UNIVERSITY OF MASSACHUSETTS
DARTMOUTH

ECE160: Foundations of Computer Engineering I

Lecture #29 – Final Review

Instructor: Dr. Liudong Xing
SENG-213C, lxing@umassd.edu
ECE Dept.

Final Exam

- Time: **8-11am, May 1, Monday**
- Please arrive at the class on time; no make up time will be given for late arrivals.
- Form:
  - Open book open notes
  - Calculators are NOT allowed
  - Visual Studio is NOT allowed
  - ChatGPT is NOT allowed
- Preparation:
  - Exam#1, #2, #3
  - Lecture #29 (refer to more details in Lecture notes #2 - #28)
  - Lab #1 - #12
  - HW#1 - #4

Format of Problems (8)

- Problem#1: True/False
  - Problem #1 in Exam#1-#3
- Problem #2: Number conversions
  - Problem #2 in Exam#1
- Problem #3: Correct errors in programs
  - Problem #4 in Exam#1
  - Problem #2 in Exam#2, #3
- Problems #4-#7:Specify outputs of given programs
  - Problem #3 in Exam#1
  - Problem #3, #4 in Exam#2, #3
- Problem #8: Write or complete programs with functions
  - Problem #5 in Exam#1-#3

Exam#1: Lectures #2 - #10

- Number systems (L#2)
- Introduction to C programming (L#3)
- Data types and variables (L#4)
- Constants (L#5)
- Formatted input/output (L#6 & 7)
- Expressions (L#8 & 9)
- Two-way selection: if…else (L#10)
- **Exam#1 review (L#11)**

Exam#2: Lectures #12 - #18

- Multi-way selection: switch and if-else-if (L#12)
- Loops (L#13)
- Functions (L#14 ~ 17)
- Files I (L#18)
- **Exam#2 review (L#19)**

Exam#3: Lectures #20 - #26

- Files II (L#20)
- Arrays (L#21-23)
- Array sorting (L#24)
- Strings (L#25)
- Pointers (L#26)
- **Exam#3 review (L#27)**
- Pointers and arrays (L#28)

Number Systems (L#2)

1. Basic number systems concepts (base, positional/place value, symbol value)
2. How to work with numbers represented in binary, octal, and hexadecimal number systems
3. How to convert back and forth between decimal numbers and their binary, octal, and hexadecimal equivalents
4. How to abbreviate binary numbers as octal or hexadecimal numbers
5. How to convert octal and hexadecimal numbers to binary numbers

Identifiers and Naming Rules (L#3)

- Identifiers are used to name data and other objects (e.g. functions) in our program.
- C is case sensitive
  - Celsius, celsius, and CELSIUS are three different identifiers.
- Rules
  - The first character can not be a digit. It has to be an alphabetic character or underscore.
  - The identifier name must consist only of alphabetic characters, digits, or underscores.
  - First 31 characters of an identifier are significant/used.
  - DO NOT use a C reserved word /keywords (e.g., **int**).

Two Types of Errors

- Syntax: the required form of the program punctuation, keywords (int, float, return, …) etc.
  - Examples:
    - putting a semicolon after main() is a compilation error
    - Forgetting to terminate a comment with */ is a compilation error.
  - The C compiler always catches these “syntax errors” or “compiler errors”
- Semantics (logic): what the program means
  - What you want it to do
  - The C compiler cannot catch these kinds of errors!
  - They can be extremely difficult to find

Standard Data Types (L#4)

- void: has no values
- int: a number without fraction part
  - 3 different sizes of the integer type: short int, int, long int
  - the size of int is machine dependent
  - C supports logical data type through the integer type
- char: a value that can be represented in the computer’s alphabet.
  - represented using 1 byte (ASCII code)
- float: a number with fraction part
  - 3 types of floating point numbers: float, double, long double

Variables

- Variables are named memory locations that have a type, identifier, and value.
- Each variable in the program must be declared and defined!
  - Declaration: to name a variable
  - Definition: to create a variable, to reserve memory for it
  - Usually a variable is declared and defined at the same time!
- The programmer must initialize any variable requiring prescribed data when the function starts

Constants (L#5)

- Four types: Integer (13), Character (‘a’), Floating point (2.3), String (“hello”)
- Three ways to code constants in the program:
  - Literal: an unnamed constant, the data itself (3.14)
  - Defined: use the preprocessor command define (*e.g.:* #define PI 3.14) --- the expression that follows the name replaces the name wherever it is found in the source program
  - Memory: Use a C type qualifier: *const (e.g.:* const float pi = 3.14;) --- memory constants fix the contents of a memory location

Formatted Output *printf()* (L#6) and Input *scanf()* (L#7)

printf(format string, data list);

- Conversion codes %d %c %f etc
- The number of conversion code should match the number of data/variables that follow the “format string”

scanf(format string, address list);

- The number of conversion code should match the number of addresses that follow the “format string”
- Each variable name in the address list must be preceded by an ampersand **&**.
- You can use field width like %2d, but there is no precision width in the input field specification. When scanf() finds a precision, it stops processing.

C Expressions (1)

- Types of expressions
  - Primary expressions: consist of only one operand with no operator
  - Binary expressions: formed by an *operand-operator-operand* combination
    - Multiplicative expressions: *, /, %
    - Additive expressions: +, -
  - Assignment expressions using assignment operator =
  - Postfix expressions: a++; a--;
  - Unary expressions:
    - Prefix increment/decrement: ++a; --a;
    - Sizeof()
    - plus/minus

C Expressions (2)

- A side effect is an action that results from the evaluation of an expression: *changing the value of a variable is a side effect*
  - side effects take place before the expression is evaluated: ++a;   --a;
  - side effects take place after the expression is evaluated: a++; a--;
- Precedence and associativity
  - Precedence determines the order in which different operations are evaluated.
  - Associativity determines how operators with the same precedence are grouped together in complex expressions (left, right)
  - Note that precedence is applied before associativity.

Two-Way Selection *if…else* (L#10)

- Logical data: true (1) or false (0)
  - C supports this through int type: zero (false), non-zero (true)

- 3 logical operators:
  - ! NOT, && (logical AND), || (logical OR)

- 6 relational operators

<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal
!=	not equal

Two-Way Selection *if…else* (2)

- *if…else* statement

```
if (expression)
     {
      Action 1
     }
    else
     {
    Action 2
    }
```

- Nested *if…else* statement: An *if…else* is included within another *if…else*

- Dangling *else* problem: when there is no matching *else* for every *if,* Solution: *Always pair an "else" to the most recent unpaired "if" in the current block!*

- Ternary conditional operator

  expression1 ? expression2 : expression3

  – This means that if expression1 is true, then the overall expression evaluates to expression 2, else it evaluates to expression3.

An Example

Good programming style:

Using indention

Line up opening and closing braces

```
#include "stdafx.h"
void main(void)
{
    int a,b;
    printf("Enter two integers:\n");
    scanf("%d%d",&a, &b);
    if(a >= b)
          {
                    if(a > b)
                              printf("%d > %d",a,b);
                    else
                              printf("%d == %d",a,b);


          }
    else
          {
                    printf("%d < %d", a, b);
          }
}
```

switch statements (Rules, L#12)

```
switch (expression)
{
  case constant-1:
                      statements
                      break;
  case constant-2:
                      statements
                      break;
   case constant-3:
                      statements
                      break;

        ……
    default:
                      statements
                      break;
}
```

- The control expression that *switch* tests must be an integral type, i.e., it can not be a float or a double for example.
- The expression followed by each case label must be a constant expression.
- Two *case* labels can not have the same value.
- However, two cases can have the same statements.
- The *switch* can include at most one *default* label. And it can be coded anywhere, but is traditionally coded last.

if-else-if control structure

```
if (expression-1)
      {
         statement-block-1
      }
else if (expression-2)
      {
         statement-block-2
      }
      ……

else if (expression-n)
      {
         statement-block-n
      }
else
      {
         statement-block-n+1
      }
```

Example

- Convert a numeric score to a letter grade
  - 90 or more → A
  - 80 - 90 → B
  - 70 - 80 → C
  - 60 - 70 → D
  - Below 60 → F

```
float score;
char grade;
int temp;
temp = score/10;
switch (temp)
{
    case 10:    grade = ‘A’;
                 break;
    case 9:     grade = ‘A’;
                 break;
    case 8:     grade = ‘B’;
                 break;
    case 7:     grade = ‘C’;
                 break;
    case 6:     grade = ‘D’;
                 break;
    default:    grade = ‘F’;
}
```

```
float score;
char grade;
if(score >= 90)
        grade = ‘A’;
else if(score >= 80)
        grade = ‘B’;
else if(score  >= 70)
       grade = ‘C’;
else if(score >= 60)
      grade = ‘D’;
else
     grade = ‘F’;
```

Loops (L#13)

- Three *C* loop statements
  - *while* loops
  - *do…while* loops
  - *for* loops

while vs. *do…while*

```
while (expression)
    {
        statement-1
        statement-2
          ……
        statement-n
    }
```

- Pre-test: loop-continuation condition is tested before the loop.
- No semicolon is needed at the end of the while statement!

```
do
{
    statement-1
    statement-2
      ……
    statement-n
} while (expression);
```

- Post-test: loop-continuation condition is tested after the loop.
- Semicolon is needed at the end of the *do…while* statement!!

Braces are not required if the loop body consists of only one statement

The for Loop

- General expression:

```
for(statement1;statement2;statement3)
{
    loop_body
}
```

  - statement1: contains initial value of control variable
  - statement2: a test expression containing inal value of control variable
  - statement3: increments/decrements the control variable
  - Braces are not required if the loop body consists of only one statement
  - The 3 expressions in the *for* structure are optional. The two semicolons are required.
  - Pre-test: loop-continuation condition (statement2) is tested before the loop.

Equivalence

```
 x= 2;
while (x < 13)
{
    printf(“%d\n”,x);
    x++;
}
```

```
for(x = 2; x < 13;x++)
{
            printf("%d\n",x);
}
```

```
x =2;
do
{
            printf(“%d\n”,x);
             x++;
} while(x < 13);
```

break vs. continue

break is used to escape from a loop (causes a loop to terminate).
continue is used to skip the remaining statements in the body of a structure and skip to the next iteration.

```
#include "stdio.h“
void main(void)
{
      int a;
      for(a =1; a <= 7; a++)
     {
           if(a == 4)
                 break;
          printf("%d\n", a);
      }
      printf("I got out of the loop at a==%d\n",a);
}
```

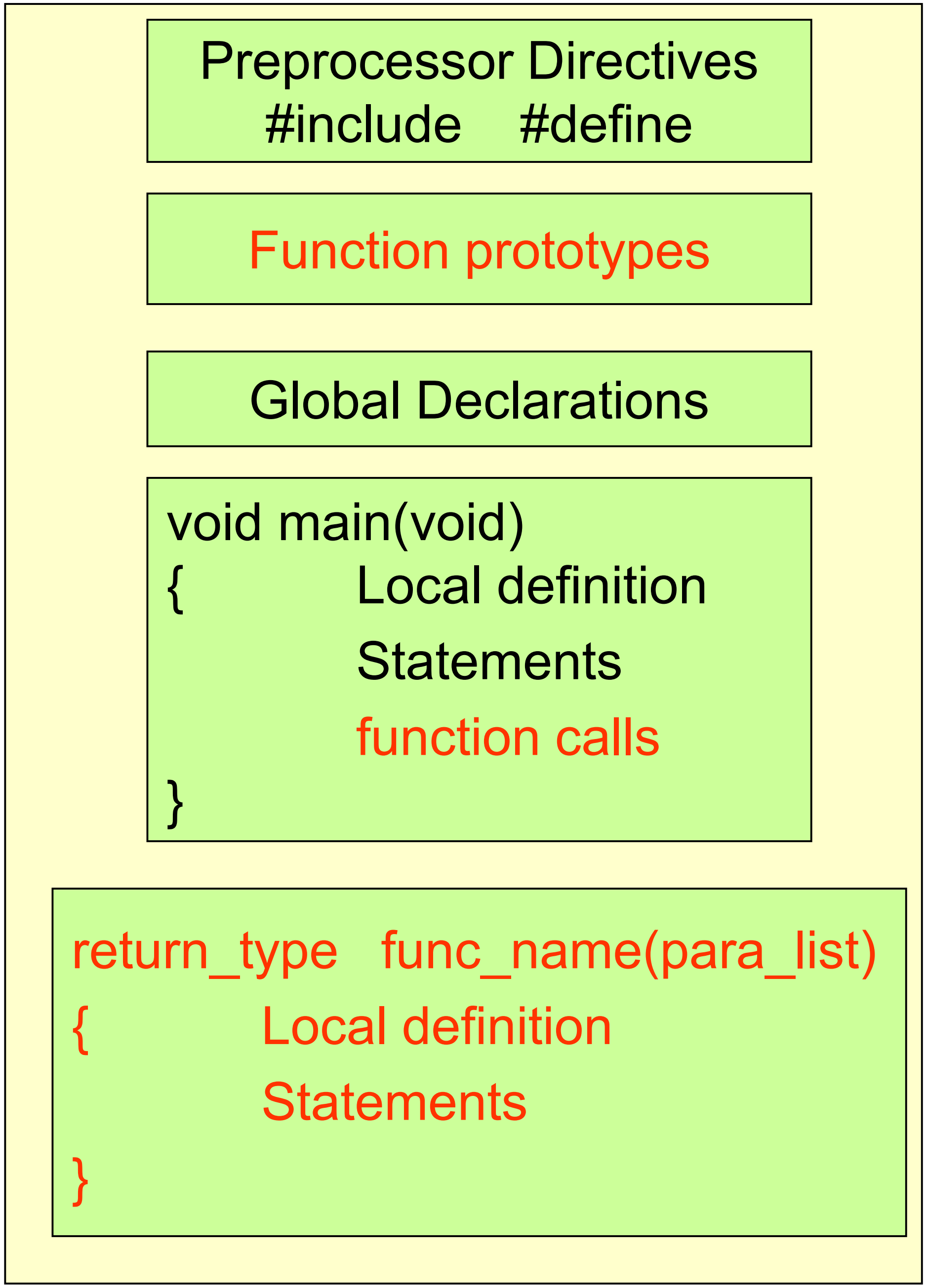
```
1
2
3
I got out of the loop at a==4
```

```
#include "stdio.h"
void main(void)
{
    int a;
    for(a =1; a <= 7; a++)
   {
            if (a == 4)
                 continue;
            printf("%d\n",a);
    }
}
```

```
1
2
3
5
6
7
```

Functions (L#14, 15)

- Every C program contains one and only one main()
- Functions must be declared before being used in a program
- Information can be passed between a function and the function that calls it



Preprocessor Directives
#include #define

Function prototypes

Global Declarations

```
void main(void)
{        Local definition
         Statements
         function calls
}
```

```
return_type   func_name(para_list)
{        Local definition
         Statements
}
```

Parameter Passing (L#16)

- Pass by value
  - A copy of the data (argument’s value) is passed to the called function.
  - The function can not modify the original variable’s value in the caller.

- Pass by reference.
  - The called function can modify the original variable’s value in the caller.
  - Any reference to a parameter is the same as a reference to the variable in the calling function
  - It uses the address operator (&) and indirection operator (*).

Example (Pass by Value)

What is the output of the program?

```
#include "stdio.h"
void test(int x);

void main(void)
{
    int a;
    a =2;
    test(a);
    printf(“the value of a after call is %d\n", a);
}

void test(int x)
{
    x = x + 5;
}
```

the value of a after call is 2

The value of a is copied into the memory cell reserved for x in the region of memory for test function

Example (Pass by Reference)

```
#include "stdio.h"
void test(int *x);
```

In a function prototype or header, * means the variable following * is to hold an address

```
void main(void)
{
    int a;
    a =2;
    test(&a);
    printf(" the value of a after call is %d\n", a);
}
```

& means the address of , a copy of the address of variable a is put into memory cell reserved for x in the memory region reserved for the variables of test function

the value of a after call is 7

```
void test(int *x)
{
    *x = *x + 5;
}
```

Standard Library Functions (L#16, 17)

- Mathematical functions
- Random number generation functions: srand(), rand()
- Character functions
  - – Classifying functions: int is…(int testchar);
  - – Converting functions: int to….(int oldchar);
- Use **include** statement to include the header files
  - – Example: **#include <stdio.h>**

Recursion (L#17)

- A repetitive process where a function calls itself.
- Recursive solution involves a two-way journey
    - First, we decompose the problem from top to bottom until reaching the base case
    - Then we solve it from bottom to top
- Examples:
    - factorial(n) (Lecture#17)
    - fibonacci(n) (Lecture#17, HW#4--Problem#4)
    - gcd(x,y) (Lab#8--Exercise#1)

Review: factorial(n)

```
long factorial(int n)
{
     int i;
     long fact=1;
     for(i=1; i<= n; i++)
      {
            fact = fact * i;
      }
     return fact;
}
```

```
long factorial(int n)
{
     if (n == 0)
            return 1;
     else
           return(n*factorial(n-1));
}
```

$$factorial(n) = \begin{cases} 1 & \text{if} \quad n = 0 \\ 1*2*...*(n-1)*n & \text{if} \quad n > 0 \end{cases}$$

$$factorial(n) = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * factorial(n-1) & \text{if} \quad n > 0 \end{cases}$$

Iterative Solution

Recursive Solution

Files (L#18, 20)

- A collection of information/related data treated as a unit
- Saved in secondary (auxiliary) memory like disks.
- Using files in C:
  - How to declare a file_pointer (**FILE**)
  - How to open a file (**fopen()**)
  - How to read from a file (**fscanf()**)
  - How to write to a file (**fprintf()**)
  - How to close a file (**fclose()**)

A Complete Example (Review)

```
#include "stdafx.h"
int main(void)
{
       FILE *fp;
       int num1=100;
       int num2=200;
       int num3=300;
       int a=0, b=0, c=0;

       fp = fopen(“Xing_file1.txt","w");
       if(!fp)
       {
          printf("I was not able to open file\n");
          return(1);
       }
       fprintf(fp,"%d\n%d\n%d\n", num1, num2, num3);
       if(fclose(fp) == EOF)
      {
        printf("I was not able to close file\n");
        return(2);
       }
```

```
fp = fopen(“Xing_file1.txt",“r");
if(!fp)
       {
          printf("I was not able to open file\n");
          return(1);
       }

fscanf(fp,"%d%d%d",&a,&b,&c);

printf(“a is %d\n b is %d\n c is %d\n",a,b,c);

if(fclose(fp) == EOF)
{
       printf("I was not able to close file\n");
     return(2);
}

}
```

Arrays (L#21, 22)

- An array is a fixed-size, sequenced collection of elements of the same data type.
- Index of the first element is 0!
- The array elements are stored in contiguous and increasing memory locations.
- Before use, an array has to be defined and declared.
  – Reserve memory space for the elements in the array!

Array Initialization (3 ways)

- At the definition time

```
int myarray[5]={1,2,10,15,0};
int myarray[] = {1,2,10,15,0};
```

- Inputting values form the keyboard

```
int myarray[5];
for(int i=0; i< 5; i++)
{        scanf(“%d”, &myarray[i]); }
```

- Assigning values

```
int myarray[5];
for(int i=0; i< 5; i++)
{          myarray[i]=i*2+1;  }
```

Arrays and Functions (L#23)

- When passing an individual array element, treat the single array element like a simple variable!
  - Pass by values: pass the values of the element without having it changed in the function
  - Pass by reference: change the value of the array element in the function

- When passing the whole array to a function
  - In the calling function, use the array name as the input parameter passed to the called function
  - In the called function, specifically, the function header, and function declaration, declare the parameter as an array

Passing the Entire Array (Example)

```
#include "stdio.h “

void add(int arr[]);

void main(void)
{
    int myarray[5]= {1,2,9,3,6};
   add(myarray); /* Pass the whole array to a function */
    printf(“The value of myarray[2] is: %d\n",myarray[2]);
}

void add(int arr[])
{
    arr[2] = arr[2] + 100;
}
```

A variable with brackets [] in function prototype and header indicate the parameter is an array!

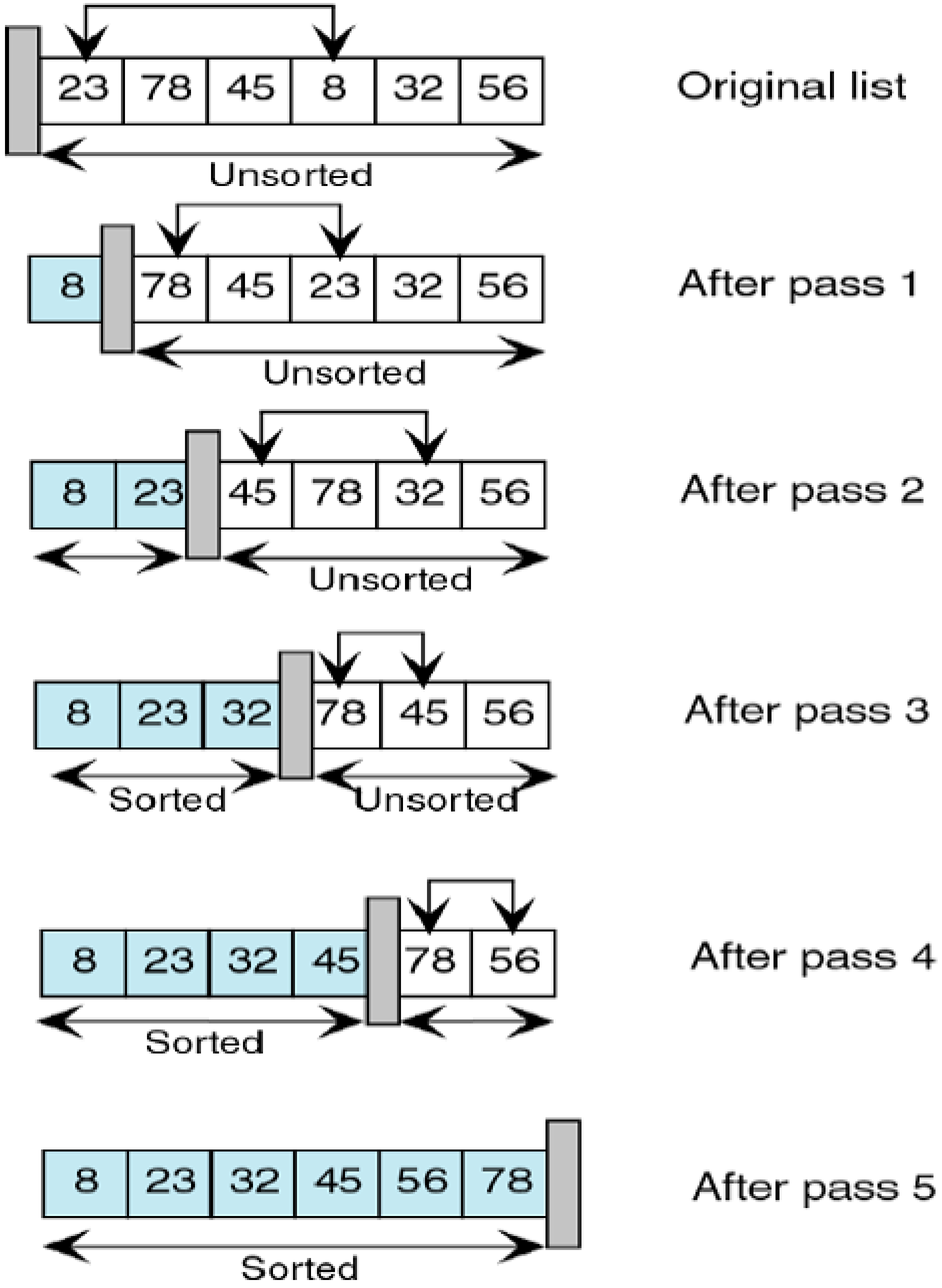
Bubble Sort (L#24)

Bubble sort works by repeatedly comparing adjacent elements and swapping adjacent elements that are out of order

Initial array	23	78	45	8	32	56
1st pass	8	23	78	45	32	56
2nd pass	8	23	32	78	45	56
3rd pass	8	23	32	45	78	56
4th pass	8	23	32	45	56	78
5th pass	8	23	32	45	56	78

Selection Sort (L#24)

Selection sort works by repeatedly selecting the smallest/largest remaining element



Strings (L#25)

- In C, a string is a variable-length array that is DELIMITED BY THE NULL CHARACTER (\0).
- Four ways to initialize a string

char month[10] = “March”;

M	a	r	c	h	\0	0	0	0	0

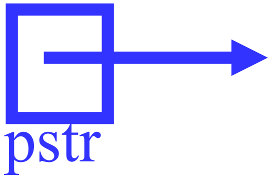
char month[] = “March”;

M	a	r	c	h	\0

char month[6] = {‘M’,‘a’, ‘r’,‘c’, ‘h’, ‘\0’};

M	a	r	c	h	\0

char *pstr=“March”;



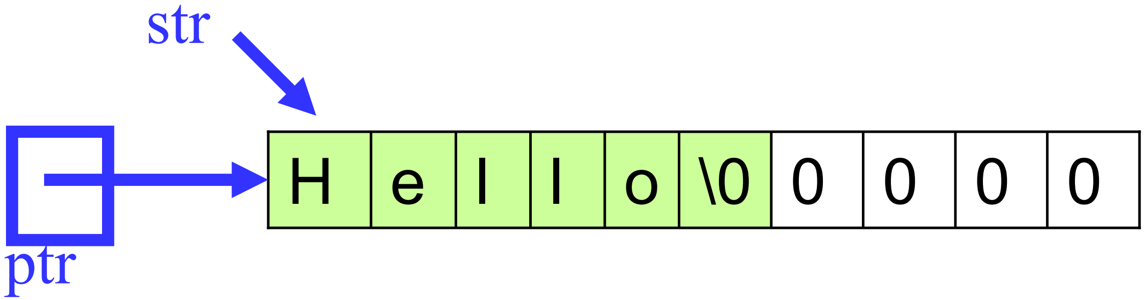
M	a	r	c	h	\0

Referencing String Literals

Array name indicates the address of the first element of the array

```
char str[10] = “Hello”;
char *ptr;
ptr=str;
```

String itself is a pointer to the first element/character of the string



H	e	l	l	o	\0	0	0	0	0

str[0]=ptr[0]=“Hello”[0]=‘H’
str[3]=ptr[3]=“Hello”[3]=‘l’

Pointers (L#26)

- A pointer variable can be declared using * in the declaration statement
- A pointer to (or the address of) a variable can be obtained using &
- Pointers provide us a way to work with addresses symbolically.

```
#include "stdio.h"
void main(void)
{
        int x=3;
        int *p= &x;
        printf("%d\n",x);
        printf("%d\n",*p);
        printf(“%d\n",p);
}
```

```
Output:
3
3
1244884
```

Ways to increment a number

- Assume

```
int a=0;
int *p=&a;
```

we need to add 1 to a:

```
a++;
++a;
a=a+1;
*p=*p+1;
(*p)++;
++(*p);
```

Passing an Array to a Function

- In the called function prototype and definition header
  - Way 1: use the traditional array notation to indicate that the parameter is an array:

    ```
    int my_func(int a[]);
    ```

  - Way 2: use pointers:

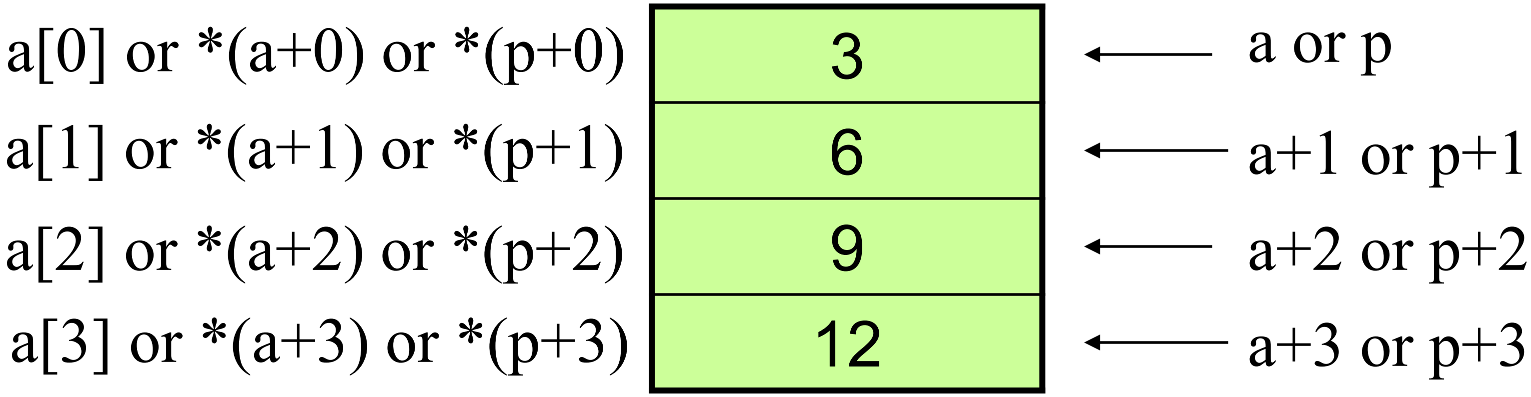
    ```
    int my_func(int *a);
    ```

- In the calling function, use the array name as the parameter in the function call

Arrays and Pointers (L#28)

- Arrays and pointers have a very close relationship
  - The array name is a pointer constant to the first element of the array
  - We can use array name anywhere we can use a pointer, specifically, with the indirection operator *

```
int a[4] = {1,10,30,4};
int *p = a;
```



a[0] or *(a+0) or *(p+0)

a[1] or *(a+1) or *(p+1)

a[2] or *(a+2) or *(p+2)

a[3] or *(a+3) or *(p+3)

Accessing the array elements

Array a

pointers

Pointer Compatibility

- Pointer types must match, otherwise, using a cast operator so that you can make an explicit assignment between incompatible pointer types!
- Void pointer is the only exception! It can be used with any pointer and any pointer can be assigned to a void pointer; however it cannot be dereferenced because a void pointer has no data type,

Illegal:

```
int* a;
char c = ‘A’;
a = &c;
```

Valid:

```
int* a;
char c = ‘A’;
a = (int*) &c;
```

Valid:

```
void* a;
char c = ‘A’;
a = &c;
```

Illegal:

```
void* a;
char c = ‘A’;
a = &c;
printf(“%c/n”, *a);
```

Final Exam

- Time: **8-11am, May 1, Monday**
- Please arrive at the class on time; no make up time will be given for late arrivals.
- Form:
  - Open book open notes
  - Calculators are NOT allowed
  - Visual Studio is NOT allowed
  - ChatGPT is NOT allowed

Good Luck to Your Finals!!!

- Preparation:
  - Exam#1, #2, #3
  - Lecture #29 (refer to more details in Lecture notes #2 - #28)
  - Lab #1 - #12
  - HW#1 - #4