CS 2124: DATA STRUCTURES
Spring 2024

Third Lecture (Part 1)

Topics: Creating Libraries, Stacks

Topics

- Assignment 1 (review)
- Tutoring for the Spring 2024 (announcement)
- Libraries
- Library Vs Header Files
  - Static vs Dynamic
  - Standard library
  - Library implementation
  - Library implementation & optimization
  - Library review

Tutoring for the Spring 2024

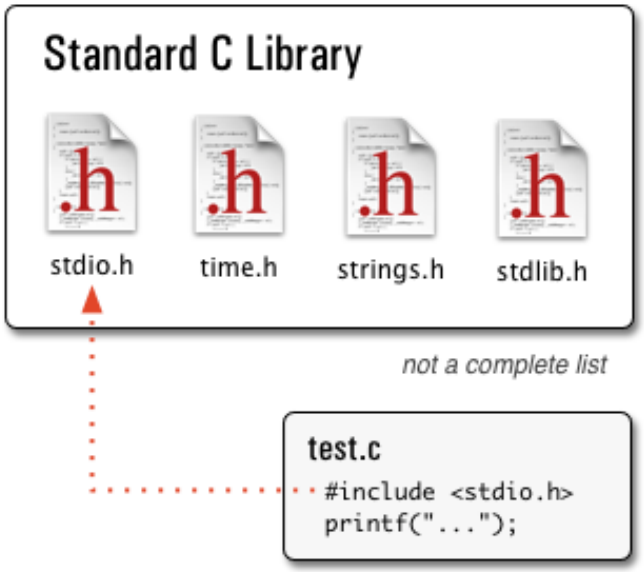
- **Our Location**
- Tomás Rivera Center for Academic Excellence
  MS 2.02.18 (Sombrilla Plaza, behind the Rowdy statue)
  UTSA Main Campus
  (210) 458-6783
- Hours of Operation: Monday-Thursday: 8 a.m. - 6 p.m.
  Friday: 8 a.m. - 5 p.m.

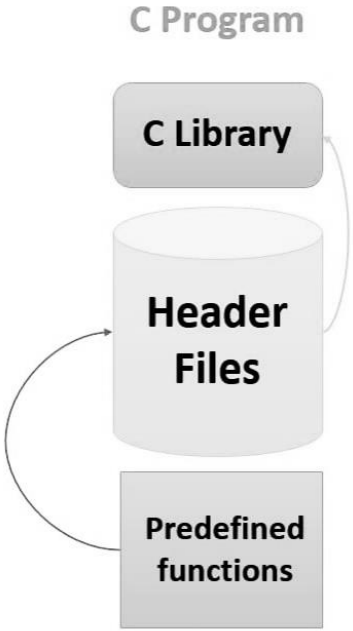
CS2124			Data Structures			
	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
10:00 AM	X	X	X	X	X	X
11:00 AM	X	X	X	X	X	X
12:00 PM	X	X	X	X	X	X
1:00 PM		X	X	X	X	X
2:00 PM		X	X	X	X	X
3:00 PM	X	X	X	X	X	X
4:00 PM	X	X	X	X	X	X
5:00 PM	X	X	X	X	X	X
6:00 PM	X	X		X	X	X
7:00 PM	X	X		X	X	X

Libraries

- Libraries in programming languages are collections of prewritten code (i.e. files, programs, routines, scripts, or functions) that users can use to optimize tasks.

- This collection of reusable code is usually targeted for specific common problems i.e. A header file contains the declaration of functions like iostream, printf, scanf, pow, etc. A library contains its definition





Library Vs Header Files

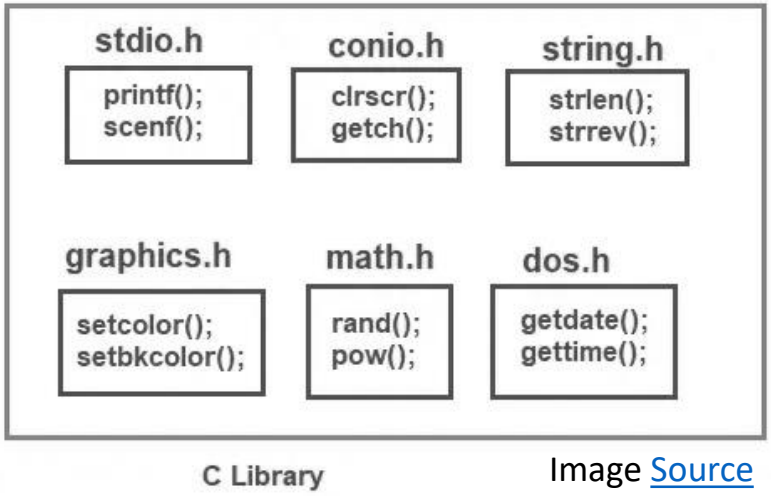
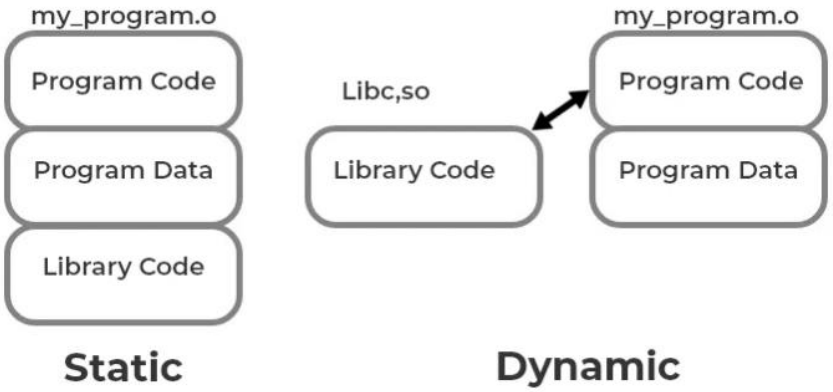


Image Source

- **Header Files**
  - The files that tell the compiler how to call some functionality (without knowing how the functionality actually works) are called header files.
  - They contain the function prototypes. They also contain Data types and constants used with the libraries.
  - We use #include to use these header files in programs. These files end with .h extension.

- **Library**
  - Library is the place where the actual functionality is implemented i.e. they contain function body. Libraries have mainly two categories (*details next slide*):
    1. Static
    2. Shared or Dynamic

Library Vs Header Files
(Static vs Dynamic)



- **Static:** Static libraries contains object code linked with an end user application and then they **become the part of the executable**. These libraries are specifically used at compile time which means the library should be present in correct location when user wants to compile the program.

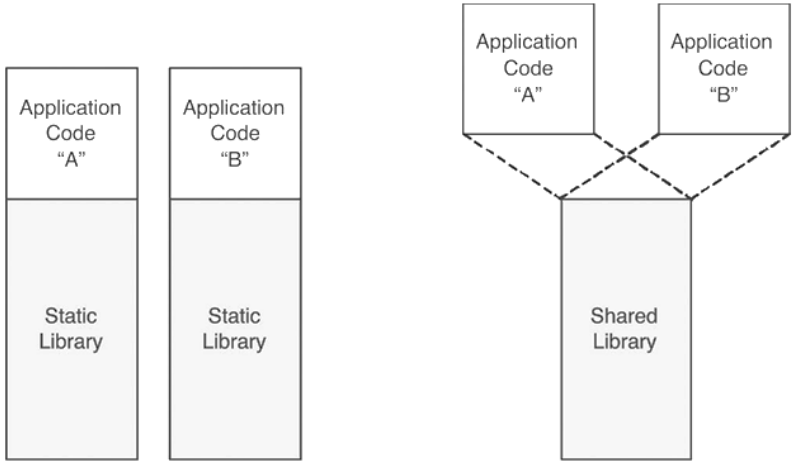
- **Shared or Dynamic:** These libraries are **only required at run-time** i.e. these libraries are linked with the program at compile time to resolve undefined references and then they are distributed to the application so that the application can load it at run time.
  - *For example, when we open our game folders we can find many .dll(dynamic link libraries) files. As these libraries can be shared by multiple programs, they are also called as shared libraries.*

What Are DLLs (*dynamic link libraries)*?

- Contain shared code which multiple program use
- Several program can use the same DLL file at the same time i.e. once it is loaded in memory
  - Example: windows pop-up dialog box, device drivers
- They are loaded once the program asks for them
- DLL provides modularity

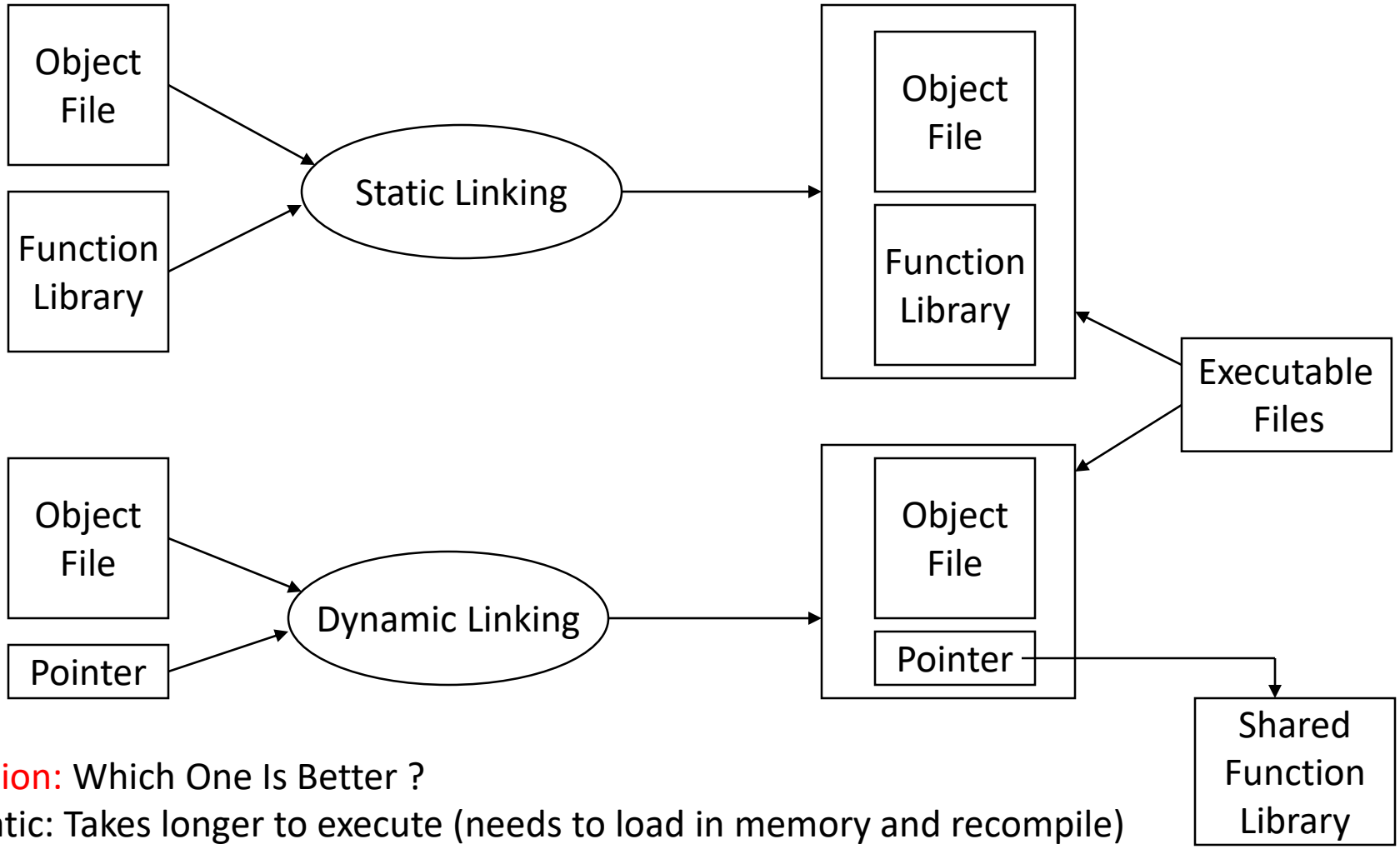
- **DLL HELL !!**
- If DLL is modified, there is no guarantee that the existing program can use the same version of the file
- Microsoft has been pushing to standardize it but it is still an ongoing process.

Library Vs Header Files
(Static vs Dynamic)



Static libraries	Dynamic libraries
Are part of the built environment. Object files are added to the executable. Have the .a extension. **Advantages :** • They are faster since all modules are in the same file (once every thing is loaded in memory). • Their distribution and installation are easier. • Avoid dependency problems. **Drawbacks :** • Use more memory space to create a copy by each executable file • Slower compilation process all source is re-compiled.	Are part of the run-time environment address of the object files are added to the execution file. Have .so extension **Advantages:** • Use less memory space. • Source code is not re-compiled • Faster compilation process **Drawbacks:** • Could cause dependency problems in application • Compatibility problems if the library is removed

Library Vs Header Files (Static vs Dynamic)



Question: Which One Is Better ?

- Static: Takes longer to execute (needs to load in memory and recompile)
- Dynamic: Fast execution but compatibility issues

Libraries (Standard library examples)

We have used a number of builtin libraries of C. For example here are the library includes from the first assignment:

- #include <stdlib.h>

This includes many useful functions. The functions of the stdlib.h (standard library) can be classified into: conversion, memory, process control, sort and search, mathematics.

- #include <stdio.h>

It is used to include the standard input output library functions like printf() function.

- #include <time.h>

Find the runtimes of various function / Contains time and date function declarations to provide standardized access to time/date manipulation and formatting.

- #include <math.h>

Includes many common math function.

Libraries (Standard library examples)

- #include <stdio.h>
- 160 lines of code

```
00053 #define __SLBF  0x0001          /* line buffered */
00054 #define __SNBF  0x0002          /* unbuffered */
00055 #define __SRD   0x0004          /* OK to read */
00056 #define __SWR   0x0008          /* OK to write */
00057         /* RD and WR are never simultaneously asserted */
00058 #define __SRW   0x0010          /* open for reading & writing */
00059 #define __SEOF  0x0020          /* found EOF */
00060 #define __SERR  0x0040          /* found error */
00061 #define __SMBF  0x0080          /* _buf is from malloc */
00062 #define __SAPP  0x0100          /* fdopen()ed in append mode - so must  write to end */
00063 #define __SSTR  0x0200          /* this is an sprintf/snprintf string */
00064 #define __SOPT  0x0400          /* do fseek() optimisation */
00065 #define __SNPT  0x0800          /* do not do fseek() optimisation */
00066 #define __SOFF  0x1000          /* set iff _offset is in fact correct */
00067 #define __SMOD  0x2000          /* true => fgetline modified _p text */
00068 #if defined(__CYGWIN__) || defined(__CYGWIN__)
00069 #define __SCLE        0x4000          /* convert line endings CR/LF <-> NL */
00070 #endif
00071
00072 /*
00073  * The following three definitions are for ANSI C, which took them
00074  * from System V, which stupidly took internal interface macros and
00075  * made them official arguments to setvbuf(), without renaming them.
00076  * Hence, these ugly _IOxxx names are *supposed* to appear in user code.
00077  *
00078  * Although these happen to match their counterparts above, the
00079  * implementation does not rely on that (so these could be renumbered).
00080  */
00081 #define _IOFBF  0               /* setvbuf should set fully buffered */
00082 #define _IOLBF  1               /* setvbuf should set line buffered */
00083 #define _IONBF  2               /* setvbuf should set unbuffered */
00084
00085 #ifndef NULL
00086 #define NULL    0
00087 #endif
```

Source: gnu.org

Libraries (Standard library examples)

#include <time.h>

The time.h header defines four variable types (2 macro* and various functions for manipulating date and time).

Variable

1. size_t: This is the unsigned integral type and is the result of the sizeof keyword. size_t is commonly used for array indexing and loop counting
2. clock_t: This is a type suitable for storing the processor time.
3. time_t: This is a type suitable for storing the calendar time.
4. struct tm: This is a structure used to hold the time and date.

```
struct tm
{
int tm_sec; /* seconds, range 0 to 59 */
int tm_min; /* minutes, range 0 to 59 */
...
}
```

**whenever the compiler encounters a macro in a program, it will replace it with the macro value*

Libraries (Standard library examples)

#include <time.h>

Macro

A macro is a name given to a block of C statements as a pre-processor directive.

1. NULL: This macro is the value of a null pointer constant.
2. CLOCKS_PER_SEC: This macro represents the number of processor/system clocks per second.

A computer's processor clock speed determines how quickly the central processing unit (CPU) can retrieve and interpret instructions.

A good speed for gaming is widely considered anything from 3.5 to 4.0 GHz—that's 3.5 to 4 billion commands a second.

Libraries (Standard library examples)

#include <time.h>
CLOCKS_PER_SEC

```c
#include <time.h>
#include <stdio.h>
int main () {
   clock_t start_t, end_t;
   double total_t;
   int i;
   start_t = clock();
   printf("Starting of the program (Start Time) = %ld\n", start_t);
   printf("Going to scan a loop (Start Time) = %ld\n", start_t);
   for(i=0; i< 2024; i++) {
   }
   end_t = clock();
   printf("End of the loop (End Time) = %ld\n", end_t);
      total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
   printf("Total time taken by CPU (End Time - Start Time)/clock per_sec: %f\n", total_t  );
   return(0);
}
```

Libraries (Standard library examples)

#include <time.h>
CLOCKS_PER_SEC

```c
#include <time.h>
#include <stdio.h>
int main () {
   clock_t start_t, end_t;
   double total_t;
   int i;
   int arr[2][3][2]
        = { { { 0, 6 }, { 1, 7 }, { 2, 8 } },
            { { 3, 9 }, { 4, 10 }, { 5, 11 } } };
   start_t = clock();
   printf("Start Time = %ld\n", start_t);
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 2; ++k) {
                printf("Element at arr[%i][%i][%i] = %d\n",
                        i, j, k, arr[i][j][k]);
            }
        }
    }
   end_t = clock();
   printf("End Time = %ld\n", end_t);
      total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
printf("Total time (End Time - Start Time)/clock per_sec: %f\n", total_t );
   return(0);
}
```

Which element will be at arr[1][0][0]?

Libraries (Standard library examples)

#include <time.h>
CLOCKS_PER_SEC

```c
#include <time.h>
#include <stdio.h>
int main () {
   clock_t start_t, end_t;
   double total_t;
   int i;
   int arr[] = {1,2};
   start_t = clock();
   printf("Start Time = %ld\n", start_t);
    for (int i = 0; i < 2; ++i) {
                printf("Element at arr[%i] = %d\n",
                        i, arr[i]);  }
   end_t = clock();
   printf("End Time = %ld\n", end_t);
      total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
printf("Total time (End Time - Start Time)/clock per_sec: %f\n", total_t );
   return(0);
}
```

Libraries (Standard library examples)

#include <time.h>

CLOCKS_PER_SEC

```c
#include <time.h>
#include <stdio.h>
int main () {
   clock_t start_t, end_t;
   double total_t;
   int i;
   int arr[] = {1,2};
   start_t = clock();
   printf("Start Time = %ld\n", start_t);
   for (int j = 0; j < 2; ++j){
    for (int i = 0; i < 2; ++i) {
                printf("Element at arr[%i] = %d J=%d\n",
                        i, arr[i], j);  } }
   end_t = clock();
   printf("End Time = %ld\n", end_t);
      total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
printf("Total time (End Time - Start Time)/clock per_sec: %f\n", total_t );
   return(0);
}
```

Libraries (Standard library examples)

#include <time.h>
CLOCKS_PER_SEC

```c
#include <time.h>
#include <stdio.h>
int fun () {
   clock_t start_t, end_t;
   double total_t;
   int i;
   int arr[] = {1,2};
   start_t = clock();
   printf("Start Time (In Function) = %ld\n", start_t);
   for (int i = 0; i < 2; ++i) {
                printf("Element at arr[%i] = %d \n",
                        i, arr[i]);  }
   end_t = clock();
   printf("End Time (In Function) = %ld\n", end_t);
      total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
printf("Total time in Function(End Time - Start Time)/clock per_sec: %f\n", total_t );
}
int main ()
{
    clock_t start_tf, end_tf;
    double total_tf;
    start_tf = clock();
    printf("Go to Function \n");
    fun();
    printf("Back in Main\n");
    end_tf = clock();
    printf("End Time (Main) = %ld\n", end_tf);
      total_tf = (double)(end_tf - start_tf) / CLOCKS_PER_SEC;
printf("Total time in main (End Time - Start Time)/clock per_sec: %f\n", total_tf );
     return(0);
}
```

Library
(implementation example)

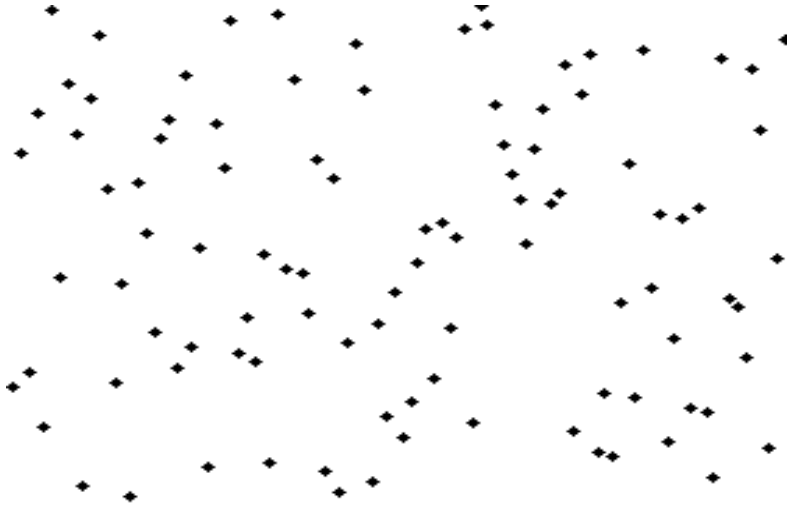
- The code (Next Slide) fills an array with random numbers, sorts them using a bubble sort, and then displays the sorted list.

- Since both the array a[] and the constant MAX are known globally, the function you create needs no parameters, nor does it need to return a result. However, you should use local variables for x, y, and t.

- But why use local variable ?

- Local variable are easy to debug as global variable may be difficult to track when updated

- Local Variable pass values between subroutines to avoid errors caused by unforeseen changes



Bubble sort animation source: Link

Library

```
1.  #include <stdio.h>
2.  #define MAX 10
3.  int a[MAX];
4.  int rand_seed=10;
5.  int rand()
6.  {
7.     rand_seed = rand_seed * 100;
8.     return (unsigned int)(rand_seed /10) % 11;
9.  }
10. void main()
11. {
12.    int i,t,x,y;
13.    /* fill array */
14.    for (i=0; i < MAX; i++)
15.    {
16.       a[i]=rand();
17.       printf("Unsorted %d: %d\n", i, a[i]);
18.    }
```

```
18. /* bubble sort the array */
19.    for (x=0; x < MAX-1; x++)
20.       for (y=0; y < MAX-x-1; y++)
21.          if (a[y] > a[y+1])
22.          {
23.             t=a[y];
24.             a[y]=a[y+1];
25.             a[y+1]=t;
26.          }
27.    /* print sorted array */
28.    printf("--------------------\n");
29.    for (i=0; i < MAX; i++)
30.       printf("Sorted %d: %d\n", i,a[i]);
31. }
```

```c
#include <stdio.h>
#define MAX 10
int a[MAX];
int rand_seed=10;
int rand()
{
    rand_seed = rand_seed * 100;
    return (unsigned int)(rand_seed /10) % 11;
}
void main()
{
    int i,t,x,y;
    /* fill array */
    for (i=0; i < MAX; i++)
    {
        a[i]=rand();
        printf("Unsorted %d: %d\n", i, a[i]);
    }
 /* bubble sort the array */
    for (x=0; x < MAX-1; x++)
        for (y=0; y < MAX-x-1; y++)
            if (a[y] > a[y+1])
            {
                t=a[y];
                a[y]=a[y+1];
                a[y+1]=t;
            }
    /* print sorted array */
    printf("--------------------\n");
    for (i=0; i < MAX; i++)
        printf("Sorted %d: %d\n", i,a[i]);
}
```

```
rand() % 10
```

Library

Original code (Slide 23)

```
1.  #define MAX 10
2.  int a[MAX];
3.  int rand_seed=10;
4.  int rand()
5.  {
6.     rand_seed = rand_seed * 100;
7.     return (unsigned int)(rand_seed /10) % 11;
8.  }
9.  void bubble_sort(int m)
10. { //earlier sorting was in main function
11.    int x,y,t;
12.     for (x=0; x < m-1; x++)
13.       for (y=0; y < m-x-1; y++)
14.          if (a[y] > a[y+1])
15.          {
16.             t=a[y];
17.             a[y]=a[y+1];
18.             a[y+1]=t;
19.          }     }
20. void main()
21. {
22.    int i,t,x,y;
23.    /* fill array */
24.    for (i=0; i < MAX; i++)
25.    {
26.       a[i]=rand();
27.       printf("Unsorted %d: %d\n", i, a[i]);
28.    }
29.    bubble_sort(MAX);
30.    /* print sorted array */
31.    printf("-------------------\n");
32.    for (i=0; i < MAX; i++)
33.       printf("Sorted %d: %d\n", i,a[i]);
34. }
```

Try to implement the code

Library

(Further optimizing or creating library of code on Slide 25)

1. You can also generalize the bubble_sort function even more by passing in a[] as a parameter:

bubble_sort(int m, int a[])

2. This line says, "Accept the integer array a[] of any size as a parameter." Nothing in the body of the bubble_sort function needs to change. To call bubble_sort, change the call to:

bubble_sort(MAX, a);

3. Note that &a has not been used in the function call even though the sort will change ‘a’. The reason for this will become clear once you understand pointers.

4. Enter the following header file and save it to a file named util.h.

```
/* util.h */

extern int rand();

extern void bubble_sort(int, int []);
```

5. These two lines are function prototypes. The word "extern" in C represents functions that will be linked in later. If you are using an old-style compiler, remove the parameters from the parameter list of **bubble_sort**.

Library

(Code after following the steps on Slide 26)

- Enter the following code into a file named util.c

```
1. /* util.c */
2. #include "util.h"
3. int rand_seed=10;
4.  int rand()
5.  {
6.      rand_seed = rand_seed * 100;
7.      return (unsigned int)(rand_seed /10) % 11;
8.  }
9.  void bubble_sort(int m,int a[])
10. {
11.   int x,y,t;
12.   for (x=0; x < m-1; x++)
13.     for (y=0; y < m-x-1; y++)
14.       if (a[y] > a[y+1])
15.       {
16.         t=a[y];
17.         a[y]=a[y+1];
18.         a[y+1]=t;
19.       }
20. }
```

```
/* util.h */
extern int rand();
extern void bubble_sort(int, int []);
```

Try to implement the code

Library

- Note that the file includes its own header file (util.h, on slide 27) and that it uses quotes instead of the symbols < and > , which are used only for system libraries.

- As you can see, this looks like normal C code.

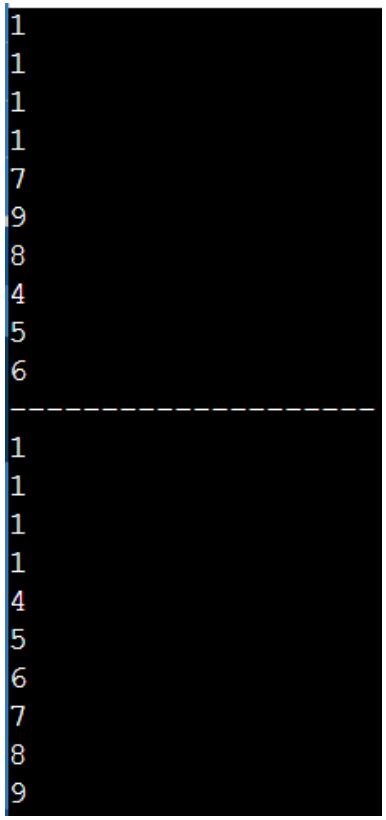
- Note that the variable rand_seed, because it is not in the header file, it cannot be seen or modified by a program using this library.

- This is called information hiding. Adding the word static in front of int enforces the hiding completely.

Library

- Enter the following main program in a file named main.c.

- This code includes the utility library. The main benefit of using a library is that the code in the main program is much shorter.



Output

```
1.  #include <stdio.h>
2.  #include "util.h"
3.  #define MAX 10
4.  int a[MAX];
5.  void main()
6.  {
7.      int i,t,x,y;
8.      /* fill array */
9.      for (i=0; i < MAX; i++)
10.     {
11.         a[i]=rand();
12.         printf("%d\n",a[i]);
13.     }
14.     bubble_sort(MAX,a);
15.     /* print sorted array */
16.     printf("--------------------\n");
17.     for (i=0; i < MAX; i++)
18.         printf("%d\n",a[i]);
19. }
```

```c
#include <stdio.h>
int rand_seed=10;
int rand()
{
    rand_seed = rand_seed * 100;
    return (unsigned int)(rand_seed /10) % 11;
}
void bubble_sort(int m,int a[])
{
    int x,y,t;
     for (x=0; x < m-1; x++)
        for (y=0; y < m-x-1; y++)
            if (a[y] > a[y+1])
            {
                t=a[y];
                a[y]=a[y+1];
                a[y+1]=t;
            }
}
```

```c
#define MAX 10
int a[MAX];
void main()
{
    int i,t,x,y;
    /* fill array */
    for (i=0; i < MAX; i++)
    {
        a[i]=rand();
        printf("Random Number: %d\n",a[i]);
    }
    bubble_sort(MAX,a);
    /* print sorted array */
    printf("--------------------\n");
    for (i=0; i < MAX; i++)
        printf("Sorted Numbers: %d\n",a[i]);
}
```

```
Random Number: 1
Random Number: 1
Random Number: 1
Random Number: 1
Random Number: 7
Random Number: 9
Random Number: 8
Random Number: 4
Random Number: 5
Random Number: 6
--------------------
Sorted Numbers: 1
Sorted Numbers: 1
Sorted Numbers: 1
Sorted Numbers: 1
Sorted Numbers: 4
Sorted Numbers: 5
Sorted Numbers: 6
Sorted Numbers: 7
Sorted Numbers: 8
Sorted Numbers: 9
```

*As I am using online C compiler so I cannot save **#include "util.h".** Students should implement the code on slide 27 and slide 29)

Try to implement the code

Library

- Based on the example:
  - Initial Code (Slide 23)
  - Optimal Code (Slide 29)

  - Initial code was long and debugging can be a concern (i.e. If you are working with a code of more then 500 lines)
  - Optimal code is easy to understand and to debug
    - With its module based approach the segments of the code can be reuse later on
    - Module based modification or further optimizing is comparatively easy to do

- **Example:** If you are working in a software house and they a MIS or CMC software's are developed on regular bases. The common modules can be made in form of libraries and can be used again. Even in case of minor modification a module can easily be modified and re-used. Such an approach can save a lot of coding hours and effort from the programmers. Plus existing modules can be further optimized based on ongoing work and experience.

End Of Lecture
Questions?