

# Tutorial Sheet (September 07, 2018)

## ESC101 – Fundamentals of Computing

---

### Announcements

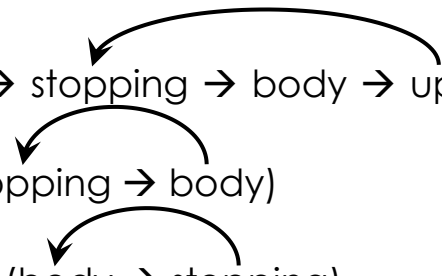
1. **Extra lecture**, Sep 08, 2018 (Saturday), L20, 12PM
  2. **Extra lab for B1, B2, B3**, Sep 08, 2018, CC-01,CC-02, 2PM
  3. **Mid-sem lab exam**, Sep 09, 2018. Please check your session and your assigned room in lecture 15 slides.
  4. Lab exam is open handwritten notes but printouts, photocopies, use of mobile phones, iPads and other communication devices prohibited – will be considered cheating.
- 

### How to find out and correct errors in your code

1. Students are often seen printing only those variables which are the final output of the question. Developing the habit of printing other (non-output) variables to see where code is wrong will help in long run. Remember, in lab exam, no TA help will be provided.
2. For example, if the final output is a sum that is coming out wrong, try print all the values that got added to that sum to find out why the sum is coming out wrong.
3. Executing your own code in your mind (as you do for minor quiz questions of the type “What is the output”) on simple inputs, is a very good way of detecting errors.
4. If you feel some portion of code should be executed but is not getting executed, a very good way is to insert a harmless “Hello World” or some such printf statement in that portion of the code to assure yourself, by viewing the output if that portion of code is indeed getting executed or not.
5. Prutor offers a visualizer (Run > Visualize). Try it out and see how it neatly shows you how different variables in your code change as Mr C executes various statements in your code.

---

## Revision (ask for doubts)

1. **for loop**: syntax, execution (init → stopping → body → update)
  2. **while loop**: syntax, execution (stopping → body)
  3. **do-while loop**: syntax, execution (body → stopping)
  4. Breaking problems into viable subproblems to construct loops
  5. If subproblems themselves require repetition, nested loops!
  6. **break and continue**: legal usage, behavior, illegal usage in conditional statements. Break allowed in switch but not continue
  7. Behavior of break and continue in nested loops
  8. Use of flags to avoid break, continue
- 
- 

## Enumeration in C

Enumeration is a very convenient tool to give human-friendly names to **integer constants** (can't be used for float, double). These can improve readability of code while making no difference to Mr C.

```
enum {No, Yes};
```

will declare **two integer constants** No = 0 and Yes = 1 i.e. if we write `printf("%d",No);` Mr C will print 0

```
enum {FALSE, TRUE}
```

will declare **two integer constants** FALSE = 0 and TRUE = 1 i.e. if we write `printf("%d",TRUE);` Mr C will print 1

```
enum {zero, one, two three}
```

will declare **four integer constants** zero = 0, one = 1, two = 2, three = 3 i.e. if we write `printf("%d",two);` Mr C will print 2

```
enum {mon = 1, tue, wed, thu, fri, sat, sun}
```

will declare **seven integer constants** mon = 1, tue = 2, wed = 3, thu = 4, fri = 5, sat = 6, sun = 7

---

## Sample Questions to discuss

Read  $n$  pairs of numbers that indicate loss and sales of factory for  $n$  days in the format (loss,sale). If loss on any day is greater than 100, print HIGHLOSS. If all sale is greater than 50 on all days, print HIGHSALE.

**Tip:** the two problems are a bit different in how we use flags

1. **Problem 1:** even a single large loss means print HIGHLOSS. Let's use a flag to assume initially, that loss will **not be high** on any day. If later we find out that loss is indeed high on one of the days, we can set that flag to a different value.
2. **Problem 2:** all sales need to be high in order to print HIGHSALE. Let's use a flag to assume initially, that sales will **indeed be high** on all days. If we later find out that sale is not high on one of the days, we will set that flag to a different value.

```
enum {lowLoss, highLoss};
enum {lowSale, highSale};
int loss, sale, i;
int lossFlag = lowLoss; // Initially optimistic – all losses will be low
int saleFlag = highSale; // Initially optimistic – all sales will be high
for(i = 1; i <= n; i++){
    scanf("(%d,%d)", &loss, &sale);
    if(loss > 100) lossFlag = highLoss;
    if(sale < 50) saleFlag = lowSale;
}
if(lossFlag == highLoss) printf("HIGHLOSS");
if(saleFlag == highSale) printf("HIGHSALE");
```

### Warning: don't write

```
if(loss > 100) lossFlag = highLoss; else lossFlag = lowLoss;
```

```
if(sale < 50) saleFlag = lowSale; else saleFlag = highSale;
```

If you do this, lossFlag/saleFlag will only depend on the last loss/sale value and not on the entire stream of loss/sale values.

**Overflow problems: Take three positive integers  $m$ ,  $n$ ,  $p$  and compute  $(m^n) \bmod p$  i.e. the remainder you get when you divide  $m^n$  by  $p$**

This question seems deceptively simple. However, `((int)pow(m,n))%p` may not give the correct result since  $m^n$  may be so large that `pow`, which gives results as double, may distort the number.

However, note that no matter how large  $m^n$  is, the final answer i.e.  $(m^n) \% p$  is always smaller than  $p$  (since it is the remainder of some ridiculously large number when divided by  $p$ ). It is just that we can't hope to first compute  $m^n$  as a long or an int (since they cannot store such large numbers) or even as a float or a double (since they distort very large numbers) and then take `%p` to get the answer.

To solve this problem, we use a trick from number theory (modular arithmetic) that tells us that if  $a$ ,  $b$ ,  $p$  are positive integers, then

$$(a \times b) \% p = ((a \% p) \times (b \% p)) \% p$$

This means we can write  $(m^n) \% p = ((m^{(n-1)} \% p) \times (m \% p)) \% p$

```
int m, n, p, rem = 1, i;  
m = m % p; // Only m % p matters according to above rule  
for(i = 1; i <= n; i++)  
    rem = (rem * m) % p;
```

The above can be done much faster using *iterated exponentiation*.

**Hint:** if  $n$  is even then  $(m^n) \% p = ((m^{(n/2)}) \% p * (m^{(n/2)}) \% p) \% p$   
What to do when  $n$  is odd?

---

## Some Pitfalls and recognizing compiler error messages

1. Do not use `break` and `continue` excessively
  2. Do not use flags excessively either
    - a. Keep comments explaining what flag values mean, or else
    - b. Assign enumerated values to flags, give meaningful names to the enum constants so that flag values make sense
-