C PROGRAMMING
Lecture 4

1st semester 2023-2024

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
- 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.

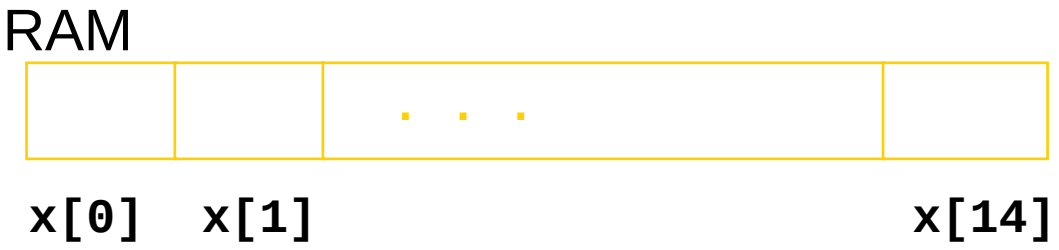
Array Declaration

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