Tutorial Sheet (November 02, 2018)

ESC101 – Fundamentals of Computing

Announcement

1. **Extra lecture** on Saturday (November 03, 12 – 1PM)
2. **Extra lab** for Wed batch on Sat (November 03, 2 – 5PM)
3. **End-sem Lab Exam**: logistics similar to midsem lab exam but seating plan and session allocation different – **come 15 min early**
    a. **Morning exam** (Mon, Tue batches) - 10:30 AM - 2 PM
        i. **CC-01**: B2, {B1 even roll numbers}
        ii. **CC-02**: B4, B5, B6
        iii. **CC-03**: B3
        iv. **MATH-LINUX**: B13, {B1 odd roll numbers}
    b. **Evening exam** (Wed, Thu batches) - 2:30 PM - 6 PM
        i. **CC-01**: B9, {B8 odd roll numbers}
        ii. **CC-02**: B7, B10, B11
        iii. **CC-03**: B12
        iv. **MATH-LINUX**: B14, {B8 even roll numbers}

Function Recap (6 rules) – only ask for doubts – don't cover

1. **RULE 1 (variable passing)**: If a variable (char, int, pointers etc) is given as input, the value stored inside that variable gets passed.
2. **RULE 2 (expression passing)**: If an expression is given as input, the value generated by that expression gets passed.
3. **RULE 3 (type mismatch)**: Avoid type mismatches in arguments of functions. Typecasting may get done which may cause errors.
4. **RULE 4 (value copying)**: Values passed (as variable or expression) to a function always get copied to fresh variables, modifying which has no effect on the source of those values.
5. **RULE 5 (return values)**: Values returned by a function may be used freely but with care.

6. **RULE 6 (address rule)**: all clones share the same memory space. If one clone modifies the value stored at a certain address location directly, other clones checking that location will see the new value. **NOTE**: if two clones have a variable with the same name, modifying one variable using its name does not affect the other since those variables are stored at different locations.

Rules of Scope – ask doubts (don't cover in all detail)

Curly braces used to encapsulate a **block of statements**. Have already seen several examples of such blocks of statements

1. **Conditional statements**: if(temp == 0){ … }else{ … }
2. **Loops**: for(i = 0; i < 5; i++){ … }, while(z < 10){ … }
3. **Functions**: int foo(char *str){ … }

We can define blocks as per our convenience too. A few simple rules govern use of blocks and variables defined inside them.

```
// Function block
int main(){
    float a = 42.5, b = 24.2, d = 11.9;
    { // User-defined block
        int a = 10, c = 9; // a shadowed
    }
    // Now a = 42.5 again
    // However, c no longer available
    for(int a = 0; a < 5; ){ // a shadowed
        int b = a++; // b shadowed
    }
    // Now a = 42.5 and b = 24.2 again!
    return 0;
}
```

1. Blocks can be defined inside other blocks.
2. Variables defined inside a block are alive only inside that block. These variables disappear afterwards (int c in above example).
3. If a variable was defined before a block began, it continues to be visible inside that block too (int d in above example).
4. If a variable was defined before a block began but the block defines a variable with the same name (possibly with a different type), then the new variable **shadows** the old variable. Within that block the old variable unavailable. Once block ends, the old variable (with the old type) becomes available again.

Use of Scoping Rules (cover in detail)

Four main practical uses of scoping rules

1. Use scoping to define a local counter variable in for loops. This will make sure you accidentally do not overwrite some other variable with the same variable name.
2. Scoping is how two functions can have variables with the same name but still not cause any interference. With each function there is an associated block. Within that block, names of that function shadow any other variable with the same name.
3. **Global scope**: variables declared outside any function block visible to all functions. WARNING – reckless use of global variables bad programming practice since it makes programs hard to understand.
4. **Static scope**: variables declared with static scope inside functions are not destroyed when function returns. Moreover, their initialization statement is also executed only once.

```
double i = 42.0;
for(int i = 0; i < 4; i++){
    …
}
// Original i = 42.0 intact ☺
```

```
void foo(int a){
    int b = a++;
}
int bar(float a){
    int b = fabs(a);
    return b;'
}
```

```
int global = 12;
int main(void){
    printf("%d", global);
    return 0;
}
```

```
void foo(){
    // executed only once
    static int a = 0;
    printf("%d", a++);
}
```

Structures (ask doubts – don't cover in detail)

Most common use case of global scope is in defining structures which are almost always defined in global scope so that they are visible to all functions which may accept variables of that datatype.

```
struct Point{
    int x, y;
};
int main(){
    …
}
```

Structures help create neat packages of datatypes which can function as a new customized datatype. Just as we have int variables, int arrays, pointers to int, functions returning int values, suppose we define a structure Point as above, can define Point variables, arrays, pointers etc.

```
struct Point p, q;
struct Point arr[3];
struct Point *qtr, *ptr = &p;
qtr = (struct Point*)malloc
(5*sizeof(Point));
free(qtr);
```

Components of a structure called **fields** – accessed using dot operator. For pointer to structure variables, use arrow shortcut.

```
(*ptr).x = 4;
ptr->y = 5;
printf("%d %d", p.x, p.y);
```

Recursion (cover in detail – see lexicographic notes at end)

Recursion is the process of a function calling itself, in an attempt to solve a problem by solving simpler versions of that same problem.

1. Try to think roughly what the simpler problem looks like and how solving it may help solve the bigger problem.
2. Sometimes the simpler problem may have some additional constraints on it (e.g. bigger problem may be "*generate all strings of length k*" but simpler problem may be "*generate all strings of length k-1 starting with an a*").
3. Sometimes several simpler problems have to be solved to solve the original problem (e.g. may have to solve the above simpler problem for strings of length k-1 starting with 'a', then those starting with 'b', then those starting with 'c' and so on).

Q1: Name the Clones (W11, MON)

Generate all strings of length k that contain only the first n letters. Print in lexicographic order.

Trick: Can solve length k problem by solving length k-1 problem ☺

```
void genStr(char* name, int k, int n, int left){
    if(left == 0){
        printf("%s\n", name); // name[k] = '\0';
        return;
    }
    for(int i = 0; i < n; i++){
        name[k - left] = 'a' + i;
        genStr (name, k, n, left - 1);
    }
}
```

Q2: Growth Curve (W11, TUE)

Find all non-decreasing functions over $[0:n-1]$ taking values in $[0, MAX]$. Print functions in lexicographic order.

Trick: Suppose f is a non-dec function on $[0:n-1]$. Then easy to see that f is a non-dec function on $[1:n-1]$ too with additional property that $f(1) \geq f(0)$.

```
void genFun(int *func, int n, int MAX, int i, int y){
    if(i == n){
        printFunc(func, n); // print the function
        return;
    }
    for(; y <= MAX; y++){
        func[i] = y;
        genFun(func, n, MAX, i + 1, y);
    }
}
```

So, one way (there can be other ways too) to solve this problem is to set a value for $f(0)$ and then generate all possible f on $[1:n-1]$ with the additional constraint that $f(1) \geq f(0)$. Then we change the value of $f(0)$ to a new value and repeat the process and so on.

However, since generating all possible f on $[1:n-1]$ is a smaller version of the same problem, we are able to solve this problem using recursion. Note that here we need to solve many smaller problems, one corresponding to each value of $f(0)$ we select. Moreover, these smaller problems have a tiny additional constraint – that $f(1) \geq f(0)$.

Q3: Zig-zag Num (W11 THU)

Generate all k length zig-zag num (k is even) using 1 … n. A num is zig zag if 2nd digit > 1st digit, 3rd < 2nd, 4th > 3rd, 5th < 4th etc e.g. 1425, 1212. Print in lexicographic order.

Trick: if you ignore, for a moment, the first two digits of a ZZ number, the rest of the number is a ZZ number too! But need to take care that the first digit of this shorter ZZ num must be smaller than 2nd digit of longer ZZ num.

```
void zz(char* num, int k, int n, int left, int max){
    if(left == 0){
        printf("%s\n", num); // num[k] = '\0';
        return;
    }
    for(int i = 1; i <= max; i++){
        num[k - left] = '0' + i;
        for(int j = i + 1; j <= n; j++){
            num[k - left + 1] = '0' + j;
            zz(num, k, n, left - 2, j - 1);
        }
    }
}
```

Lexicographic Ordering

Just as given two integers, we can tell if the two are equal or if one is smaller or the other is smaller, we can take two character strings (possibly of different lengths) and tell if they are equal, or if one is "smaller". A simple set of rules follow. Suppose we have two strings str1 and str2. Suppose str1 has m characters and str2 has n characters. Then the following rules are used.

1. If m = 0 and n = 0 (both strings empty) then both declared equal.
2. If m = 0 and n > 0 then str1 is declared smaller.
3. If m > 0 and n = 0 then str2 is declared smaller.
4. If m > 0 and n > 0, then the first character of str1 and str2 are compared (using their ASCII values). Let the first character of str1 be c1 and the first character of str2 be called c2.
    a. If c1 < c2 then str1 is declared smaller.
    b. If c1 > c2 then str2 is declared smaller.
    c. If c1 = c2 then look at the 2nd characters – call them d1, d2.
        i. If both strings don't have a second character then we declare the two strings equal.
        ii. If one string has a second character and one does not then the smaller string is declared smaller.
        iii. If d1 < d2 then str1 is declared smaller.
        iv. If d1 > d2 then str2 is declared smaller.
        v. If d1 = d2 then look at the 3rd characters and so on …

A lexicographic ordering of strings means printing the "smallest" string first then the 2nd smallest string then the 3rd smallest string and so on. Dictionaries are ordered lexicographically. Note that the lexicographic comparison described above can compare strings of different lengths too!

```
// If str1 is smaller return -1. If str2 is smaller
return 1, If both are equal, return 0
int isSmaller(char* str1, char *str2, int m, int n){
    if(m == 0 && n == 0) return 0;
    if(m == 0 && n > 0) return -1;
    if(m > 0 && n == 0) return 1;
    if(str[1] < str[2]) return -1;
    if(str[1] > str[2]) return 1;
    if(str[1] == str[2])
        return isSmaller(str1+1, str2+1, m-1,n-1);
}
```