 9
1 6
```

Output:

```
1 2 3 4 5
```

5 Sum, mean and variance

Problem description: Write functions to sum an array, find its mean and variance and also the number of items greater than the mean.

Specification: The function `sum()`,`mean()`,`variance()` and `count()` all take the array, lower bound and upper bound as inputs and return the sum, mean, variance and number of items above mean value in the sub-array respectively.

Prototype:

```
int sum(int a[], int l, int h)
float mean(int a[], int l, int h)
float variance(int a[], int l, int h)
int count(int a[], int l, int h)
```

Program design: The functions are `sum(int a[], int l, int h)` for sum of a sub-array, `mean(int a[], int l, int h)` to find mean of a[l:h], `variance(int a[], int l, int h)` for the variance and `count(int a[], int l, int h)` for counting the number of items above the mean. Function `main()` calls these for testing and also receives user inputs.

Program:

```
#include<stdio.h>
int sum(int a[], int l, int h)
{
  int s = 0;
  for(int i = l; i < h; i++)
    s += a[i];
  return s;
}
float mean(int a[], int l, int h)
{
  return sum(a, l, h)/(1.0*(h - l));
}
float variance(int a[], int l, int h)
{
  float m = mean(a, l, h), s = 0;
  for(int i = l; i < h; i++) {
    s += ((a[i] - m)*(a[i] - m));
  }
  return s/(1.0*(h - l));
}
int count(int a[], int l, int h)
{
  float m = mean(a, l, h);
  int s = 0;
  for(int i = l; i < h; i++) {
    if(a[i] > m) s++;
  }
  return s;
}
int main()
{
  int a[5], l, h;
  for(int i = 0; i < 5; i++) {
    scanf("%d", &a[i]);
  }
  scanf("%d %d", &l, &h);
  printf("%d %f %f %d", sum(a, l, h), mean(a, l, h), variance(a, l, h), count(a,
  return 0;
}
```

Test Input:

```
1 2 3 4 5 1 4
```

Output:

```
9 3.000000 0.666667 1
```

6 Prime numbers

Problem description: Define a function `is_prime(n)` that tests whether a non-negative integer `n` is a prime number and returns `true` if `n` is prime and `false` if `n` is not prime.

Specification: The function to check if the number is prime takes the number as input and returns 1 if prime, 0 if not prime.

Prototype:

```
int is_prime(int n)
```

Program design: The function `is_prime(int n)` checks if a number is prime, while `main()` tests it for numbers from 1 to 100.

Algorithm: The algorithm to check for prime number is as follows:

```
is_prime(n):
  factors = 0
  for i in range(1, n+1):
   if n%i == 0:
     factors++
  if factors is 2 then prime
```

Program:

```
#include<stdio.h>
int is_prime(int n)
{
  int factors = 0;
  int i = 1;
  while(i <= n) {
      if(n % i == 0) factors++;
      i++;
  }
  if(factors == 2)
    return 1;
  else
    return 0;
}
int main()
{
  for(int i = 2; i <= 100; i++) {
    if(is_prime(i))
    printf("%d ",i);
  }
  return 0;
}
```

Output:

```
2 3 5 7 11 13
17 19 23 29 31
37 41 43 47 53
59 61 67 71 73
79 83 89 97
```

7 Search in an array

Problem description: Write functions to implement linear and binary search in an array. (Array is sorted)

Specification: For linear search, the array, length and target are parameters and the index is returned if found, else `n`. Binary search returns index if found, -1 if not found.

Prototype:

```
int linear_search(int a[], int n, int t)
int linear_search_n(int a[], int n, int t)
int binary_search(int a[], int n, int t)
```

Program design: The three search functions include linear search with break, without break and binary search. The `main()` tests all three to verify working.

Algorithm: The algorithm to search is as follows:

```
linear_search(a[], n, t):
  for i in range(n):
    if a[i] is t then break
  return i
linear_search_n(a[], n, t):
  for i in range(n) and a[i] is not t:
    i++
  return i
binary_search(a[], n, t):
  l = 0, u = n - 1, flag = 0
  while l <= u and flag is 0
    mid = (l + u)//2
    if a[mid] is t then flag = 1
    elif a[mid] lesser than t then l = mid + 1
    else u = mid
  if flag is zero return -1 else return mid
```

Program:

```
#include<stdio.h>
int linear_search(int a[], int n, int t)
{
  int i = 0;
  for(i = 0; i < n; i++) {
    if(a[i] == t) break;
  }
  return i;
}
int linear_search_n(int a[], int n, int t)
{
  int i = 0;
  while(i < n && a[i] != t) i++;
  return i;
}
int binary_search(int a[], int n, int t)
{
  int l = 0, u = n - 1, flag = 0, mid;
  while(l <= u && flag == 0) {
    mid = (l + u)/2;
    if(t == a[mid]) flag = mid;
    else if(a[mid] > t) u = mid;
    else l = mid + 1;
  }
  if(flag == 0) return -1;
  else return flag;
}
int main()
{
  int a[100], n, t;
  scanf("%d", &n);
  for(int i = 0; i < n; i++) {
    scanf("%d", &a[i]);
  }
  scanf("%d", &t);
  printf("%d ",linear_search(a,n,t));
  printf("%d ",linear_search_n(a,n,t));
  printf("%d ", binary_search(a,n,t));
  return 0;
}
```

Test Input:

```
5 1 2 3 4 5 3
```

Output:

```
2 2 2
```

8 Minimum

Problem description: Write a function to return the index of the smallest item in the sub-array a[low:high], in an array a[0:n] of comparable items.

Specification: The function `minimum()` takes the array, low, high as inputs and returns the index of the smallest item of the sub-array.

Prototype:

```
int minimum(int a[], int l, int h)
```

Program design: The function `minimum(int a[], int l, int h)` returns the index of the minimum of a[l:h], and `main()` obtains inputs and tests the function.

Algorithm: The algorithm to find minimum is as follows:

```
min = l
for i in range(l, h):
  if a[min] > a[i] then min = i
```

```
return i
```

Program:

```
#include<stdio.h>
int minimum(int a[],int l, int h)
{
  int min = l;
  for(int i = l; i < h; i++) {
      if(a[min] > a[i]) min = i;
  }
  return min;
}
int main()
{
    int a[10], l, h;
    for(int i = 0; i < 10; i++) scanf("%d", &a[i]);
    scanf("%d %d", &l, &h);
    printf("%d", a[minimum(a, l, h)]);
    return 0;
}
```

Test Input:

```
9 8 7 6 5 4 3 2 1 0 2 7
```

Output:

```
3
```

9 Armstrong numbers

Problem description: Write functions to:

- Convert an integer to an array of single digit numbers.
- Find the cube of a number.
- Check if a number is an Armstrong number or not.

Specification: The function `to_digits()` takes inputs as the integer and the array for storing, outputs being the length of array returned and the array itself being stored with each elemnt as an individual digit of the integer. The function `cube()` takes a number as the input and returns its cube. The function `is_armstrong()` returns `true` if the number passed is Armstrong, and `false` if not. The `main()` tests all the functions.

Prototype:

```
int to_digits(int n, int s[])
int cube(int n)
bool is_armstrong(int n)
```

Program design: The three functions `to_digits()`,`cube()` and `is_armstrong()` form the heart of the program, with the first two being used by the last to identify an Armstrong number. The `main()` receives inputs and is used for testing.

Algorithm: The algorithm for converting integer to array is as follows:

```
to_digits(int n, int s[]):
  a = n, r = 0
  while a is not 0:
    r = r * 10 + a % 10
    a //= 10
  i = 0
  while r is not 0:
    s[i] = r % 10
    r //= 10
    i++
  s[i] = -1
  return i
```

Program:

```
#include<stdio.h>
#include<stdbool.h>
int to_digits(int n,int s[]) // Makes an array with the digits
{
  int a = n, r = 0;
  while(a != 0) {
    r = r * 10 + a % 10;
    a /= 10;
  }
  int i = 0;
  while(r != 0) {
    s[i] = r % 10;
    r /= 10;
    i++;
  }
  s[i] = -1;
  return i;
}
int cube(int n) // Returns n^3
{
  return n*n*n;
}
bool is_armstrong(int n) // Checks if n is an Armstrong Number
{
  int s[100], a = 0;
  to_digits(n,s);
  for(int i = 0; s[i] != -1; i++)
    a += cube(s[i]);
  if (a == n)
    return true;
  else
    return false;
}
int main()
{
  int n, k, s[100];
  scanf("%d", &n);
  k = to_digits(n,s);
  for(int i = 0; i < k; i++) printf("%d ", s[i]);
  if(is_armstrong(n)) printf("\nArmstrong number");
  else printf("\nNot an Armstrong number");
  printf("\n");
  for(int j = 100; j <= 500; j++) {
    if(is_armstrong(j)) printf("%d ", j);
  }
  return 0;
}
```

Test Input:

```
351
```

Output:

```
3 5 1
Not an Armstrong number
153 370 371 407
```