C PROGRAMMING
Lecture 2

1st semester 2021-2022

Standard Input and Output

- Usually any program needs at least to print an output, some of them need also an input
- Formatted I/O: scanf, printf
- Character I/O: getchar, putchar
- Line I/O: gets, puts

Printf

- Sends output to *standard out*, the default output device can be seen as the terminal screen.
- General form

```
printf(format descriptor, var1, var2, …);
```

- format descriptor is composed of
  - Ordinary characters
    - copied directly to output
  - Special characters
    - Characters preceded by \
  - Conversion specifiers
    - Causes conversion and printing of next *argument* to printf
    - Each conversion specification begins with %

Printf

```
printf(“hello world\n”);
```

result: “print the string hello world followed by new line”, BUT!, according to general form, it should be

```
printf(“%s\n”, “hello world”);
```

result: “print hello world as a *string* followed by a newline character”

```
printf(“%d, %d\t%d\n”, var1, var2, var3);
```

result: “print the value of the variable var1 as an integerfollowed by “,” then the value of var2 as an integer followed by a tab followed by the value of the variable var3 as an integer followed by a new line.”

```
printf(“%f,  \b%f\n”, var1, var2);
```

Result:?

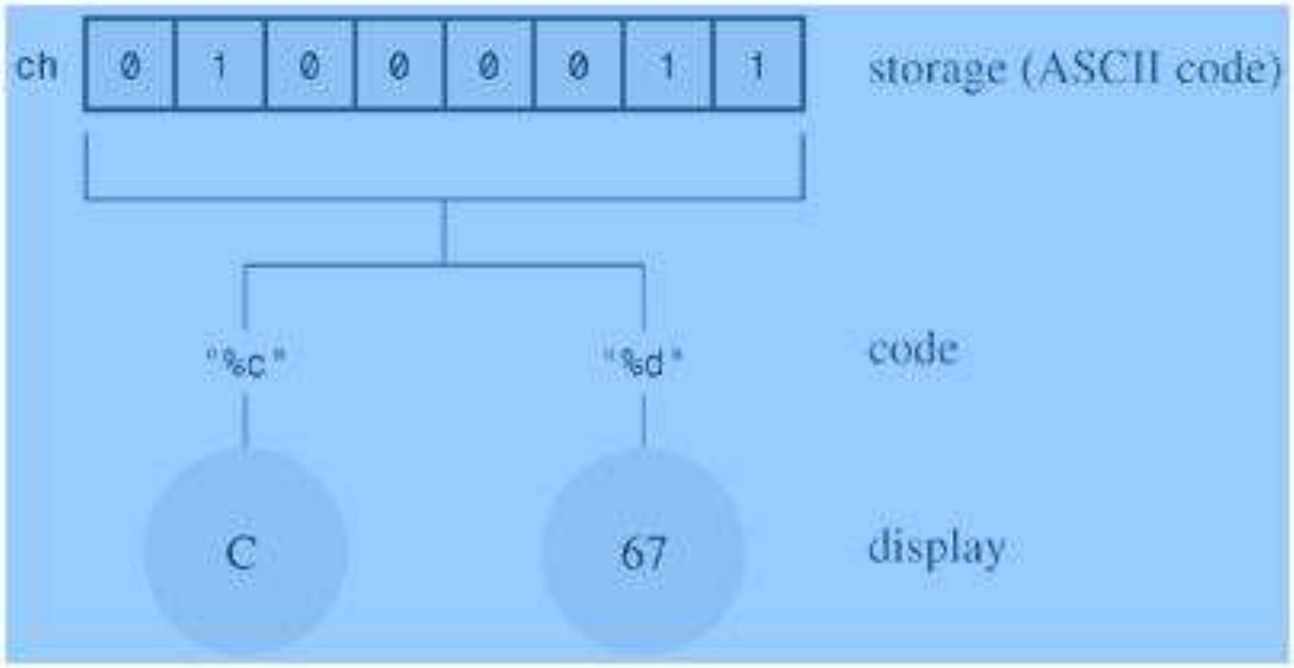
Printf

- The format specifier in its simplest form is one of:
  - %s
    - sequence of characters known as a **String** (an *array* of chars, will be studied later)
    - Not a fundamental data type in C
  - %d
    - Decimal integer (base ten)
  - %f
    - Floating point
  - %c
    - Character

There are others also (see Unix command “man 3 printf”)

Printf Char Type

- Storage vs display values



Printf

- Special characters
  - Not wysiwyg; all begin with “\” char
    - \n new line
    - \t horizontal tab
    - \v vertical tab
    - \b backspace
    - \r carriage return
    - \c produce no further output
    - \f form feed
    - %% a single %

Printf

- Alignment and width options:
  - A minus(-) sign tells left alignment.
  - A number after % specifies the minimum field width to be printed; if the characters are less than the size of width the remaining space is filled with space and if it is greater than it printed as it is without truncation.
  - A period(.) symbol separates field width with the precision.
  - Precision tells the minimum number of digits in integer, maximum number of characters in string and number of digits after decimal part in floating value.

Printf

flag	effect
none	print normally (right justify, space fill)
-	left justify
0	leading zero fill
+	print plus on positive numbers
	invisible plus sign

Printf

```
/* prints 100 in decimal, octal, and hex */
#include <stdio.h>
int main(void)
{
    int x = 100;
    printf("dec = %d; octal = %o; hex = %x\n",
  x, x, x);
    printf("dec = %d; octal = %#o; hex = %#x\n",
  x, x, x);
    return 0;
}
```

```
dec = 100; octal = 144; hex = 64
dec = 100; octal = 0144; hex = 0x64
```

Printf

```
//Fill spaces          ---
("%-3d",0)             0
("%-3d",-1)            -1
("%-3d",12345)         12345
("%-3d",-12345)        -12345

//Justify              ---
("%+3d",0)              +0
("%+3d",-1)             -1
("%+3d",12345)         +12345
("%+3d",-12345)        -12345
```

Printf

```
//Fill zeros          ---
("%03d",0)            000
("%03d",-1)           -01
("%03d",12345)        12345
("%03d",-12345)       -12345

//Invisible + Sign    ---
("% -3d",0)            0
("% -3d",-1)          -1
("% -3d",12345)        12345
("% -3d",-12345)      -12345
```

Printf

Let a floating point number be `nr=1.8765432`

```
// Precision digits
("%.0f",nr)          2
("%.0f.",nr)         2.
("%.1f",nr)          1.8
("%.6f",nr)          1.876543

// Width and precision -----
("%5.0f",nr)             2
("%5.0f.",nr)            2.
("%5.1f",nr)           1.8
("%5.6f",nr)       1.876543
```

Scanf

- reads user-typed input from *Standard input,* default input device can be seen the keyboard
- General form:

```
scanf( format descriptor, &var1, &var2, ...);
```

- format descriptor is the same as for printf
- Blocking statement, until receives input
- Note the “&” sign (will be covered later in more depth)

double var1; scanf(“%f”,&var1);

- !!! Discussion reading char

Reading/writing chars

- The pair of functions getchar/putchar used for keyboard input/console output
- Reads/writes single char from/to standard input/output
- getchar() blocks until data is entered
- If more than 1 char entered, only first 1 is read

```
int c;
c=getchar();
putchar(c);
```

Reading/writing lines

- The pair of functions gets/puts used for keyboard input/console output
- Reads/writes lines of chars from/to standard input/output
- The way to read strings with whitespaces

```
char var_s[250];
printf("Input a string: ");
gets(var_s);
printf("The input string is: ");
puts(var_s);
```

Statements

- Represent the flow of the program
- Basic statements
  - empty statement
  - expression statements
  - sequential statements
  - iterative statements
  - selection statements
  - jump statement
- Compound statements
  - combines basic statements

Empty Statement

- Statements that contain only “;” character
- Used where the syntax needs a statement but the program doesn’t have to do something

Expression Statement

- Composed by an expression followed by “;” character:

```
expression;
```

- Most often encountered statements
- Based on arithmetic/increment/decrement expressions, sometimes in conjunction with the assignment operator:

```
a=a+1;
b++;
```

Compound Statement

- Composed by grouping several statements and variable declarations
- Used where syntax requests one statement but the logic needs several actions;
- Grouping done by enclosing statements and declarations between {}:

```
{
  variables declarations;
  statements;
}
```

Flow Control Statements

- while
- if...else
- for
- do... while
- switch

While statement

- repeating a statement or group of statements until some specified condition is met
- General form:

```
while(expr){
   statement1;
   …
   statementn;
}
```

- If expr evaluates to true (different from 0), then execute body, else go to next statement after body
- Repeat until expr is false (equals 0)

While statement

```
/* factorial of n */
#include<stdio.h>
int main(){
   int n, nfact;
   printf(“Enter a number >0:  “);
   scanf(“%d”,&n);
   while(n>0){
        nfact *= n;
        n -= 1;
   }
   printf(“Value of factorial is: %d \n”,nfact);
   return 0;
}
```

If … else statement

- General form:

```
if(expr)
   statement1
else
   statement2
```

- Else part is optional, statement can be simple or compound
- Expression is evaluated. If true (not 0), statement1 is executed and statement2 skipped (if exists). If false (0), statement1 is skipped and if else exists statement2 is executed

If … else example

```
#include <stdio.h>
int main(){
   float n1,n2;
   printf(“Enter 2 numbers:”);
   scanf(“%f %f”, &n1, &n2);
   if(n2==0) /* careful, (n2=0) may not be false*/
      printf(“Divizion by 0\n”);
   else
      printf(“%6.2f divided by %6.2f is: %6.2f\n”,
  n1,n2,n1/n2);
   return 0;}
```

For statement

- Looping statement
- General form:

```
for(expr1 ; expr2 ; expr3)
        statement
```

- expr1 is called the initialization step; performed when for is to be executed
- expr2, the test/condition to control the execution of statement
- expr3, reinitialization step

For statement

- It is executed as follows:
  - expr1 is evaluated
  - expr2 is evaluated; if true, statement is executed, followed by expr3 and this step is repeated
  - if expr2 is evaluated to false, the next statement after if's statement is executed

equivalence:

```
expr1;
    while(expr2){
        statement;
        expr3; }
```

For statement

```
/* print a multiplication table */
#include<stdio.h>
int main(){
   int type, start, end, j;
   printf("Type of table?");
   scanf("%d",&type);
   printf("start of table?");
   scanf("%d",&start);
   printf("end of table?");
   scanf("%d",&end);
   for(j=start;j<=end;j++)
      printf("\n%2d x %2d = %3d", j, type, j *
  type);
   printf("\nEnd of program\n");
   return 0;
}
```

Do … while statement

- General form:

```
do
        statement
while(expr)
```

- statement is executed
- expr is then evaluated; if true, repeat the above; if false, the next statement after while is executed
- !!! statement is executed at least once

Do … while statement

```
/* computes the greatest common divisor */
#include<stdio.h>
int main(){
   int m, n, r;
   do{
     printf("\nEnter two positive integers:");
     scanf("%d %d",&m, &n);
   }while(m<=0||n<=0);
   do{
     r=m%n;
     m=n;
     n=r;
   }while(r>0);
   printf("result is %d\n",m);
   return 0;
}
```

Switch statement

- Multi-way branching
- General form:

```
switch (expr) statement
```

- expr evaluates to an int; statement is almost always compound statement
- Any statement within compound statement:

```
case <constant expr>:
```

- Constant expr is an int; no 2 constants can be the same; one statement can be labeled

```
default:
```

Switch statement

- Once the control is given to a given statement as its label matches the value of expr, all statements down to the end are executed unless break or return jumps out of the switch

```
switch(ch){ /* counts lowercase vowels and
  nonvowels
   case 'a':
   case 'e':
   case 'i':
   case 'o':
   case 'u': vowels ++;
                  break;
   default: nonvowels++;
                  break;  /* not needed, just for
  clarity */
}
```

Exercises

- Write a program that reads a positive integer and determines:
  - Whether is even or odd
  - Whether is prime or not
  - Whether is perfect square

Debugging with gdb

- allow you to see what is going on “inside” another program while it executes, or what another program was doing at the moment it crashed.
- Gdb can do 4 types of actions:
  - Start your program, specifying anything that might affect its behavior.
  - Make your program stop on specified conditions.
  - Examine what has happened, when your program has stopped.
  - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Gdb – program to debug

```
# include <stdio.h>
int main()
{
    int i, num, j;
    printf ("Enter the number: ");
    scanf ("%d", &num );

    for (i=1; i<num; i++)
       j=j*i;
    printf("The factorial of %d is %d\n",num,j);
}
```

Gdb

Examine how to debug in 6 simple steps

- Step 1: compile with debug option; This allows the compiler to collect the debugging information.

```
gcc -g factorial_err.c
```

- Step 2: launch gdb;

```
gdb a.out
```

- Step 3: setup a break point inside the program

`break line_number` (in our case line 10)

Gdb

- Step 4: execute program in debugger

```
run
```

  - program can be started using the run command in the gdb debugger
  - The program executed until first break point then gives you the prompt for debugging

- Step 5: print variable values

```
print i
print j
print num
```

Gdb

- Step 6: continue, step over or step in
  - `c or continue`: Debugger will continue executing until the next break point.
  - `n or next`: Debugger will execute the next line as single instruction.
  - `s or step`: Same as next, but does not treats function as a single instruction, instead goes into the function and executes it line by line.

Gdb

Once started, it reads commands from the terminal until you tell it to exit with the GDB command quit

- Some command shortcuts:
  - `l` – `list`
  - `p` – `print`
  - `c` – `continue`
  - `s` – `step`
  - `ENTER`: pressing enter key would execute the previously executed command again.
  - `help` – View help for a particular gdb topic — help topic.
  - `quit` – Exit from the gdb debugger.

User Defined Types

- Type definition that allows programmers to define an identifier/name that would represent an existing data type

general form: `typedef type identifier;`

- Example: `typedef int age;`
- For example, when we require to store people's age, we will create variable of type age like:
  age first_person;
- typedef just increase readability by creating meaningful data type names, it doesn't really create a new datatype.

Enumerated Data Type

- Suppose we want to store week days using their names: Monday, Tuesday, …, Saturday, Sunday
- It means we know the possible values in advance
- Can be done using enumerated data type

```
enum identifier
{value1,value2,..};
```

- Example: `enum day {Mon,Tue,Wed,Thur,Fri,Sat,Sun};`
  `enum day week_day;`

Here `week_day` can have any possible value from the list

Enumerated Data Type

- When using enumeration data type, enumeration constant Mon is assigned 0, Tue assigned 1, Wed is assigned 2, and so on.
- Automatic assignments can be overridden by:

```
enum day {Mon=10,Tue,Wed,…..};
```

- Now `Mon` is assigned value `10`, `Tue` will have next value: `11`, `Wed` will have `12`, and so on.
- Question:

```
int interval;  interval = Wed – Mon;
interval = ?
```

Variable Storage Classes

- Provides information about the location and visibility/scope of a variable.
- Storage class specifiers
  - Storage duration – how long an object exists in memory
  - Scope – where object can be referenced in program
  - Linkage – specifies the files in which an identifier is known
- There are four storage class specifiers `auto,register,static,extern.`

Automatic storage

- auto variables
- Defined within a function
- Storage (memory) allocated when the function is executing
- Storage released when the function returns
- Local to that function so can’t be used outside the function
- Gets garbage initialized
- auto: default for local variables

```
auto double nr1, nr2;
```

Register storage

- Usually applied to heavily used variables
- register: **tries** to put variable into high-speed registers
- Only int and char can be stored in registers
- Can only be used for automatic variables

```
register int counter = 1;
```

Static and Extern Storage

- Static storage
  - variables exist for entire program execution
  - default value of zero
  - static: local variables defined in functions.
    - Keep value after function ends
    - Only known in their own function
- Extern storage
  - default for global variables and functions
  - known in any function

Identifiers as Constants

- Suppose we want to make some identifier whose value need not to be changed during the execution

```
const int array_size = 10;
```

- You can use const qualifier. This ensures that

```
x= array_size;
array_size = 10;
```

valid

invalid, array_size is constant and can't change it's value.

Volatile Variables

- A variable may change value at any time by some external sources (source from outside the program).

```
volatile int calendar_date;
```

- The value of calendar_date may be changed by some external factors
- The compiler will examine the value of the variable each time it is encountered to see whether it is changed by an external source.

Symbolic Constants

- Suppose we want to use some constant value in many places in the program

```
value_of_pi   3.142
```

- If we write the constant value 3.14 at each places and at some point want to change to 3.1429 then the problem can be:

- do the replacement at each place
- harder understanding of the code; value is less meaningfull as a word

#define

#define is used to define symbolic constants in our program.

We face two problems if we are using some numbers in the program at many places.

1. Problem in modification of the program.
2. Problem in understanding of program.

By using the #define macro we can overcome these two problems.

#define

A constant can be defined as follows:

#define symbolic-name value_of_constant

Valid examples are:

#define PI 3.14159
#define MAX 100

#define

1.Symbolic name have the same form as variable names. But usually we are using CAPITALS for symbolic names. But that is just a convention.

```
#define MAX 200
```

2.No blank space between the pound sign'#' and the word define is permitted.

```
#   define MAX 200
```
<- Not permitted

3.'#' must be the first character in the line.

4.A blank space is required between #define and symbolic name and between the symbolic name and value.

5.#define statements must not end with a semicolon.

```
#define PI 3.1415;
```
<- Not valid

#define

6.After definition, the symbolic name should not be assigned any other value within the program using an assignment statement.

```
#define STRENGTH 100
main()
{
STRENGTH = 200;
}
```

Is illegal.

7.Symbolic names are NOT declared for data types. Its data type depends on the type of constant.

8.#define statements may appear anywhere in the program but before it is referenced in the program.

Arrays

A one dimensional array is a list of data values, with all values having the same data type (the base type), such as

- integer
- float
- double
- char

Technically, an array is a uniform data structure. Individual array elements are referenced using the array name and a subscript that identifies the element position in the array.

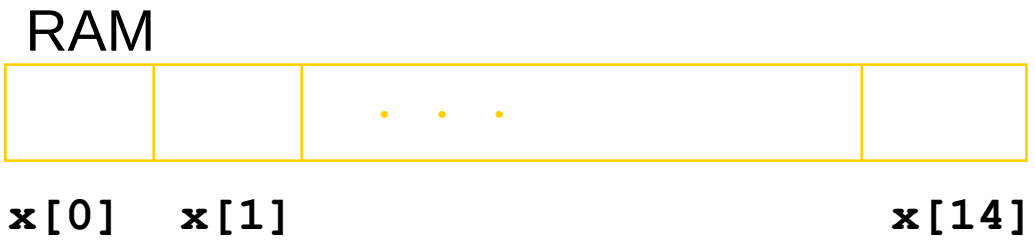
Array Declaration

For a one dimensional array, we specify the array name, its base data type, and the number of storage locations required using declarations such as

```
int x[15];
float y[100], z[100];
```

which specifies 15 integer locations for x and 100 floating-point locations for arrays y and z.

Storage for array elements are in contiguous locations in memory, referenced by subscript(or index) values starting at 0. Thus for array **x** above, the storage is



Array Declaration and Initialization

The array size could also be specified using a symbolic constant:

```
#define ARRAY_SIZE 15
int x[ARRAY_SIZE];
```

We can initialize array elements in declaration statements:

```
int integers[5] = {1, 2, 3, 4, 5};
int even_num[] = {2, 4, 6, 8}; /* has 4 elements */
```

We cannot specify more initial values than there are array elements, but we can specify fewer initial values, and the remaining elements will be initialized to 0.

Example:

```
int b[10] = {2};
double x[250] = {0};
```

Array Subscripting

Given that the array x is declared as:

```
int x[250];
```

To initialize a large array to a nonzero value - say 10, we can use a loop. e.g.,

```
for (k = 0; k <= 249; k = k+1)
 x[k] = 10;
```

k is a subscript

Note: when referencing a particular element of an array use square brackets, not parenthesis or curly braces.

Array Subscripting

A subscript value for an array element can be specified as any integer expression.

For example, given the following declarations:

```
double y[4] = {-1.0, 12.0, 3.5, 3.2e-2};
int c = 2;
```

the following statement would assign the value of 5.5 to y[1]

```
y[2 * c - 3] = 5.5;
```

Array - Example

- Given the following problem:
  Write a program that prompts the user for an integer n (n<=100). The program prints the average and the amount by which numbers differ from average.
- Guard against n<1or n>100, n undefined
- Cover the case when average is not an integer number

Example

```
/*  C program to print the average of n
  numbers and each difference to the average.
  */
#include <stdio.h>
int main()
{
  int number[100], count;
  float sum, average;
  int n=0; //we guard against undef value n
  do{
     printf("\nGive the value for n (1-100)");
 scanf("%d", &n);
  } while ((n<1)||(n>100));
```

Example

```
sum=0;
  for(count=0;count<n;count++){
     scanf("%d",&number[count]);
 sum+=number[count];
  }
  average=sum/n;
  printf("\nAverage of the %d numbers is
%7.2f\n",n, average);
  /* In order to print the difference between each
element and
  the average, we need to iterate through the array
*/
  for(count=0;count<n;count++)
     printf("For the element %d, the difference
between %d and average %7.2f is: %7.2f\n", count,
number[count], average, number[count]-average);
  return 0;
} /* end of main */
```

Strings

A string is an array of char
Each character in string occupies a location
'\0' is put at the end, after last character in array

Example: “C Programming”

Is stored as 14 positions array of char having on the last position '\0'

Strings

In order to read a string from input using scanf, “%s” has to be used as format specifier

Example:

```
char name[25];
scanf(“%s”, name);
/* note that we didn't use & */
```

Strings

Characters are stored starting at name[0] with the first non-whitespace character until the next whitespace character '\0' is added automatically at the end

`Scanf` can't read whitespace characters; use instead `gets`

Dedicated functions for string comparison and assignment:

`strcmp(string1, string2)` // returns negative, 0 or positive

`strcpy(string1, string2)` // string2 is copied in string1