Faculty of Computer Science, Dalhousie University *19-Oct-2018*

CSCI 2132 — Software Development

Lecture 19: Generating Permutations

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Mergesort implementation and discussion
- Mergesort complexity calculation
- Quicksort vs Mergesort
- Example: Generating permutations

Slide notes:

Planning Permute Algorithm

- The algorithm will be recursive
- Algorithm will keep first k elements fixed and permute the rest
- Algorithm parameters: `Permute(A, k, n)`
- Initial call to the algorithm: `Permute(A, 0, n)`
- Base case: `k == n-1`

Before we start drafting the recursive algorithm, we will first consider what will be the *base case* of the algorithm. The algorithm is recursive based on increasing parameter k, we can use the base case the value when `k==n-1`, since if we have the first $n-1$ elements of the array fixed, the remaining one can have only one value as well and there is only one permutation that needs to be printed.

Having this in mind, we can write our first version of the Permute pseudocode:

```
Permute(A, k, n)   /* version 1 */
INPUT: A - the array containing numbers 1..n in some
           order
       n - the length of array A
       k - (0<=k<=n) indicating k elements A[0]..A[k-1]
           that will be fixed
OUTPUT: prints all permutations of A that keep first k
        elements fixed, and at the end leaves the order
        of elements of A the same as initially found
1: IF k == n-1 THEN print A
2: ELSE
3:   Permute(A, k+1, n)
4:   FOR i = k+1 TO n-1 DO
5:     swap A[k] with A[i]
6:     Permute(A, k+1, n)
7:     swap A[k] with A[i]    /* swap back */
```

If we want a slightly less efficient but tighter algorithm, we can notice that the line 3 in the previous algorithm would become a part of loop 4–7 if we start with `i = k`. In this case, the algorithm becomes:

```
Permute(A, k, n)   /* version 2 */
```

```
INPUT: A - the array containing numbers 1..n in some
           order
       n - the length of array A
       k - (0<=k<=n) indicating k elements A[0]..A[k-1]
           that will be fixed
OUTPUT: prints all permutations of A that keep first k
        elements fixed, and at the end leaves the order
        of elements of A the same as initially found
1: IF k == n-1 THEN print A
2: ELSE
3:   FOR i = k TO n-1 DO
4:     swap A[k] with A[i]
5:     Permute(A, k+1, n)
6:     swap A[k] with A[i]    /* swap back */
```

The operation `swap` was seen previously in the Quicksort algorithm. It swaps two elements of the array, and instead of '`swap A[k] with A[i]`' it could be rewritten as:
`t=A[k]; A[k]=A[i]; A[i]=t`

Let us us look at the C code available at: `~prof2132/public/permute.c-blanks` (with some blanks that need to be filled). In this example, we use a separate 'swap' function to swap two elements of the array.

```
/* Program: permute.c
   Purpose: prints all permutations of numbers 1..LEN
*/
#include <stdio.h>

#define LEN 4

void swap(int array[], int i, int j);
void permute(int array[], int k, int n);

int main() {
  int array[LEN], i;

  for (i = 0; i < LEN; i++)
    array[i] = i + 1;

  permute(array, 0, LEN);
}

/* Function permute prints all permutations of the array 'array' of
   length n, such that the first k elements are not permuted (they
   stay fixed).
 Parameters:
   array - the array of elements to be permuted
   n - the length of array 'array'
   k - the first k elements of array are note permuted
*/
void permute(int array[], int k, int n) {
  int i;

  if ( _______________ ) {
```

```
    for (i = 0; i < n; i++)
      printf("%d ", array[i]);
    printf("\n");
  }
  else {
    for (i = k; i < n; i++) {
      swap(array, k, i);
      permute(array, ________ , _________ );
      swap(array, k, i);
    }
  }
}

/* Function swaps swaps the elements array[i] and array[j] */
void swap(int array[], int i, int j) {
  int temp = array[i];
  array[i] = array[j];
  array[j] = temp;
}
```

17 Multidimensional Arrays as Arguments

Slide notes:

Multidimensional Arrays as Arguments

- Multidimensional arrays can be passed as function arguments
- Compiler must know all dimensions, except optionally the first
- C89 and earlier: Dimensions had to be constant
  - Example: `int f(a[N][M]) {...}` if `N` and `M` are constants
- C99 and later allows non-constant dimensions expressed using parameters and constants; example:
  `int f(int n, int a[][n]);` or
  `int f(int n, int a[][*]);`
- Important than `n` parameter comes before `a` parameter
- Definition examples:
  `int f(int n, int a[][n]) {...` or
  `int f(int n, int a[][n+1]) {...`
- Not valid:
  `int f(int a[][n], int n) {...`

Multidimensional arrays can be passed as function arguments. This is done in an efficient way by passing their reference, which means that any changes made to those arrays will remain after function call. The compiler needs to know the array dimensions, except the first one, so they must be passed as well.

The C89 standard and earlier required that the dimensions of multidimensional array arguments are known at the compile time. The C99 and later standard allow passing of the variable-length multidimensional arrays with non-constant dimensions, as long as the dimensions are expressed using other function arguments.

For example, in C99 or later, we can use the following prototype:
`int f(int n, int a[][n]);` or
`int f(int n, int a[][*]);`
In this second example, the asterisk (‘`*`’) give a clue to the compiler that the length of the array will depend on the

previous parameters in some way.

Code Example (Compiler Error)

```
#include <stdio.h>

/* ERROR: Compiler will not allow this: */
void fun_print(int n, int a[][]);

int main() {
  int n;
  printf("Enter matrix dim: ");
  scanf("%d", &n);
  int a[n][n];
  for (int i=0; i<n; ++i)
    for (int j=0; j<n; ++j)
      a[i][j] = i*n+j;
  fun_print(n, a);
  return 0;
}

void fun_print(int n, int a[][]) {
  for (int i=0; i<n; ++i) {
    for (int j=0; j<n; ++j)
      printf(" %2d", a[i][j]);
    printf("\n");
  }
}
```

Code Example (Correct Version, C99)

```
#include <stdio.h>
void fun_print(int n, int a[][n]);
// Important: n must come before a
// E.g. not valid: void fun_part(int a[][n], int n);

int main() {
  int n;
  printf("Enter matrix dim: ");
  scanf("%d", &n);
  int a[n][n];
  for (int i=0; i<n; ++i)
    for (int j=0; j<n; ++j)
      a[i][j] = i*n+j;
  fun_print(n, a);
  return 0;
}

void fun_print(int n, int a[][n]) {
  for (int i=0; i<n; ++i) {
    for (int j=0; j<n; ++j)
```

```
      printf(" %2d", a[i][j]);
    printf("\n");
  }
}
```

18 Program Organization

18.1 Local Variables

We know that local variables are variables defined inside the body of a function. When we learned the call stack, we also learned that their storage is allocated automatically once the stack frame of the function is created, and deallocated automatically when the stack frame is popped out of the call stack. That is why we say that the local variables have automatic storage duration, as their memory storage is automatically allocated when the enclosing function is called and deallocated when it returns.

Local variables also have block scope, which means that they are visible from their point of declaration to the end of the enclosing function body. Here visible means that they can be used.

The parameters of a function also have automatic storage duration and block scope. The only difference is that they are always initialized with the value of arguments, while local variables do not have to be assigned initial values when they are declared.