CS1100 – Introduction to Programming

Instructor:

Shweta Agrawal (shweta.a@cse.iitm.ac.in)

Lecture 21

Macros in C

- A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive. Example # define c 299792458 (speed of light)

Macros in C

- A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive. Example # define c 299792458 (speed of light)

```
#include <stdio.h>
#define PI 3.1415

int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%f", &radius);

    // Notice, the use of PI
    area = PI*radius*radius;

    printf("Area=%.2f",area);
    return 0;
}
```

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: `enum boolean {No, Yes};` defines two constants No = 0, and Yes = 1.

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: enum boolean {No, Yes}; defines two constants No = 0, and Yes = 1.
- enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: enum boolean {No, Yes}; defines two constants No = 0, and Yes = 1.
- enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};
- When a value is explicitly specified (jan=1) then it starts counting from there

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: enum boolean {No, Yes}; defines two constants No = 0, and Yes = 1.
- enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};
- When a value is explicitly specified (jan=1) then it starts counting from there
- Values start from 0 unless specified otherwise.

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: `enum boolean {No, Yes};` defines two constants No = 0, and Yes = 1.
- `enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};`
- When a value is explicitly specified (jan=1) then it starts counting from there
- Values start from 0 unless specified otherwise.
- Not all values need to be specified. If some values are not specified, they are obtained by increments from the last specified value.

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: `enum boolean {No, Yes};` defines two constants No = 0, and Yes = 1.
- `enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};`
- When a value is explicitly specified (jan=1) then it starts counting from there
- Values start from 0 unless specified otherwise.
- Not all values need to be specified. If some values are not specified, they are obtained by increments from the last specified value.
- Better than #define, as the constant values are generated for us.

Enumerated Constants

```
#include <stdio.h>

enum week {Sun, Mon, Tue, Wed, Thur, Fri, Sat};

int main()
{
    // creating today variable of enum week type
    enum week today;
    today = Wed;
    printf("Day %d",today+1);
    return 0;
}
```

Output is: Day 4.

- Note that the variable values are treated as integers though they look like strings!
- In the program, can use *Wed* > 0 etc. Wed will be treated as an (unisgned) integer.

Declaring Constants

- The qualifier const applied to a declaration specifies that the value will not be changed.

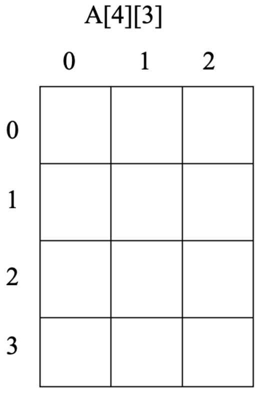
Declaring Constants

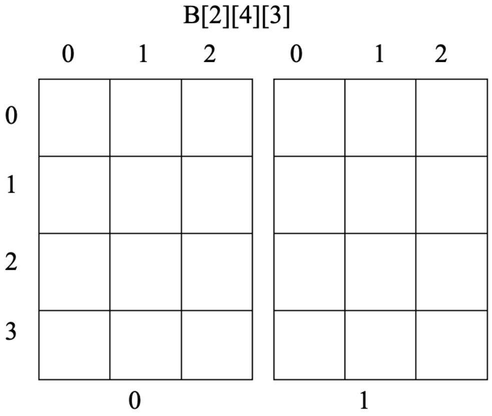
- The qualifier const applied to a declaration specifies that the value will not be changed.
- If I declare const int J = 25; , this means that J is a constant throughout the program.

Declaring Constants

- The qualifier const applied to a declaration specifies that the value will not be changed.
- If I declare const int J = 25; , this means that J is a constant throughout the program.
- Response to modifying J depends on the system. Typically, a warning message is issued while compilation.

Multi-Dimensional Arrays





Storage and Initialization are row by row

Multi-Dimensional Arrays

- double array3d[100][50][75];

Multi-Dimensional Arrays

- double array3d[100][50][75];
- double array4d[60][100][50][75];
  Requires 60*100*50*75*8 = 171.66 MB!

Multi-Dimensional Arrays

- double array3d[100][50][75];
- double array4d[60][100][50][75];
  Requires 60*100*50*75*8 = 171.66 MB!
- Find out how many dimensions your system/compiler can handle.

Initializing 2D Arrays

- int a[3][2] $= \{\{1,4\},\{5,2\},\{6,5\}\}$;
  Recommended that each value is initialized explicitly.

Initializing 2D Arrays

- int a[3][2] $= \{\{1, 4\}, \{5, 2\}, \{6, 5\}\}$;
  Recommended that each value is initialized explicitly.
- int a[3][2] $= \{1, 4, 5, 2, 6, 5\}$;
  Stored in row major order (better not to assume).

Initializing 2D Arrays

- int a[3][2] $= \{\{1,4\},\{5,2\},\{6,5\}\}$;
  Recommended that each value is initialized explicitly.
- int a[3][2] $= \{1,4,5,2,6,5\}$;
  Stored in row major order (better not to assume).
- int a[3][2] $= \{\{1\},\{5,2\},\{6\}\}$; Some elements are not initialized explicitly – they are initialized to 0.

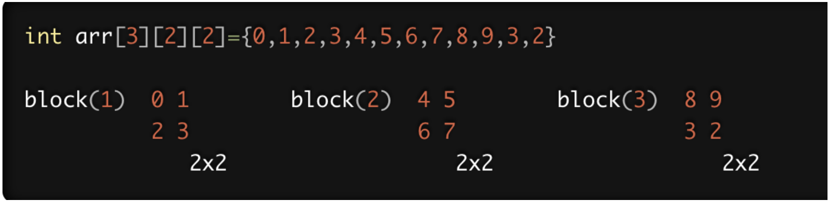
Initializing 2D Arrays

- int a[3][2] $= \{\{1,4\},\{5,2\},\{6,5\}\}$;
  Recommended that each value is initialized explicitly.
- int a[3][2] $= \{1,4,5,2,6,5\}$;
  Stored in row major order (better not to assume).
- int a[3][2] $= \{\{1\},\{5,2\},\{6\}\}$; Some elements are not initialized explicitly – they are initialized to 0.
- a[0][1] $= 0; a[2][1] = 0$;

Initializing 2D Arrays

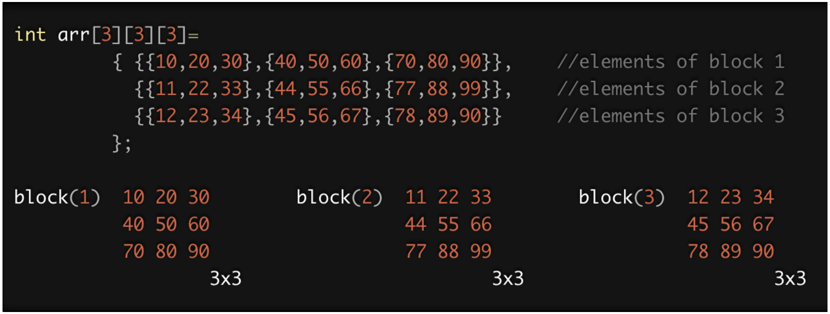
- int a[3][2] $= \{\{1,4\},\{5,2\},\{6,5\}\}$;
  Recommended that each value is initialized explicitly.
- int a[3][2] $= \{1,4,5,2,6,5\}$;
  Stored in row major order (better not to assume).
- int a[3][2] $= \{\{1\},\{5,2\},\{6\}\}$; Some elements are not initialized explicitly – they are initialized to 0.
- a[0][1] $= 0; a[2][1] = 0$;
- Better not to assume!

Initializing 3D Arrays: Block by Block!



```
int arr[3][2][2]={0,1,2,3,4,5,6,7,8,9,3,2}

block(1)  0 1          block(2)  4 5          block(3)  8 9
          2 3                    6 7                    3 2
             2x2                    2x2                    2x2
```



```
int arr[3][3][3]=
        { {{10,20,30},{40,50,60},{70,80,90}},     //elements of block 1
          {{11,22,33},{44,55,66},{77,88,99}},     //elements of block 2
          {{12,23,34},{45,56,67},{78,89,90}}      //elements of block 3
        };

block(1)  10 20 30        block(2)  11 22 33        block(3)  12 23 34
          40 50 60                  44 55 66                  45 56 67
          70 80 90                  77 88 99                  78 89 90
                 3x3                       3x3                       3x3
```

Passing by Value or Reference

Consider a swap function to swap two integers.

Passing by Value or Reference

Consider a swap function to swap two integers.

```
#include<stdio.h>
void swap (int a, int b) {
 int temp = a;
 a = b;
 b = temp;
 return;
}
void main( ) {
 int x = 20;
 int y = 40;
 swap(x, y);
 printf("x= %d;y= %d\n", x, y);
}
```

Passing by Value or Reference

Consider a swap function to swap two integers.

```
#include<stdio.h>
void swap (int a, int b) {
 int temp = a;
 a = b;
 b = temp;
 return;
}
void main( ) {
 int x = 20;
 int y = 40;
 swap(x, y);
 printf("x= %d;y= %d\n", x, y);
}
```

- What is the output of the program?

Passing by Value or Reference

Consider a swap function to swap two integers.

```
#include<stdio.h>
void swap (int a, int b) {
 int temp = a;
 a = b;
 b = temp;
 return;
}
void main( ) {
 int x = 20;
 int y = 40;
 swap(x, y);
 printf("x= %d;y= %d\n", x, y);
}
```

- What is the output of the program?
- What is the output if we print `a` and `b` inside the function swap? (at the end of the function).

Passing by Value or Reference

Consider a swap function to swap two integers.

```
#include<stdio.h>
void swap (int a, int b) {
 int temp = a;
 a = b;
 b = temp;
 return;
}
void main( ) {
 int x = 20;
 int y = 40;
 swap(x, y);
 printf("x= %d;y= %d\n", x, y);
}
```

- What is the output of the program?
- What is the output if we print `a` and `b` inside the function swap? (at the end of the function).
- Variables are passed by value in C

Passing by Value or Reference

Consider a swap function to swap two integers.

```
#include<stdio.h>
void swap (int a, int b) {
 int temp = a;
 a = b;
 b = temp;
 return;
}
void main( ) {
 int x = 20;
 int y = 40;
 swap(x, y);
 printf("x= %d;y= %d\n", x, y);
}
```

- What is the output of the program?
- What is the output if we print `a` and `b` inside the function swap? (at the end of the function).
- Variables are passed by value in C always!

Passing by Value or Reference

Consider a swap function to swap two integers.

```
#include<stdio.h>
void swap (int a, int b) {
 int temp = a;
 a = b;
 b = temp;
 return;
}
void main( ) {
 int x = 20;
 int y = 40;
 swap(x, y);
 printf("x= %d;y= %d\n", x, y);
}
```

- What is the output of the program?
- What is the output if we print `a` and `b` inside the function swap? (at the end of the function).
- Variables are passed by value in C always!

Take-away: This is an incorrect swap program.

Passing by Value or Reference

Consider a swap function to swap two integers.

```
#include<stdio.h>
void swap (int a, int b) {
 int temp = a;
 a = b;
 b = temp;
 return;
}
void main( ) {
 int x = 20;
 int y = 40;
 swap(x, y);
 printf("x= %d;y= %d\n", x, y);
}
```

- What is the output of the program?
- What is the output if we print `a` and `b` inside the function swap? (at the end of the function).
- Variables are passed by value in C always!

Take-away: This is an incorrect swap program.
How do we write a correct swap program?

Passing by Value or Reference

Consider a swap function to swap two integers.

```
#include<stdio.h>
void swap (int a, int b) {
 int temp = a;
 a = b;
 b = temp;
 return;
}
void main( ) {
 int x = 20;
 int y = 40;
 swap(x, y);
 printf("x= %d;y= %d\n", x, y);
}
```

- What is the output of the program?
- What is the output if we print `a` and `b` inside the function swap? (at the end of the function).
- Variables are passed by value in C always!

Take-away: This is an incorrect swap program.
How do we write a correct swap program? needs pointers.

replace string

```
#include<stdio.h>

void replace(char s[10]) {
   int i = 0;
   while (s[i] != 0) {
     if (s[i] == 's')
         s[i] = 'S';
      i++;
   }
   printf("%s\n", s);
}

int main() {
  char arr[10] = "Maths";
  replace(arr);

  printf("%s\n", arr);
}
```

replace string

```
#include<stdio.h>

void replace(char s[10]) {
   int i = 0;
   while (s[i] != 0) {
     if (s[i] == 's')
         s[i] = 'S';
      i++;
   }
   printf("%s\n", s);
}

int main() {
  char arr[10] = "Maths";
  replace(arr);

  printf("%s\n", arr);
}
```

- What is the output of the program?

replace string

```
#include<stdio.h>

void replace(char s[10]) {
   int i = 0;
   while (s[i] != 0) {
     if (s[i] == 's')
         s[i] = 'S';
      i++;
   }
   printf("%s\n", s);
}

int main() {
  char arr[10] = "Maths";
  replace(arr);

  printf("%s\n", arr);
}
```

- What is the output of the program?
- Recall: Variables are passed by value in C always!

replace string

```
#include<stdio.h>

void replace(char s[10]) {
   int i = 0;
   while (s[i] != 0) {
     if (s[i] == 's')
         s[i] = 'S';
      i++;
   }
   printf("%s\n", s);
}

int main() {
  char arr[10] = "Maths";
  replace(arr);

  printf("%s\n", arr);
}
```

- What is the output of the program?
- Recall: Variables are passed by value in C always!
- `printf("%p\n", arr);`
  `printf("%p\n", &arr);`

replace string

```
#include<stdio.h>

void replace(char s[10]) {
   int i = 0;
   while (s[i] != 0) {
     if (s[i] == 's')
         s[i] = 'S';
      i++;
   }
   printf("%s\n", s);
}

int main() {
  char arr[10] = "Maths";
  replace(arr);

  printf("%s\n", arr);
}
```

- What is the output of the program?
- Recall: Variables are passed by value in C always!
- `printf("%p\n", arr);`
  `printf("%p\n", &arr);`
- The address of the array is copied when the function is called.

replace string

```
#include<stdio.h>

void replace(char s[10]) {
   int i = 0;
   while (s[i] != 0) {
     if (s[i] == 's')
         s[i] = 'S';
      i++;
   }
   printf("%s\n", s);
}

int main() {
  char arr[10] = "Maths";
  replace(arr);

  printf("%s\n", arr);
}
```

- What is the output of the program?
- Recall: Variables are passed by value in C always!
- `printf("%p\n", arr);`
  `printf("%p\n", &arr);`
- The address of the array is copied when the function is called.
- This behaves like pass by reference which is supported by other languages like Pascal, C++.

replace string

```
#include<stdio.h>

void replace(char s[10]) {
   int i = 0;
   while (s[i] != 0) {
     if (s[i] == 's')
         s[i] = 'S';
      i++;
   }
   printf("%s\n", s);
}

int main() {
  char arr[10] = "Maths";
  replace(arr);

  printf("%s\n", arr);
}
```

- What is the output of the program?
- Recall: Variables are passed by value in C always!
- `printf("%p\n", arr);` `printf("%p\n", &arr);`
- The address of the array is copied when the function is called.
- This behaves like pass by reference which is supported by other languages like Pascal, C++.

replace string

```c
#include<stdio.h>

void replace(char s[10]) {
   int i = 0;
   while (s[i] != 0) {
     if (s[i] == 's')
         s[i] = 'S';
      i++;
   }
   printf("%s\n", s);
}

int main() {
  char arr[10] = "Maths";
  replace(arr);

  printf("%s\n", arr);
}
```

- What is the output of the program?
- Recall: Variables are passed by value in C always!
- `printf("%p\n", arr);` `printf("%p\n", &arr);`
- The address of the array is copied when the function is called.
- This behaves like pass by reference which is supported by other languages like Pascal, C++.

Selection Sort Modularized

Selection Sort : Sort n numbers in descending order

Pseudo-code :

for i ranging from 1 to n

- maxindex $=$ the index of the max element in the part of the array indexed from i to n. Find maxindex.
- swap elements array[i] and array[maxindex];

Selection Sort Modularized

Selection Sort : Sort n numbers in descending order

Pseudo-code :

for i ranging from 1 to n

- maxindex = the index of the max element in the part of the array indexed from i to n. Find maxindex.
- swap elements array[i] and array[maxindex];

Subtasks identified:

`FindMax(A, i,n)` : find the index of maxelement in the subarray from i to n.

`Swap(A, i,j)` : swap i^{th} and j^{th} elements of A.

Selection Sort: Modularized

```
#include<stdio.h>
int getMaxIndex(int A[], int low, int high) {
  int maxIndex = low; // omitted braces below to fit in screen.
  for (int j=low+1; j <= high; j++)
       if (A[j] > A[maxIndex])
          maxIndex = j;
  return maxIndex;
}
void swapA (int A[], int i, int j) {
   int temp = A[i];    A[i] = A[j];    A[j] = temp;
}
int main() {
  int arr [10] = {25, 7, 9, 30, 44, 8, -12, 7, 8, 10};
  for (int i=0; i<10; i++) {
     int mIndex = getMaxIndex(arr, i, 9);
     swapA(arr, mIndex, i);
  }
}
```