WELCOME ALL

- This is Introduction to C course
- Tamanna Chhabra

WELCOME ALL

- PhD from Aalto University, Finland
- Research papers in various international forums
- 4 years of teaching experience at college and university level
- My email is tamanna.chhabra@senecacollege.ca

Expectations- Classroom Policy

- My role as a teacher
  - ❑Organized class sessions
  - ❑Post PPTs and other material on time
  - ❑Keep room for your input in class
  - ❑Acknowledge your questions
  - ❑Post grades on time
  - ❑Respond to emails in one business day
  - ❑Zero tolerance for favouritism
  - ❑Maintain positive learning environment

Expectations- Classroom Policy

- Your role as a student

❑Respectful behavior (Disrespectful behavior would be directed to Student's conduct office)

❑Professionalism – punctuality and participation is expected

❑No social media during class time, devices would be allowed for in class activities

Functions, Arrays and Structs

Function prototype

- In C all functions must be declared before they are used.
- This is accomplished using function prototype. Prototypes enable complier to provide stronger type checking.
- When prototype is used, the compiler can find and report any illegal type conversions between the type of arguments used to call a function and the type definition of its parameters.
- It can also find the difference between the no. of arguments used to call a function and the number of parameters in the function.

Function Prototype

- Thus function prototypes help us trap bugs before they occur.
- In addition, they help verify that your program is working correctly by not allowing functions to be called with mismatched arguments.

Function Prototype

- A general function prototype looks like following :

  return_type func_name(type param_name1, type param_name2, …,type param_nameN);

- The type indicates data type, parameter names are optional in prototype.

Example

```
void sqr_it(int  i);     //prototype of function sqr_it
int main()
{
   int num;
   num = 10;
   sqr_it(num);
   return 0;
}
void sqr_it(int i)
{
   i = i * i;
}
```

include directive

- A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files.
- There are two types of header files: the files that the programmer writes and the files that comes with your compiler.
- You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

include directive

- Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.
- A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

include

- The #include directive takes either of two forms

1. #include <*filename*> // *filename* is in the system directories
2. #include "*filename*" // *filename* is in the current directory

For #include "filename" the preprocessor searches in the same directory as the file containing the directive.

For #include <filename> the preprocessor searches in the system directory.

Need of #include

- You might be wondering why you need to #include files and why you would want to have multiple .c files for a program.

1. It speeds up compile time. As your program grows, so does your code, and if everything is in a single file, then everything must be fully recompiled every time you make any little change. This might not seem like a big deal for small programs (and it isn't), but when you have a reasonable size project, compile times can take *several minutes* to compile the entire program. Can you imagine having to wait that long between every minor change?

Need of # include

2. It keeps your code more organized. If you separate concepts into specific files, it's easier to find the code you are looking for when you want to make modifications (or just look at it to remember how to use it and/or how it works).

Stdio.h

- The header file that contains the prototypes for the **printf()** and **scanf()** functions is called **stdio.h** and is stored in a system directory.
- We include this header file in our source code whenever we call either of these functions:

```
#include <stdio.h>
int main(void)
{
        printf("This is C\n");
        return 0;
}
```

Scope of a variable

- The scope of a variable defines the sections of a program from where a variable can be accessed.
- The scope of variable depends upon it's place of declaration.

Scope

- There are four places where a variable can be declared in a C program:

1. Outside of all functions(including main). This type of variables are known as global variables.

2. Inside a function or inside a block of code. This type of variables are known as local variables.

3. Declared as the formal parameter in a function definition. This is similar to a local variable in scope of function's body and has a function scope.

4. Declared inside a block.

Local Variables

- Local variables in C are declared inside a function body or block of code(inside {} braces).
- Local variables can only be accessed by other statements or expressions in same scope.
- Local variables are not known outside of their function of declaration or block of code.
- Local variables gets created every time control enters a function or a block of code. When control returns from function or block of code, all local variables defined within that scope gets destroyed.
- By default, local variables contains garbage values unless we initialize it.

Example

```
#include <stdio.h>
int main ()
{       /* local variable declaration */
        int a, b;
        int c;
        a = 10;
        b = 20;
        c = a + b;
        printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
        return 0;
}
```

Global variables

- Global variables in C are declared outside of a function, normally it is declared before main function and after header files.
- Global variables can be accessed form anywhere in a program. The scope of the global variables are global throughout the program.
- Any function can access and modify value of global variables.
- Global variables retains their value throughout the execution of entire program.

Global variables

- By default, global variables are initialized by the system when you define. Global variables of data type int, float and double gets initialized by 0 and pointer variables gets initialized by NULL.
- We can use same name(identifiers) for local and global variables but inside a function value of local variable gets priority over global variable.

Example

```
#include <stdio.h>
/* global variable declaration */
int g;
int main ()
{ /* local variable declaration */
        int a, b;
        a = 10; b = 20;
        g = a + b;
        printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
        return 0;
}
```

Another Example

```
#include <stdio.h>
/* global variable declaration */
int g = 20;
int main ()
{
        /* local variable declaration */
        int g = 10;
        printf ("value of g = %d\n", g);
        return 0;
}
```

Function Scope

- A variable that we declare within a function header has function scope.
- We call such variables function parameters.
- The scope of the parameter extends from the function header to the closing brace of the function body.
- We say that the parameter goes out of scope at this closing brace.

Function Scope

- The compiler allocates memory for a function parameter when the function is called and releases that memory when the function return control to its caller.
- C compilers initialize the values of the function parameters to the values of the arguments in the function call.

Example

```
#include <stdio.h>
/* global variable declaration */
int a = 20;
int main ()
{
        /* local variable declaration in main function */
        a = 10; int b = 20; int c = 0;
        printf ("value of a in main() = %d\n", a);
        c = sum( a, b);
        printf ("value of c in main() = %d\n", c);
        return 0;
 }
```

Example

```
/* function to add two integers */
int sum(int a, int b)
{
        printf ("value of a in sum() = %d\n", a);
        printf ("value of b in sum() = %d\n", b);
        return a + b;
}
```

Example

When the code is compiled and executed, it produces the following result

value of a in main() = 10

value of a in sum() = 10

value of b in sum() = 20

value of c in main() = 30

Block Scope

- A Block is a set of statements enclosed within left and right braces ({ and } respectively).
- Blocks may be nested in C (a block may contain other blocks inside it).
- A variable declared in a block is accessible in the block and all inner blocks of that block, but not accessible outside the block.

Example

```
int main()
{
  {
     int x = 10, y  = 20;
     {
        // The outer block contains declaration of x and y, so
        // following statement is valid and prints 10 and 20
        printf("x = %d, y = %d\n", x, y);
        {
           // y is declared again, so outer block y is not accessible
           // in this block
           int y = 40;

           x++;  // Changes the outer block variable x to 11
           y++;  // Changes this block's variable y to 41

           printf("x = %d, y = %d\n", x, y);
        }

        // This statement accesses only outer block's variables
        printf("x = %d, y = %d\n", x, y);
     }
  }
  return 0;
}
```

Output

Output:

x = 10, y = 20

x = 11, y = 41

x = 11, y = 20

Another Example

```
int main()
{
  {
     int x = 10;
  }
  {
     printf("%d", x);
  }
  return 0;
}
```

Output

error: 'x' undeclared (first use in this function)

Passing arrays to functions in c

Array is collection of elements of similar data types .

Passing array to function :

Array can be passed to function by two ways :

- **Pass Entire array**
- **Pass Array element by element**

Pass Entire array

- Here entire array can be passed as a argument to function .
- Function gets complete access to the original array .
- While passing entire array Address of first element is passed to function , any changes made inside function , directly affects the Original value .
- Function Passing method : “Pass by Address“

Pass Array element by element

- Here individual elements are passed to function as argument.
- Duplicate carbon copy of Original variable is passed to function .
- So any changes made inside function does not affect the original value.
- Function doesn’t get complete access to the original array element.
- Function passing method is “Pass by Value“

Example

```
#include <stdio.h>
void display(int age)
{
        printf("%d", age);
}
int main()
{
int ageArray[] = { 2, 3, 4 };
display(ageArray[2]); //Passing array element ageArray[2] only.
return 0;
}
```

Passing entire array to function :

- Parameter Passing Scheme : Pass by Reference
- Pass name of array as function parameter.
- Name contains the base address i.e (Address of 0th element)
- Array values are updated in function.
- Values are reflected inside main function also.

Example

```
#include<stdio.h>
#include<conio.h>
//--------------------------------
void fun(int arr[])
{
int i;
for(i=0;i< 5;i++)
 arr[i] = arr[i] + 10;
}
//--------------------------------
void main()
{
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
 scanf("%d",&arr[i]);
```

Example

```
printf("\nPassing entire array .....");
fun(arr);  // Pass only name of array
for(i=0;i< 5;i++)
 printf("\nAfter Function call a[%d] : %d",i,arr[i]);
getch();
}
```

Output

Enter the array elements : 1 2 3 4 5

Passing entire array .....

After Function call a[0] : 11

After Function call a[1] : 12

After Function call a[2] : 13

After Function call a[3] : 14

After Function call a[4] : 15

Example

```
#include<stdio.h>
#include<conio.h>
void modify(int b[3]);
void main()
{
        int arr[3] = {1,2,3};
        modify(arr);
        for(i=0;i<3;i++)
                printf("%d",arr[i]);
        getch();
}
void modify(int a[3])
{
        int i;
        for(i=0;i<3;i++)
        a[i] = a[i]*a[i];
}
```

Output

Output :

- 1 4 9

Alternate Way of Writing Function Header

- void modify(int a[3])

OR

- void modify(int *a)

Contd…

- A function header that receives an array's address takes the form

type function_identifier(*type array_identifier*[], ...)

or

type function_identifier(*type *array_identifier*, ...)

The empty brackets following ***identifier*** in the first alternative tell the compiler that the parameter holds the address of a one-dimensional array.

Passing structures to functions

In C, structure can be passed to functions by two methods:

1. Passing by value (passing actual value as argument)

2. Passing by reference (passing address of an argument)

Passing structure by value

- A structure variable can be passed to the function as an argument as a normal variable.
- If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

Example: Pass by value

```
#include <stdio.h>
struct student
{
char name[50];
int roll;
};
void display(struct student stu); // function prototype should be below to the structure declaration otherwise compiler shows error
int main()
{
        struct student stud;
        printf("Enter student's name: ");
        scanf("%s", &stud.name);
```

Contd…

```
        printf("Enter roll number:");
        scanf("%d", &stud.roll);
        display(stud); // passing structure variable stud as argument
        return 0;
}
void display(struct student stu)
{
        printf("Output\nName: %s",stu.name);
        printf("\nRoll: %d",stu.roll);
}
```

Passing structure by reference

- The memory address of a structure variable is passed to function while passing it by reference.
- If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

Example

```
#include <stdio.h>
#include <string.h>
struct student
{
        int id;
        char name[20];
        float percentage;
};
void func(struct student *record);
 int main()
{
        struct student record;
        record.id=1;
        strcpy(record.name, "Robin");
        record.percentage = 86.5;
                func(&record);
        return 0;
}
void func(struct student *record)
{
        printf(" Id is: %d \n", record->id);
        printf(" Name is: %s \n", record->name);
        printf(" Percentage is: %f \n", record->percentage);
}
```

Output

Id is: 1

Name is: Robin

Percentage is: 86.500000

Example: Pass by reference

- C program to add two distances (feet-inch system) and display the result.

```
#include <stdio.h>
struct distance
{
int feet;
float inch;
};
void add(struct distance d1,struct distance d2, struct distance *d3);
int main()
{
        struct distance dist1, dist2, dist3;
```

Contd…

```
    printf("First distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);
    printf("Second distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist2.feet);
    printf("Enter inch: ");
    scanf("%f", &dist2.inch);
    add(dist1, dist2, &dist3); //passing structure variables dist1 and dist2 by value whereas passing structure variable dist3 by reference
    printf("\nSum of distances = %d\'-%.1f\"", dist3.feet, dist3.inch);
    return 0;
}
```

Contd…

```
void add(struct distance d1,struct distance d2, struct distance *d3)
{ //Adding distances d1 and d2 and storing it in d3
        d3->feet = d1.feet + d2.feet;
        d3->inch = d1.inch + d2.inch;
        if (d3->inch >= 12) { /* if inch is greater or equal to 12, converting it
        to feet. */
        d3->inch -= 12;
        ++d3->feet;
}
}
```

Pass by reference

- In this program, structure variables dist1 and dist2 are passed by value to the add function (because value of dist1 and dist2 does not need to be displayed in main function).
- But, dist3 is passed by reference ,i.e, address of dist3 (&dist3) is passed as an argument.
- Due to this, the structure pointer variable d3 inside the add function points to the address of dist3 from the calling main function. So, any change made to the d3 variable is seen in dist3 variable in main function.

Contd…

- As a result, the correct sum is displayed in the output.