EE204 Scientific Programming for EE
C Review

Mehmet Çalı

Table of Contents

Variable Declaration

- Variable classification
  - Local variables
  - Global variables
  - ➢ Static variables

```
1   int a1;
2   static int a2;

4   void myFunction(){
5       int b1;
6       static int b2;
7   }

9   int main(){
10      int c1;
11      static int c2;
12      return 0;
13  }
```

global

local

local

Global local variable declaration

```
3   int f1(){
4       int a=0;
5       return a++;
6   }
7
8   int f2(){
9       static int a=0;
10      return a++;
11  }
12
13  void main(){
14      printf("%d\t",f1());
15      printf("%d\t",f1());
16      printf("%d\t",f2());
17      printf("%d\t",f2());
18      // Output: 0 0 0 1
19  }
```

Static and non-static variable declaration

Variable Declaration

- Scope of variables:

```
 3  int a=0;
 4  void f1(){
 5      {
 6          int b=1;
 7          printf("%d\t%d\n",a,b); //Out: 0 1
 8      }
 9      int c=2;
10      printf("%d\t%d\n",a,c);
11      {
12          printf("%d\t%d\n",a,c); //Out: 0 2
13      }
14  }
15
16  void main(){
17      int d=3;
18      f1();
19      printf("%d\t%d",a,d); //Out: 0 3
20  }
```

Scope of b

Scope of c

Scope of d

Scope of a

Variable Declaration

- Declaration Location:
  - Depends on the C standard (C89,C90,C99,C11 etc.)
  - Need to be at the beginning of blocks in C89 (ANSI C)

C89:

```
4  void main(){
5      int a=1;
6      printf("%d\n",a);
7      int b=2;
8      printf("%d\n",b);
9  }
```

C99 and succeeding standards:

```
4  void main(){
5      int a=1;
6      printf("%d\t",a);
7      int b=2;
8      printf("%d\n",b);
9  }   //Out: 1 2
```

- For loop initial declarations:

C89,C90:

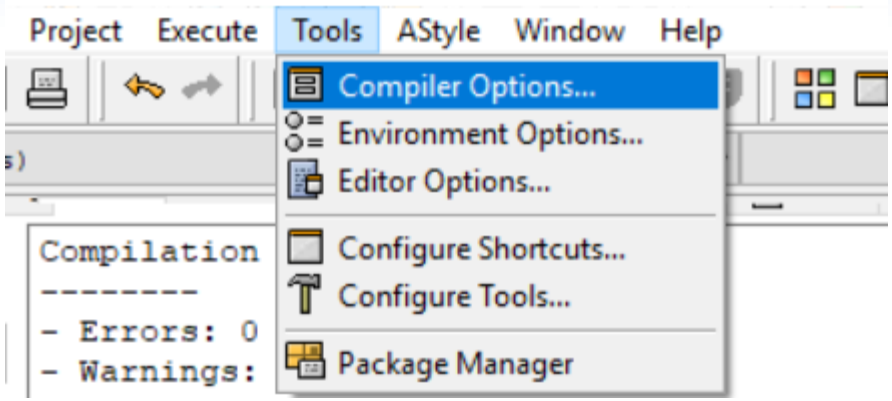
```
4  void main(){
5      for(int i=0;i<2;i++){
6          printf("%d\t",i);
7      }
8  }
```

C99 and succeeding standards:

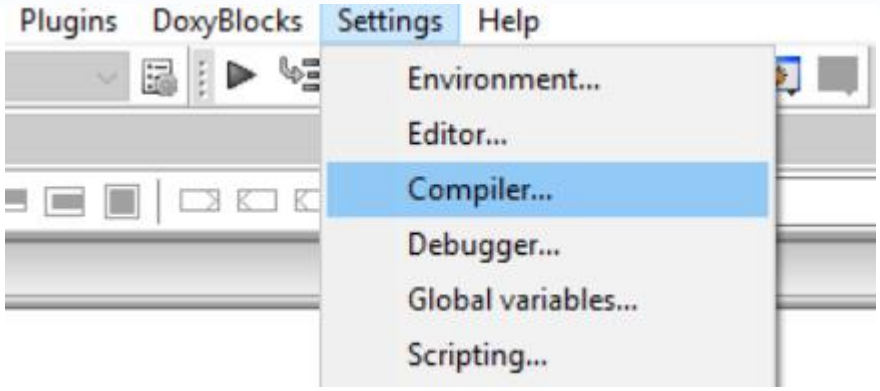
```
4  void main(){
5      for(int i=0;i<2;i++){
6          printf("%d\t",i);
7      }
8  } //Outputs 0 1
```

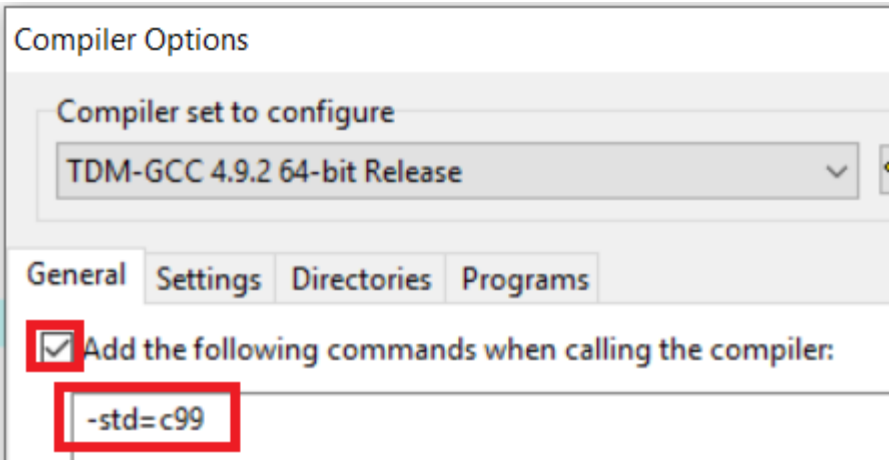
Compiler Options

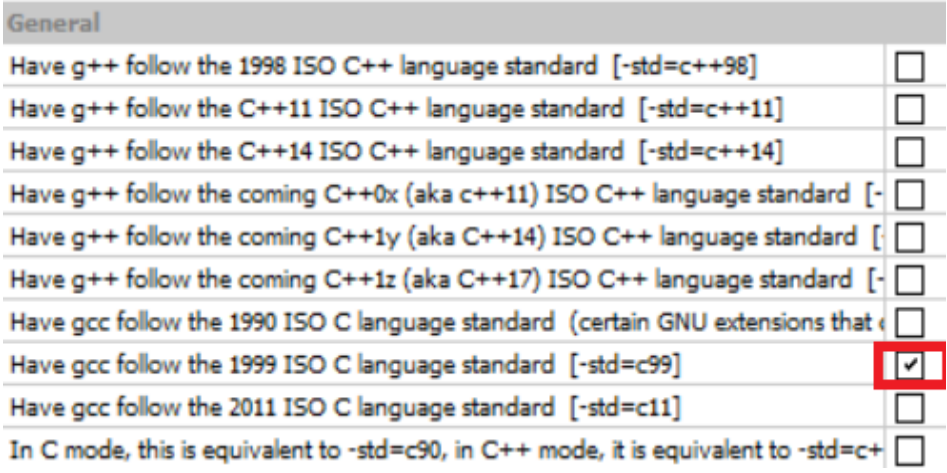
- Dev-C++



- Codeblocks

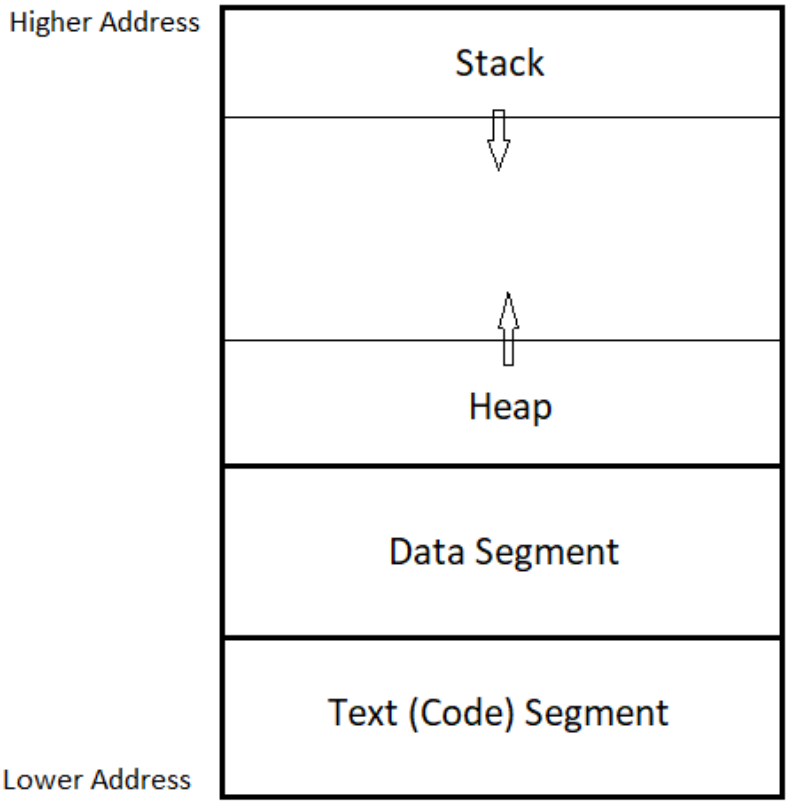






Memory Layout of C

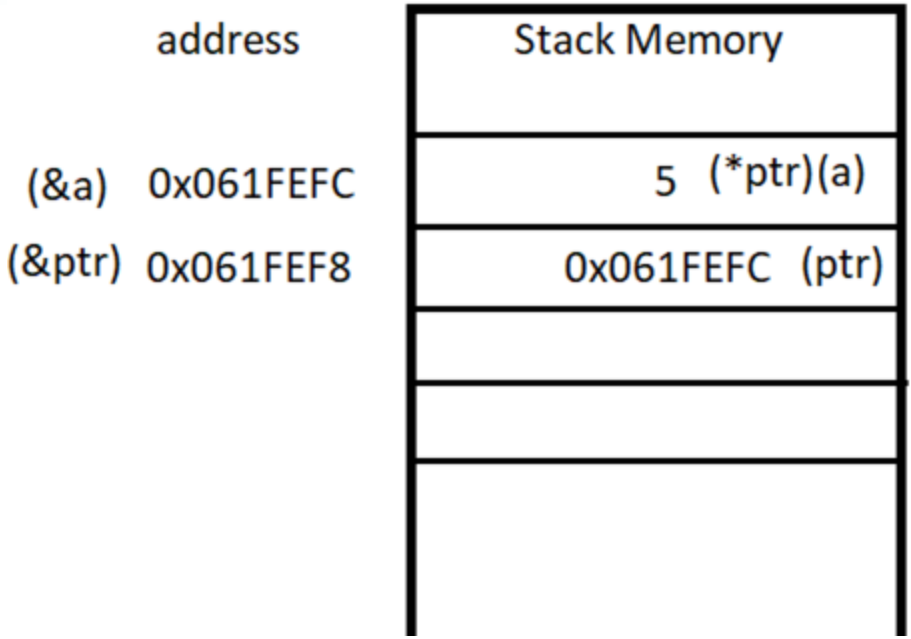
- In a C program memory is divided into 4 segments:
  - Stack
  - Heap
  - Data
  - Text (or Code)

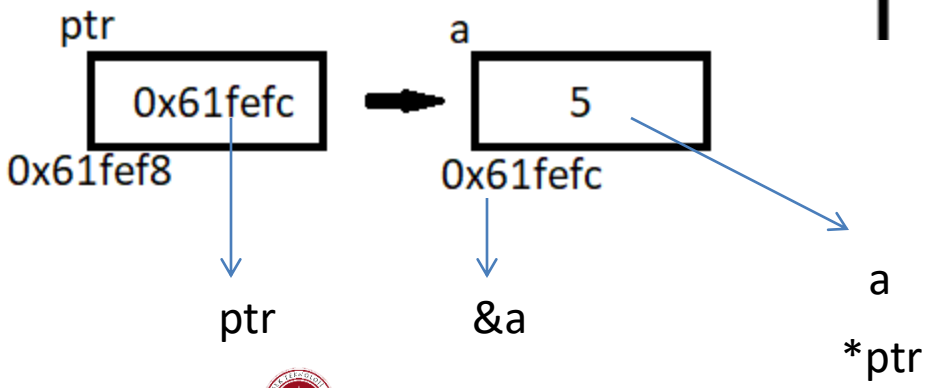


- Non-static local variables — Stack
- Global and static variables — Data
- Dynamically allocated memory using malloc(), calloc() etc. — Heap

Pointers

```
4  void main(){
5      int a=5;
6      int *ptr=&a;
7      printf("%p\n",&a); //0061fefc
8      printf("%d\n",a); //5
9      printf("%p\n",ptr); //0061fefc
10     printf("%d\n",*ptr); //5
```

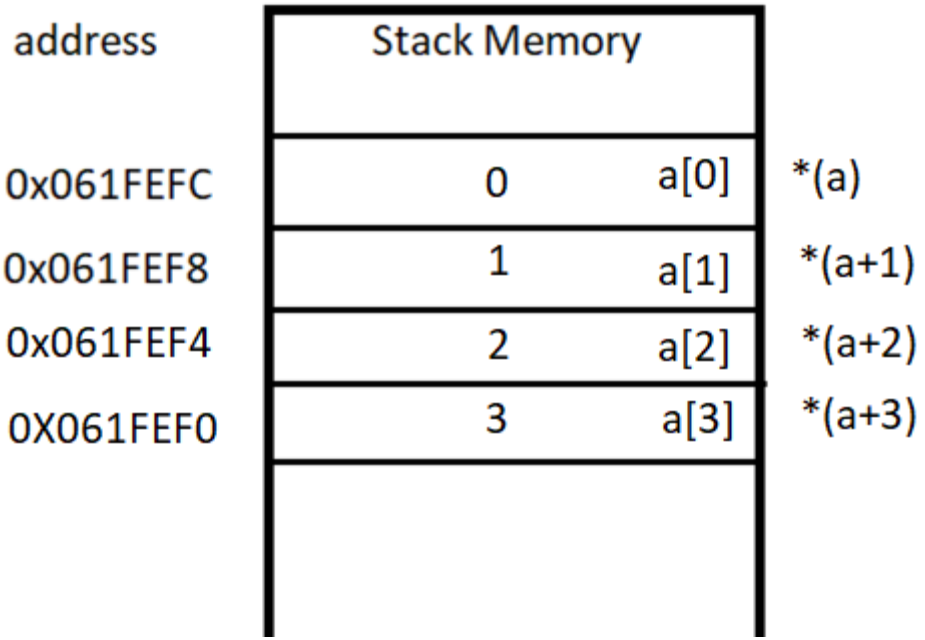




Arrays and Pointers

- The name of the array can be used as a pointer that points the first element of the array

```
4  void main(){
5      int a[3];
6      for(int i=0;i<3;i++){
7          *(a+i)=i;
8          //equivalent to a[i]=i;
9          printf("%d\t",a[i]);
10     }
11 }
```



Case Study: Functions with multiple outputs

- Assume we want to write a function that outputs square root, square and cube of a number. Since we cannot return multiple variables or an array in C we need to think another way:
- **Wrong Attempt :**

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

double * multi_output(double a){
    double out[3];
    out[0]=sqrt(a);
    out[1]=a*a;
    out[2]=a*a*a;
    return out;
}

int main()
{
    double *x=multi_output(3);
    printf("%f\t%f\t%f\n",x[0],x[1],x[2]);
    return 0;
}
```

Run time error!

Local variables are unavailable outside the scope

Case Study: Functions with multiple outputs

- **Method 1:** Using static keyword correct outputs can be achieved.

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

double * multi_output(double a){
    static double out[3];
    out[0]=sqrt(a);
    out[1]=a*a;
    out[2]=a*a*a;
    return out;
}

int main()
{
    double *x=multi_output(3);
    printf("%f\t%f\t%f\n",x[0],x[1],x[2]);
    return 0;
}

```

Output: 1.732051 9.0000 27.0000

Warning: Not memory efficient!

Case Study: Functions with multiple outputs

- **Method 2: Using pass by reference**

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   void multi_output(double a, double *x, double *y, double *z){
6       *x=sqrt(a);
7       *y=a*a;
8       *z=a*a*a;
9   }
10
11  int main()
12  {
13      double x, y, z;
14      multi_output(3,&x,&y,&z);
15      printf("%f\t%f\t%f\n",x,y,z);
16      return 0;
17  }
```

Output: 1.732051 9.0000 27.0000

- Better than previous method but what happens when we have lots of outputs?

Case Study: Functions with multiple outputs

- **Method 3:** We can switch multiple pointer type parameters with an array

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

void multi_output(double a, double x[]){
    x[0]=sqrt(a);
    x[1]=a*a;
    x[2]=a*a*a;
}

int main()
{
    double x[3];
    multi_output(3,x);
    printf("%f\t%f\t%f\n",x[0],x[1],x[2]);
    return 0;
}
```

- Brackets are important to indicate this is an array.
- We can either put the size or leave it empty
- It is same as `double *x`

- Do not put any brackets while calling

- When passing 2D arrays:

```c
void multi_output(double a, double x[3][2]){
    //...
}
int main()
{
    double x[3][2];
    multi_output(3,x);
```

Case Study: Arrays Operations

- Assume that we want to perform some vector operations in which the size of the vectors should be defined by the user.

Method-1: Variable Length Arrays (VLA) (do not use in this course):

```
int main()
{
    int i,size;
    scanf("%d",&size);
    int a1[size];
    for(i=0;i<size;i++){
        int temp;
        scanf("%d",&temp);
        a1[i]=temp;
    }
```

First gets the size from user

Declares the array using the size

Assign its elements in a for loop

Drawbacks:

- VLAs are inside the C standard only for C99
- After C11 (next C standard) it becomes optional (compilers does not necessarily supports it)
- We are not able to deallocate the memory
- Only valid inside local scopes

```
2  #include <stdlib.h>
3
4  int size2=5;
5  int a2[size2];
6
```

Case Study: Array Operations

```c
#include <stdio.h>
#include <stdlib.h>
#define N 100
int main()
{
    int a[N],b[N],sum[N];
    int size;
    scanf("%d",&size);
    for(int i=0;i<size;i++){
        scanf("%d",&a[i]);
    }

    for(int i=0;i<size;i++){
        scanf("%d",&b[i]);
    }

    //summation
    for(int i=0;i<size;i++){
        sum[i]=a[i]+b[i];
    }
```

Method-2 (not memory efficient):

- Define a limit to the vector size such as 100 (this value will be size of the array not the actual vector)
- Define another variable to keep the size of the vector
- Use this vector size in any operation related to these arrays

Drawbacks:

- Most of the memory is lost in most of the cases
- We need to use additional mechanicsms to check whether we exceed the array size or not

Third method is dynamic memory allocation but we will go back to that after we go over dynamic memory allocation.

Matrix Creation

- Using 2D array:

```
6   int a[3][2];
7   for(int i=0;i<3;i++){
8       for(int j=0;j<2;j++){
9           a[i][j]=2*i+j;
10      }
11  }
12  for(int i=0;i<3;i++){
13      for(int j=0;j<2;j++){
14          printf("%d\t",a[i][j]);
15      }
16      printf("\n");
17  }
```

Output:

```
0       1
2       3
4       5
```

- Using 1D array:

```
3   #define a(i,j) *(a+i*2+j)
4   void main(){
5       int a[6];
6       for(int i=0;i<3;i++){
7           for(int j=0;j<2;j++){
8               a(i,j)=2*i+j;
9           }
10      }
11      for(int i=0;i<3;i++){
12          for(int j=0;j<2;j++){
13              printf("%d\t",a(i,j));
14          }
15          printf("\n");
16      }
```

Output:

```
0       1
2       3
4       5
```

Pointers to Function

- Example: We have multiple polynomial functions and we need to check for a given input if the output of polynomial is bigger than of input or not.
- We can achieve that by passing address of polynomial functions as parameter.

```
4  double poly1(double x){
5      return x*x+2*x+2;
6  }
7  double poly2(double x){
8      return x*x+3*x;
9  }
10 int check_poly(double x,
11                double (*f)(double)){
12     if(f(x)>x)
13         return 1;
14     else
15         return 0;
16 }
17 void main(){
18     double x=0;
19     printf("%d\t",check_poly(x,poly1));
20     printf("%d",check_poly(x,poly2));
21 }
```

Output: 1 0

Dynamic Memory Allocation

- malloc(): allocates memory from the heap
- calloc(): allocates memory and initialize the content as zero
- free(): deallocates the memory so that it can be used from other programs

```
2   #include <stdlib.h>
3
4   void main(){
5       int *a=malloc(sizeof(int)*4);
6       for(int i=0;i<3;i++){
7           a[i]=i;
8           printf("%d\t",a[i]);
9       }
10      free(a);
11  }
```

- Do not forget to free the memory that you allocated when you do not need it any more!

Case Study: Array Operations

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   int main()
4   {
5       int size;
6       int *a, *b, *sum;
7       scanf("%d",&size);
8       a = malloc(sizeof(int)*size);
9       b = malloc(sizeof(int)*size);
10      sum = malloc(sizeof(int)*size);
11      for(int i=0;i<size;i++){
12          scanf("%d",&a[i]);
13      }
14      for(int i=0;i<size;i++){
15          scanf("%d",&b[i]);
16      }
17      //summation
18      for(int i=0;i<size;i++){
19          sum[i]=a[i]+b[i];
20      }
```

Method-3 Dynamic Allocation:

- Pointers are defined to hold address of the first element which is allocated dynamically.
- Dynamic allocations are made.
- Use the vector size which is also the allocated array size in any operation related to these arrays

Note: Variables a,b and sum are deallocated at the end of the operations

Case Study: Functions with multiple outputs

- **Method 2: Using dynamic memory allocation inside the function**

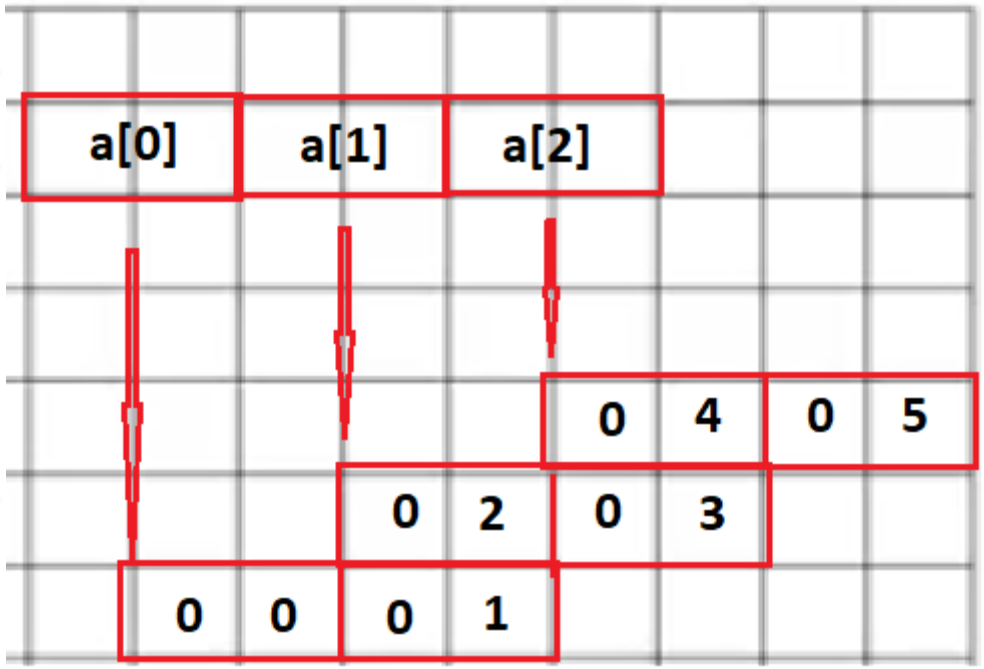
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double * multi_output(double a){
    double *out=malloc(sizeof(double)*3);
    out[0]=sqrt(a);
    out[1]=a*a;
    out[2]=a*a*a;
    return out;
}

int main()
{
    double *x=multi_output(3);
    printf("%f\t%f\t%f\n",x[0],x[1],x[2]);
    free(x);
    return 0;
}
```

Caller should not forget to deallocate

Dynamic Memory Allocation for Matrix

- Using 2D arrays:

```
2   #include <stdlib.h>
3
4   void main(){
5       int **a=malloc(sizeof(int*)*3);
6       for(int i=0;i<3;i++){
7           a[i]=malloc(sizeof(int)*2);
8           for(int j=0;j<2;j++){
9               a[i][j]=i*2+j;
10          }
11      }
12      for(int i=0;i<3;i++){
13          for(int j=0;j<2;j++){
14              printf("%d\t",a[i][j]);
15          }
16          printf("\n");
17      }
18      for(int i=0;i<3;i++){
19          free(a[i]);
20      }
21      free(a);
22  }
```



Dynamic Memory Allocation for Matrix

- Using 1D array:

```
 2  #include <stdlib.h>
 3
 4  #define a(i,j) *(a+i*2+j)
 5  void main(){
 6      int *a=malloc(sizeof(int)*6);
 7      for(int i=0;i<3;i++){
 8          for(int j=0;j<2;j++){
 9              a(i,j)=i*2+j;
10          }
11      }
12      for(int i=0;i<3;i++){
13          for(int j=0;j<2;j++){
14              printf("%d\t",a(i,j));
15          }
16          printf("\n");
17      }
18      free(a);
19  }
```

Structures

```
 4  typedef struct
 5  {
 6      int x;
 7      int y;
 8  } vector;
 9  int scalar_product(vector v1,vector v2){
10      return v1.x*v2.x+v1.y+v2.y;
11  }
12  vector vector_sum(vector v1,vector v2){
13      vector v3;
14      v3.x=v1.x+v2.x;
15      v3.y=v1.y+v2.y;
16      return v3;
17  }
18  void main(){
19      vector v1,v2,v3;
20      v1.x=1;
21      v1.y=2;
22      v2.x=3;
23      v2.y=2;
24      printf("%d\n",scalar_product(v1,v2));
25      v3=vector_sum(v1,v2);
26      printf("%d\t%d",v3.x,v3.y);
27  }
```

Output: 7
4 4

Structures

```
4   struct vector
5   {
6       int x;
7       int y;
8   };
9   int scalar_product(struct vector v1, struct vector v2){
10      return v1.x*v2.x+v1.y+v2.y;
11  }
12  struct vector vector_sum(struct vector v1,struct vector v2){
13      struct vector v3;
14      v3.x=v1.x+v2.x;
15      v3.y=v1.y+v2.y;
16      return v3;
17  }
18  void main(){
19      struct vector v1,v2,v3;
20      v1.x=1;
21      v1.y=2;
22      v2.x=3;
23      v2.y=2;
24      printf("%d\n",scalar_product(v1,v2));
25      v3=vector_sum(v1,v2);
26      printf("%d\t%d",v3.x,v3.y);
27  }
```

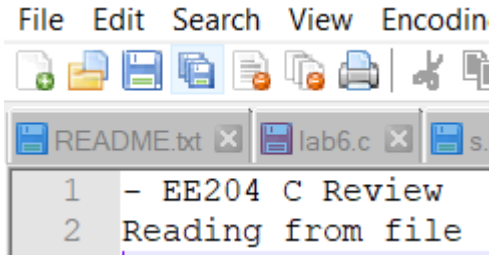
Output: 7
4 4

File Input, Output

- Reading from file:

```c
#include <stdio.h>

void main(){
    char c;
    FILE *f=fopen("D:\\EE204\\2020\\lab2\\mytext.txt","r");
    //FILE *f=fopen("mytext.txt","r"); //if the file is in the same directory
    while(1){
        c=fgetc(f);
        if(c==EOF)
            break;
        printf("%c",c);
    }
    fclose(f);
}
```

- File:



- Output:

```
- EE204 C Review
Reading from file
--------------------------------
```

File Input, Output

- Writing to a file:

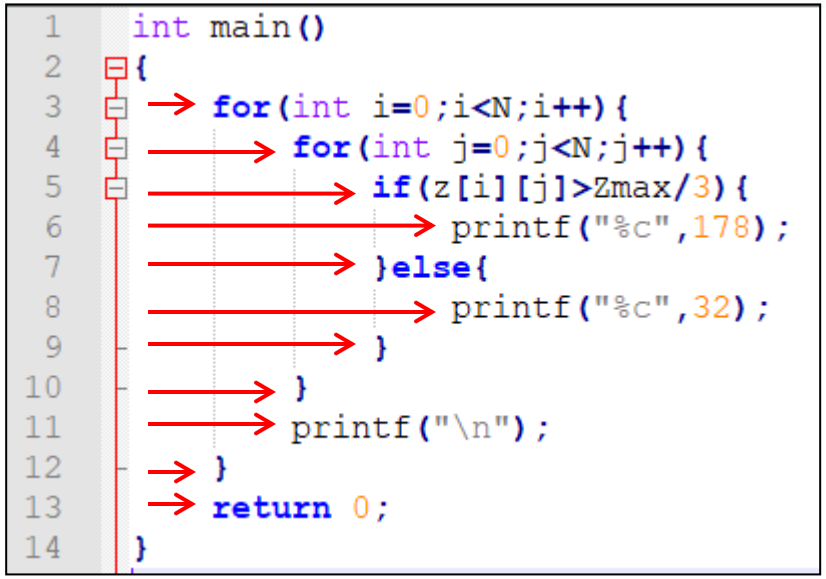
```
2   #include <stdio.h>
3
4   void main(){
5       char c='a';
6       FILE *f=fopen("D:\\EE204\\2020\\lab2\\mytext.txt","w");
7       //FILE *f=fopen("mytext.txt","r"); //if the file is in the same directory
8       while(1){
9           fputc(c,f);
10          c++;
11          if(c=='z')
12              break;
13      }
14      fputc('\n',f);
15      fputs("EE204 Write string to a file",f);
16      fclose(f);
17  }
```

- File:

```
1  abcdefghijklmnopqrstuvwxy
2  EE204 Write string to a file
```

Good Programming Practices

- **Indentation** compliance



```
1  int main()
2  {
3      for(int i=0;i<N;i++){
4          for(int j=0;j<N;j++){
5              if(z[i][j]>Zmax/3){
6                  printf("%c",178);
7              }else{
8                  printf("%c",32);
9              }
10         }
11         printf("\n");
12     }
13     return 0;
14 }
```

- Macro naming: `#define ALLUPPERCASE 5`
- Do not use goto unless it is absolutely necessary
- Do not use global variables unless it is absolutely necessary