Arrays and Pointers: Miscellaneous

Arrays of pointers: We have encountered an array of pointers in Lecture 21, when we created a 2D array-like variable. Consider the following code segment in which `B` becomes like a 2D integer array of dimensions `3x4`:

```
int i,**B;
B=(int **)calloc(3,sizeof(int *));
for(i=0;i<3;i++)
{
B[i]=(int*)calloc(4,sizeof(int));
}
```

After the first call to the `calloc()`, three contiguous spaces have been created in the memory where each space can hold the address of an integer variable. Thus, these three contiguous locations serve as an array of pointers. Each of these pointers then points to four contiguous places in memory, creating a 2D array-like variable `B`.

Here, we follow similar ideas. However, the array of pointers corresponds to a variable. Additionally, each pointer may point to spaces of different sizes. To explain this, we consider the following program, which reads a sentence, separates its words, and then sorts the words.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define N 100              // Max no. of words
#define MXWORD 50    // max length of a word
void sentn_to_wrd(char **,int *);  // extract words
void sort_wrd(char **,int);        // sort words
int main()
{
int i,n;                           // n actual no of words
char *W[N];     // W array of pointers
sentn_to_wrd(W,&n);                // function to extract words
sort_wrd(W,n);     // function to sort words
printf("Words in sorted order are:\n");
for(i=0;i<n;i++)
    {
    printf("%s\n",W[i]);
    }
 return 0;
}
```

```
void sort_wrd(char **A, int n)
{
int i,j;
char *temp;
// use bubble-sort
for(i=0;i<n-1;i++)
{
 for(j=i+1;j<n;j++)
 {
  if(strcmp(A[i],A[j])>0)   // compare strings
  {
   temp=A[i];
   A[i]=A[j];
   A[j]=temp;
  }
 }
}
 return;
}
void sentn_to_wrd(char **A, int *n)
{
 char c,word[MXWORD];
 int i,k;               // k actual no of words
 printf("Enter a sentence: "); // type a sentence and press return/enter key
 i=0;                          // i count the length of a word
 k=0;
 c=getchar();                  // use getchar() to read a char in c
 while(c!='\n')       // A sentence ended by return/enter key
 {
  if(c !=' ')          // if char is not a blank space, it is part of a word
  {
   word[i]=c;
   i++;
  }
  else                 // if char is blank space
  {
   if(i !=0)    // end of word
   {
    word[i]='\0';      // add '\0' at the end
```

```
    A[k]=(char *)calloc(i+1,sizeof(char));  // create space for word
     strcpy(A[k],word);                      // copy word to created space
    i=0;                                    // look for new word
    k++;
   }
  }
  c=getchar();    // read next char
 }
 if(i !=0)            // end of input not blank space. Here '\n'
 {
  word[i]='\0';
  A[k]=(char *)calloc(i+1,sizeof(char));
   strcpy(A[k],word);
  i=0;
  k++;
 }
 *n=k;           // assign to n of main
 return;
}
```

Listing 58: C Program "arrpntr.c"

```
Enter a sentence: Joy and sorrow sleep in the same bed
Words in sorted order are:
Joy
and
bed.
in
same
sleep
sorrow
the
```

Please note that a few statements are omitted from the code, and those should be inserted for a good code. For example, we have not checked whether the total number of words exceeds `N` or a word-length exceeds `50`. Also, we have not checked whether the `calloc()` returns a non-null pointer, etc.

Arguments to main(): Until now, we have declared `main()` as `int main()` (for example, see program "arrpntr.c" in Listing 58), which declares that `main()` has no argument and it returns an integer. To compile it, we type `gcc arrpntr.c` in the terminal, which produces the default

executable `a.out`. Note that `gcc arrpntr.c`, which we type in the terminal, is an example of a command line that has two arguments `gcc` (name of the command) and `arrpntr.c`. Similarly, to run the code, we type `./a.out`, which is also a command line with a single argument `./a.out` (name of the command). Until now, any input in a program is carried out via C functions like `scanf()` or `getchar()`. But now we will see that it is also possible to pass arguments to `main()` from a command line. Two arguments, called `argc` and `argv`, can be used with `main()` for this purpose. Here, `argc` is an `int` variable that counts the number of command-line arguments. The argument `argv` is an array of pointers to `char` that make up the command line. The element `argv[0]` contains the name of the command, and hence the value of `argc` is at least one. Below is a simple C program that echoes the command-line argument when executed. It also prints the number of command-line arguments and the number of apples.

```
#include <stdio.h>
#include<stdlib.h>
int main(int argc,char *argv[])
{
 int i,n;
 for(i=0;i<argc;i++)
 {
  printf("%s  ",argv[i]);
 }
 printf("\n");
 printf("argc = %d\n",argc);
 n=atoi(argv[2]);
 printf("Number of apples = %d\n",n);
 return 0;
}
```

Listing 59: C Program "cmdarg.c"

Here is the output of the command line `./a.out eat 1 apple a day!`

```
./a.out  eat  1  apple a day!
argc = 6
Number of apples = 1
```

Note that each argument of the command line is stored as a string (array of characters). The `1` of the command line "`./a.out  eat  1  apple a day!`" is stored as a string in `argv[2]`. We may use `atoi()` function (available in `stdlib.h`) to convert a string (character array) to an integer value. The command `./a.out` is stored in `argv[0]`. Further, `eat` is stored in `argv[1]` and so on.

Functions as arguments: A popular method to find a root α of the equation $f(x) = 0$ is the Newton method. Given an initial guess x_0 for the root, we iterate according to

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, 2, \cdots.$$

With $k = 0$, we find x_1, which is hopefully a better guess than the initial guess x_0. Next, we find x_2 with $k = 1$ and so on. If the initial guess is good and the function satisfies certain conditions, then these iterates converge to the root α. If $f(x_{k+1}) \approx 0$ or $x_{k+1} \approx x_k$, then we may take x_{k+1} as the root. In many cases, the Newton method may not converge. Hence, we stop the process if the number of iterations exceeds a threshold. We apply this method to compute a root of $x^2 - 2 = 0$ for which the roots are $\pm\sqrt{2}$. Here, $f(x) = x^2 - 2$ and $f'(x) = 2x$. Below is the code, which also calculates the number of iterations.

```
#include<stdio.h>
#include<math.h>
#define ITRMAX 20
double fnc(double);
double dfnc(double);
void printroot(int,int,double);
int newton(double *,int *,double,double,double);
int main()
{
 double x0,root,epsf=1.e-7,epsn=1.e-5;
 int flag,niter;
 printf("Enter initial guess x0:");
 scanf("%lf",&x0);
 flag=newton(&root,&niter,x0,epsf,epsn);
 printroot(flag,niter,root);
 return 0;
}

 int newton(double *x,int *n,double x0,double epsf,double epsn)
{
 int i=0;
 double x1;  // x0 for x_n and x1 for x_{n+1}
 x1=x0-fnc(x0)/dfnc(x0);
 i++;
 while(i<ITRMAX && fabs(x1-x0)>epsn && fabs(fnc(x1))>epsf)
 {
  x0=x1;
```

```
  x1=x0-fnc(x0)/dfnc(x0);
  i++;
 }
 if(i==ITRMAX)
  return 0;
 *x=x1;
 *n=i;
 return 1;
}

double fnc(double x)
{
 return x*x-2;
}
double dfnc(double x)
{
 return 2*x;
}

void printroot(int flag, int niter, double root)
{

 if(flag==1)
 {
  printf("Newton method converges to root %0.4e in %d iterations\n",root,niter);
 }
 else
 {
  printf("Newton method does not converge in %d iterations\n",ITRMAX);
 }
 return;
}
```

Listing 60: C Program "newtono.c"

Below is a typical input-output for the code. Clearly, the method converges fast to the positive root.

```
Enter initial guess x0:5
Newton method converges to root 1.4142e+00 in 5 iterations
```

Now suppose that we want to find the roots of the two equations $x^2 - 2 = 0$ and $x - \cos x = 0$. The way we have written the code given in Listing 60 can handle only one equation. To handle both equations, we definitely need two functions for the second function and its derivative. Also, we need another function similar to `newton()` to handle the second equation. However, this last function can be avoided when we pass the function as an argument of a function.

Below is a code where we pass functions as arguments of another function. This is a modification of the code given in Listing 60, and it calculates the roots of two equations.

```
#include<stdio.h>
#include<math.h>
#define ITRMAX 20
double fnc1(double);
double dfnc1(double);
double fnc2(double);
double dfnc2(double);
int newton(double f(double), double g(double),double *,int *,double,double,double);
void printroot(int,int,double);
int main()
{
 double x0,root,epsf=1.e-7,epsn=1.e-5;
 int flag,niter;
 printf("===Solvoing x^2-2=0:\n");
 printf("Enter initial guess x0:");
 scanf("%lf",&x0);
 flag=newton(fnc1,dfnc1,&root,&niter,x0,epsf,epsn);
 printroot(flag,niter,root);

 printf("===Solvoing x-cos x=0:\n");
 printf("Enter initial guess x0:");
 scanf("%lf",&x0);
 flag=newton(fnc2,dfnc2,&root,&niter,x0,epsf,epsn);
 printroot(flag,niter,root);

 return 0;
}
int newton(double fnc(double x), double dfnc(double x), double *x,int *n,
double x0,double epsf,double epsn)
{
 int i=0;
```

```
 double x1;  // x0 for x_n and x1 for x_{n+1}
 x1=x0-fnc(x0)/dfnc(x0);
 i++;
 while(i<ITRMAX && fabs(x1-x0)>epsn && fabs(fnc(x1))>epsf)
 {
  x0=x1;
  x1=x0-fnc(x0)/dfnc(x0);
  i++;
 }
 if(i==ITRMAX)
  return 0;
 *x=x1;
 *n=i;
 return 1;
}

double fnc1(double x)
{
 return x*x-2;
}
double dfnc1(double x)
{
 return 2*x;
}

double fnc2(double x)
{
 return x-cos(x);
}
double dfnc2(double x)
{
 return 1+sin(x);
}
void printroot(int flag, int niter, double root)
{

        if(flag==1)
 {
  printf("Newton method converges to root %0.4e in %d iterations\n",root,niter);
```

```
 }
 else
 {
  printf("Newton method does not converge in %d iterations\n",ITRMAX);
 }
 return;
}
```

Listing 61: C Program "newtonm.c"

Below are two sample input-output of the code:

```
===Solvoing x^2-2=0:
Enter initial guess x0:5
Newton method converges to root 1.4142e+00 in 5 iterations
===Solvoing x-cos x=0:
Enter initial guess x0:1
Newton method converges to root 7.3909e-01 in 3 iterations

===Solvoing x^2-2=0:
Enter initial guess x0:1
Newton method converges to root 1.4142e+00 in 4 iterations
===Solvoing x-cos x=0:
Enter initial guess x0:4
Newton method does not converge in 20 iterations
```

Clearly, the Newton method may not converge if the initial guess is not suitable.
Note that the function `newton()` is defined as

```
int newton(double fnc(double x), double dfnc(double x), double *x,int *n,......)
{
......
```

It is interesting to note that we have `double fnc(double x)` and `double *x`, which should have produced an error due to duplicate definitions. However, the `x` in `double fnc(double x)` is ignored by the compiler just like an array dimension in a one-dimensional array. Thus, we could have written

```
int newton(double fnc(double), double dfnc(double), double *x,int *n,......)
{
.......
```

When a function appears in a parameter declaration, as above, the compiler treats it as a pointer. Here is an equivalent header to the function definition

```
int newton(double (*fnc)(double), double (*dfnc)(double), double *x,int *n,.....)
{
.......
```

Note that `double(*fnc)(double)` is read as "`func` is a pointer to a function that takes a single argument of type double and returns a double." The parentheses around `*func` are necessary. If we write `double *fnc(double)`, then it implies that `fnc()` takes a double argument and returns an address of type `double` (which is not the intention). Further, the expression `x1=x0-fnc(x0)/dfnc(x0)` can be kept as it is, or it may be replaced by `x1=x0-(*fnc)(x0)/(*dfnc)(x0)`. Again, the parentheses around `*func` are necessary.

Note that the function `newton()` is called from the `main()` as
`flag=newton(fnc1,dfnc1,&root,&niter,x0,epsf,epsn);`
To see the pointer nature of the function, we can modify the `main()` as follows, where we declare two pointers to functions, and they can be assigned `fnc1()`and `dfnc1()`. Similar operations can be done to any function that is passed as arguments in other functions.

```
........
........
int main()
{
 double x0,root,epsf=1.e-7,epsn=1.e-5;
 int flag,niter;
 double (*fnc)(double),(*dfnc)(double); //fnc & dfnc are two pointers to
                                        //functions that return double
 printf("===Solvoing x^2-2=0:\n");
 printf("Enter initial guess x0:");
 scanf("%lf",&x0);
 fnc=fnc1;    // assign fnc1 to fnc
 dfnc=dfnc1;  // assign dfnc1 to dfnc
 flag=newton(fnc,dfnc,&root,&niter,x0,epsf,epsn); // pass fnc and dfnc
.........
.........
```

Return to Lecture Topics