Com S // Cpr E // Math 5250

Numerical Analysis of High-Performance Computing

Instructor: Songting Luo

Lecture 8: Arrays, Functions, & Simple File I/O in C

Outline

1. `#include<math.h>`
2. Arrays in C
3. Functions in C
4. Simple I/O in C

```
#include<math.h>
```

Math header file

$ISUHPC/lectures/lecture8/codes/somemath.c:

```c
#include <math.h>
#include <stdio.h>

int main()
{
    const double pi = 4.0*atan(1.0);
    double x1 = sqrt(17.3);
    double x2 = pow(4.5,3);
    double x3 = fabs(-45.401);
    double x4 = cos(0.25*pi);

    printf(" pi = %23.16e\n",pi);
    printf(" x1 = %23.16e\n",x1);
    printf(" x2 = %23.16e\n",x2);
    printf(" x3 = %23.16e\n",x3);
    printf(" x4 = %23.16e\n",x4);
    return 0;
}
```

Math header file

```
$ gcc somemath.c

$ ./a.out

 pi =  3.141592653589793e+00
 x1 =  4.159326868617084e+00
 x2 =  9.112500000000000e+01
 x3 =  4.540100000000003e+01
 x4 =  7.071067811865475e-01
```

Arrays in C

A basic array

```
1  // Declaration of an array of integers of length 10
2  int arr[10];
3
4  // To set the first element
5  arr[0] = -5;
6
7  // To set the fourth element
8  arr[3] = 18;
9
10 // To set the last element
11 arr[9] = -1928;
```

A basic array

```
1  // Declaration of an array of integers of length 10
2  int arr[10];
3
4  // To set the first element
5  arr[0] = -5;
6
7  // To set the fourth element
8  arr[3] = 18;
9
10 // To set the last element
11 arr[9] = -1928;
```

Arrays are stored sequentially in memory.

This means that the above array of 10 integers consumes a single 40-byte block of memory (assuming a 32-bit integer).

When you access an array with `arr[N]`, the computer knows where `arr[0]` is stored and it knows that `N*sizeof(int)` is the offset.

A basic array

```
1  // Declaration of an array of integers of length 10
2  int arr[10];
3
4  // To set the first element
5  arr[0] = -5;
6
7  // To set the fourth element
8  arr[3] = 18;
9
10 // To set the last element
11 arr[9] = -1928;
```

Arrays are stored sequentially in memory.

This means that the above array of 10 integers consumes a single 40-byte block of memory (assuming a 32-bit integer).

When you access an array with `arr[N]`, the computer knows where `arr[0]` is stored and it knows that `N*sizeof(int)` is the offset.

Warning: If you try to access `arr[100]`, the compiler lets you; all it knows are offsets. However, the program may crash at runtime. This is called an overflow.

A simple example

$ISUHPC/lectures/lecture8/codes/array1.c:

```
1  #include <stdio.h>
2  int main()
3  {
4      int factorial[10];
5      int i;
6      // compute some factorials
7      factorial[0] = 1;
8      for (i=1; i<10; i++)
9      {
10         factorial[i] = factorial[i-1]*i;
11     }
12     // output each array element's value
13     for (i=0; i<10; i++)
14     {
15         printf(" factorial[%i] = %i\n",i,factorial[i]);
16     }
17     return 0;
18 }
```

A simple example

```
$ gcc array1.c
$ ./a.out
 factorial[0] = 1
 factorial[1] = 1
 factorial[2] = 2
 factorial[3] = 6
 factorial[4] = 24
 factorial[5] = 120
 factorial[6] = 720
 factorial[7] = 5040
 factorial[8] = 40320
 factorial[9] = 362880
```

A simple example

```
$ gcc array1.c
$ ./a.out
 factorial[0] = 1
 factorial[1] = 1
 factorial[2] = 2
 factorial[3] = 6
 factorial[4] = 24
 factorial[5] = 120
 factorial[6] = 720
 factorial[7] = 5040
 factorial[8] = 40320
 factorial[9] = 362880
```

Direct iniatialization:

```
1 // An array of double of length 5
2 double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Multidimensional arrays

```
1 // Declaration of an 2D array
2 int arr[10][10];
3
4 // To set some elements
5 arr[0][0] = -5;
6 arr[3][4] = 18;
7 arr[9][9] = -1928;
```

The implementation of the array stores all the elements in a single contiguous block of memory.

In memory, the array is arranged with the elements of the rightmost index next to each other. In other words, `arr[1][8]` comes right before `arr[1][9]` in memory.

It's efficient to access memory which is near other recently accessed memory. This means that the most efficient way to read through a chunk of the array is to vary the rightmost index the most frequently since that will access elements that are near each other in memory.

Multidimensional arrays

Consider the following matrix with 2 rows and 3 columns:

$$A = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}$$

In C this 2D array could be declared and iniatialized as follows:

```
1  int A[2][3] ={{10, 20, 30},{40, 50, 60}};
```

Suppose the array storage starts at memory location 3401.

This is how the 2D array would be layed out in memory:

```
loc 3401      A[0][0] = 10
loc 3402      A[0][1] = 20
loc 3403      A[0][2] = 30
loc 3404      A[1][0] = 40
loc 3405      A[1][1] = 50
loc 3406      A[1][2] = 60
```

Multidimensional arrays

This means that if we want to loop through the array, e.g., to print the entries, the most efficient approach would be the following:

```
1 for (int i=0; i<2; i++)
2     for (int j=0; j<3; j++)
3     {
4         printf(" A[%i][%i] = %i\n",i,j,A[i][j]);
5     }
```

Which would yield:

```
A[0][0] = 10
A[0][1] = 20
A[0][2] = 30
A[1][0] = 40
A[1][1] = 50
A[1][2] = 60
```

Functions in C

Pascal's Triangle

$ISUHPC/lectures/lecture8/codes/pascal.c:

```
1  int main()
2  {
3      // function declaration
4      int binomial(int n,int k);
5
6      // Pascal's triangle
7      printf("\n Pascal's Triangle:\n\n");
8      for (int n=0; n<=10; n++)
9          for (int k=0; k<=n; k++)
10         {
11             int ans = binomial(n,k);
12             printf("%5i",ans);
13             if (k==n)
14             { printf("\n"); }
15         }
16     printf("\n");
17     return 0;
18 }
```

Pascal's Triangle

$ISUHPC/lectures/lecture8/codes/pascal.c:

```
1 // binomial coefficient (i.e., n choose k)
2 int binomial(int n,int k)
3 {
4     // function declaration
5     int factorial(int n);
6
7     // return n choose k
8     return factorial(n) / (factorial(k) * factorial(n -
          k));
9 }
```

Pascal's Triangle

$ISUHPC/lectures/lecture8/codes/pascal.c:

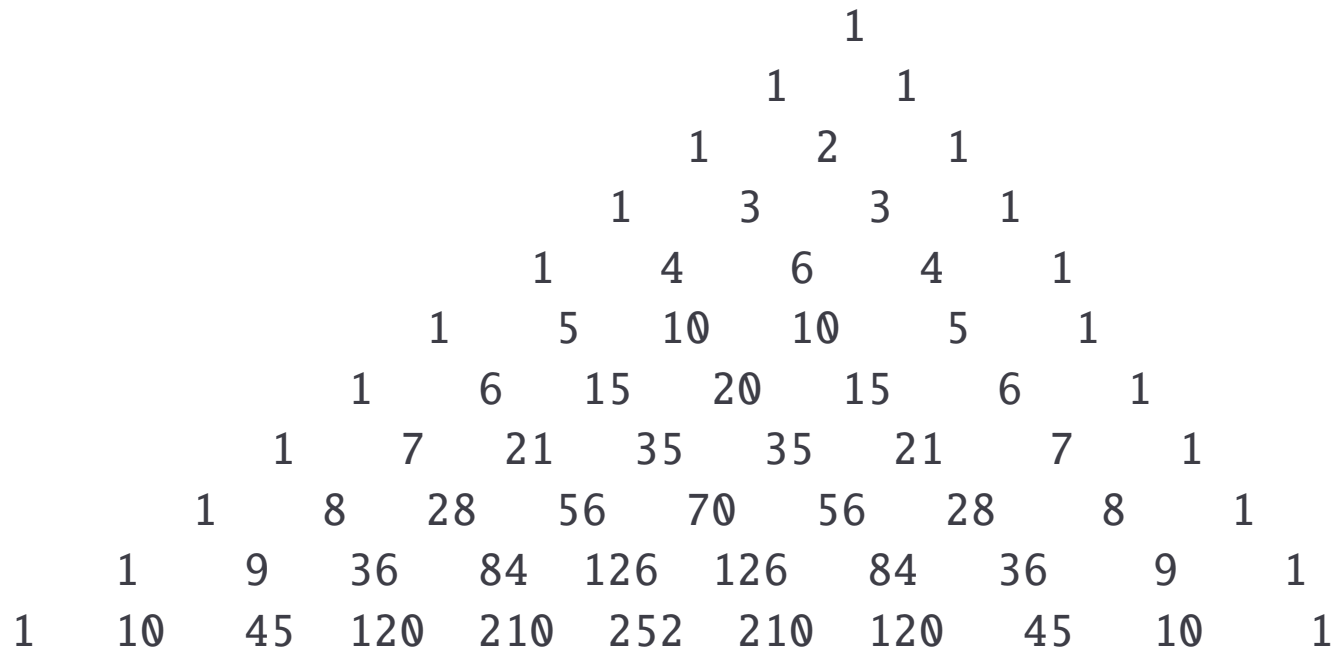
```
1  // recursive factorial function
2  int factorial(int n)
3  {
4     if (n<=1)
5     {
6        // base case
7        return 1;
8     }
9     else
10    {
11       // return n*(n-1)!
12       return n*factorial(n - 1);
13    }
14 }
```

Simple file I/O

```
$ gcc pascal.c

$ ./a.out

Pascal's Triangle:

    1
    1     1
    1     2     1
    1     3     3     1
    1     4     6     4     1
    1     5    10    10     5     1
    1     6    15    20    15     6     1
    1     7    21    35    35    21     7     1
    1     8    28    56    70    56    28     8     1
    1     9    36    84   126   126    84    36     9     1
    1    10    45   120   210   252   210   120    45    10     1
```

Simple file I/O

Pascal's Triangle:



Simple I/O in C

Simple file I/O

$ISUHPC/lectures/lecture8/codes/WriteRead.c:

```
1  int main()
2  {
3      // some data
4      double A[2][3] ={{10,-20,30},{-40,50,-60}};
5
6      // create an output file
7      FILE* outfile = fopen("out1.data","w");
8
9      // output data
10     for (int i=0; i<2; i++)
11         for (int j=0; j<3; j++)
12         {
13             fprintf(outfile,"%23.16e\n",A[i][j]);
14         }
15
16     // close output file
17     fclose(outfile);
```

Simple file I/O

$ISUHPC/lectures/lecture8/codes/WriteRead.c:

```
    FILE* infile = fopen("out1.data","r");

    // input data
    double B[2][3];
    for (int i=0; i<2; i++)
       for (int j=0; j<3; j++)
       {
          double z;
          fscanf(infile,"%lf",&z);
          B[i][j] = z;
          printf(" B[%i][%i] = %8.4f\n",i,j,B[i][j]);
       }

    // close input file
    fclose(infile);

    return 0;
}
```

Simple file I/O

```
$ gcc WriteRead.c

$ ./a.out

B[0][0] =  10.0000
B[0][1] = -20.0000
B[0][2] =  30.0000
B[1][0] = -40.0000
B[1][1] =  50.0000
B[1][2] = -60.0000

$ more out1.data

 1.00000000000000e+01
-2.00000000000000e+01
 3.00000000000000e+01
-4.00000000000000e+01
 5.00000000000000e+01
-6.00000000000000e+01
```

Simple user prompt

$ISUHPC/lectures/lecture8/codes/simpleio.c:

```c
#include <stdio.h>

int main()
{
    int a;

    printf("Please input an integer value: ");
    scanf("%i", &a);
    printf("You entered: %i\n", a);

    return 0;
}
```

```
$ gcc simpleio.c
$ ./a.out
Please input an integer value: 9
You entered: 9
```

Lab assignment

1. Develop source codes ("main program") with name **lab8.c**: define two functions (1) computing factorial of an integer n and (2) computing exponential of a real number x.
   - For computing $factorial(n)$, define a function with recursive calls.
   - For computing $\exp(x)$, define a function with Taylor expansions (recall Python codes), where, the constant $e = 2.718281828459$, $x_0 = round(x)$ is the closest integer to x, and use the $pow(\cdot, \cdot)$ from *math.h* for computing powers. Use your own factorial function.
   - Generate a set of points $x = 0, 0.02, 0.04, \ldots, 0.98, 1$, compute $\exp(x)$ at the given points (with your own function).
   - Output the data to a data file, and visualize the $\exp(x)$ function with the data in Python.
2. Update Git repository.
3. Submit codes, output data file and screen shots.