Introduction to Scientific and Engineering Computation

(BIL 102E)

LECTURE 12

Dynamic Memory Allocation

In many cases, the exact sizes of arrays to be used by the program cannot be determined before the compilation of the program. This brings inflexibility to the program if a static array is used.

An alternative is allocating the memory required for a variable (or for an array) dynamically.

The following functions, which are defined in *stdlib* library can be used for this purpose:

1) malloc()
2) free()
3) calloc()
4) realloc()

malloc()

malloc() function allocates a specified size of memory space and returns the beginning address of the newly allocated memory space. For example,

```
int *pk;
pk = malloc(10 * sizeof(int));
```

reserves space in memory that could hold 10 integers. Then, it is possible to use this space by using the pointer pk. For example,

```
*pk = 5; /* Let the first element be 5 */
*(pk+1) = 10; /* Let the second element be 10 */
*(pk+9) = -65; /* Let the ninth element be -65 */
printf(“%d”,*(pk+1)); /* prints 10 on the screen */
```

free()

Every reserved memory space must be relased when it is no more necessary. The function free() can be used for this purpose. For example,

```
free(pk); /* frees the memory spaces pointed by pk */
```

calloc()

This works like malloc() but two arguments are used the of which determines the number of elements in the memory area and the second of which determines the number of bytes required by each element. All elements are initialized to 0 in the beginning. For example,

```
pk = calloc(10,sizeof(int));
```

```
void studswap(student *std1, student *std2){
  student temp;
  temp=*std1;
  *std1=*std2;
  *std2=temp;
}

void studsort2_surname(student *std1, student *std2){
 if( strcmp(std1->individual.surname,std2->individual.surname)>0 )
         studswap(std1,std2);
}
void ReadStudent(student *stud){
  printf("Name : ");
  scanf("%15s",&(stud->individual.name));
  printf("Surname : ");
  scanf("%15s",&(stud->individual.surname));
  printf("Department : ");
  scanf("%15s",&(stud->department));
}

void PrintStudent(student *stud){
  printf("%s\t\t",stud->individual.name);
  printf("%s\t\t",stud->individual.surname);
  printf("%s\n",stud->department);
}
```

```
main()
{
  int i,j, num_studs=0;
  student *studs;
  printf("Enter the number of students : ");
  scanf("%d",&num_studs);
  studs=malloc(num_studs * sizeof(student));
  if(studs == 0) {
    printf("Memory Allocation Error!\n");
        return 1;
  }
  for(i=0;i<num_studs;i++) {
         printf("Information on Student %d:\n",i+1);
         ReadStudent(studs+i);
  }

  for(i=0;i<num_studs-1;i++)
          for(j=i+1; j<num_studs; j++)
                  studsort2_surname(studs+i, studs+j);
  /* Print out the result */
  printf("Name\t\tSurname\t\tDepartment\n");
  for(i=0;i<num_studs;i++)
          PrintStudent(studs+i);
  return 0;
}
```

A possible output:

```
Enter the number of students : 4
Information on Student 1:
Name : Ali
Surname : Pala
Department : Meteo
Information on Student 2:
Name : Ayse
Surname : Akpinar
Department : Electrical
Information on Student 3:
Name : Canan
Surname : Aksin
Department : Computer
Information on Student 4:
Name : Ahmet
Surname : Celesun
Department : Mining
Name            Surname         Department
Ayse            Akpinar         Electrical
Canan           Aksin           Computer
Ahmet           Celesun         Mining
Ali             Pala            Meteo
```

Scope of Variables

All variables defined in functions are called *local variables*. Normally, memory corresponding to these variables are reserved dynamically when the function is called and and is marked as empty when the function terminates. Other functions cannot reach these variables by just using their names.

It is possible to define variables outside functions. Such variables are called *global variables*. The memory corresponding to these variables are reserved when the program starts and marked as empty when the program terminates. All functions defined after the definition of these variables can reach and change them.

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

char *name; /* This is a global variable */

void printName() {
  printf("%s",name);
}

main() {
  char buffer[51];
  printf("What is your name?\n");
  scanf("%50s",buffer);
  name=malloc(strlen(buffer)+1);
  strcpy(name,buffer);
  printf("Your name is “);
  printName();
  free(name);
}
```

The global variable can be used anywhere after its declaration.

```
What is your name?
Mustafa
Your name is Mustafa
```

The *static* specifier

Normally local variables are erased when the function returns to the calling function. If a local variable is defined with the *static* specifier, however, its value is saved.

```
#include <stdio.h>
void counter(void){
  static int k=1;
  int x=1;
  printf("The variable k in counter is %d\n",k);
  printf("The variable x in counter is %d\n",x);
  k++; x++;
}

main() {
  int k=5;
  printf("The variable k in the main program is %d\n",k);
  counter();
  printf("The variable k in the main program is %d\n",k);
  counter();
  printf("The variable k in the main program is %d\n",k);
  counter();
}
```

The value of k is saved after termination of the function. Whereas, the value of x is not saved and initialized to 1 each time the function starts.

Note that the variable k in the main function and the local variable k in the counter function are different.

The output of the program:

```
The variable k in the main program is 5
The variable k in counter is 1
The variable x in counter is 1
The variable k in the main program is 5
The variable k in counter is 2
The variable x in counter is 1
The variable k in the main program is 5
The variable k in counter is 3
The variable x in counter is 1
```

Recursive Functions

If a function calls itself either directly or through a chain of function calls the function is called a recursive function.

It is programmers responsibility to make sure that the recursion (calling itself) does not continue infinitely and there exists a segment of the function where it is possible to exit from the function without recursion.

When planned carefully recursive functions are easy to implement. However, they might slow down the program and can use excessive amount of memory unnecessarily. Therefore, should be avoided when possible.

```
#include <stdio.h>
long int Factorial(int n) {
  if(n<2)
    return 1;
  else
    return n*Factorial(n-1);
}

main(){
  int k;
  for(k=0;k<7;k++)
    printf("%d! = %ld\n", k, Factorial(k));
}
```

Make sure that the recursion does not continue infinitely.

Make a recursive call. Note that n! = n (n-1)!

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

```
#include <stdio.h>
long int Fibonacci(int n) {
  if(n<0)
    return -1;
  else if (n==0)
    return 0;
  else if (n==1)
    return 1;
  else
    return Fibonacci(n-1)+Fibonacci(n-2);
}

main(){
  int k;
  for(k=0;k<10;k++)
    printf("F%d = %2ld\n",k,Fibonacci(k));
}
```

A recursive Fibonacci function.

```
F0 =  0
F1 =  1
F2 =  1
F3 =  2
F4 =  3
F5 =  5
F6 =  8
F7 = 13
F8 = 21
F9 = 34
```