CS 335 : Group 33

Milestone 1

Group Members:

- Bommareddy Indra Sena Reddy (190241)
- Suraj Prakash (190882)
- Rahul Rathod Kethavathu (190667)
- Jami Sai Chandra Prakash (190395)

Lexical Elements

Identifiers

These are sequences of characters used for naming variables, functions, new data types, and preprocessor macros. They contain letters, decimal digits, and the underscore character '_' in identifiers. The first character of an identifier cannot be a digit.

Keywords

They are special identifiers. You cannot use them for any other purpose.

List of keywords we are supporting by our compiler are:

```
auto, break, case, char, const, continue, default, do, double, else, enum
extern, float, for, goto, if, int, long, register, return, short, signed,
sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while
```

Constants

A constant is a literal numeric or character value, such as `5` or `'m'`.

1. **Integer Constants**

   An integer constant is a sequence of digits, with an optional prefix to denote a number base.

2. **Character Constants**

   A character constant is usually a single character enclosed within single quotation marks, such as `'Q'`. A character constant is of type `int` by default.

3. **Real Number Constants**

   A real number constant is a value that represents a fractional (floating point) number.

4. **String Constants**

   A string constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks. A string constant is of type "array of characters".

Operators

An operator is a special token that performs an operation, such as addition or subtraction, on either one, two, or three operands.

Separators

A separator separates tokens.

E.g. `( ) [ ] { } ; , . :`

White Space

White space is the collective term used for several characters: the space character, the tab character, the newline character, the vertical tab character, and the form-feed character. White space is ignored.

Data Types

Primitive Data Types

1. Integer Types

The integer data types range in size from 8 bits to 32 bits.

- `signed char`
- `unsigned char`
- `char`
- `short int`
- `unsigned short int`
- `int`
- `unsigned int`
- `long int`
- `unsigned long int`
- `long long int`
- `unsigned long long int`

2. Real Number Types

There are three data types that represent fractional numbers.

- `float`
- `double`
- `long double`

Enumerations

An enumeration is a custom data type used for storing constant integer values and referring to them by names. By default, these values are of type `signed int`.

1. Defining Enumerations

`enum` [name (optional)] `{`constant names, …`};`

- `enum fruit {grape, cherry, lemon, kiwi};`

  Here, `grape = 0`, `cherry = 1`, `lemon = 2`, and `kiwi = 4.`

- `enum more_fruit {banana = -17, apple, blueberry, mango};`

  Here, `banana = -17`, `apple = -16`, `blueberry = -15`, and `mango = -14`.

- `enum yet_more_fruit {kumquat, raspberry, peach, plum = peach + 2};`

  Here, `kumquat = 0`, `raspberry = 1`, `peach = 2`, and `plum = 4`.

2. Declaring Enumerations

Can be declared by two ways, when the enumeration is defined and afterward.

- `enum fruit {banana, apple, blueberry, mango} my_fruit;`

-
  ```
  enum fruit {banana, apple, blueberry, mango};
  enum fruit my_fruit;
  ```

Structures

A structure is a programmer-defined data type made up of variables of other data types.

1. Defining Structures

`struct` [name (optional)] `{`structure's members`};`

If you leave the name of the structure out, you can't refer to that structure data type later on.

-
  ```
  struct point
  {
     int x, y;
  };
  ```

Structures may contain instances of other structures, but not themselves.

2. Declaring Structure Variables

Can be declared both when initializing definition of structure and after the definition (if name is given).

a. Declaring Structure Variables at Definition

```
struct point
{
   int x, y;
```

```
} first_point, second_point;
```

b. Declaring Structure Variables After Definition

```
struct point
{
   int x, y;
};
struct point first_point, second_point;
```

c. Initializing struct members

i. Specify the values in a set of braces and separated by commas. Those values are assigned to the structure members in the same order that the members are declared in the structure in definition.

```
struct point
  {
    int x, y;
  };
struct point first_point = { 5, 10 };
```

ii. You can also initialize the structure variable's members when you declare the variable during the structure definition:

```
struct point
  {
    int x, y;
  } first_point = { 5, 10 };
```

iii. You can also initialize fewer than all of a structure variable's members:

```
struct pointy
  {
    int x, y;
    char *p;
  };
struct pointy first_pointy = { 5 };
```

3. Accessing structure members

Use member access operator (.)

You put the name of the structure variable on the left side of the operator, and the name of the member on the right side.

```
struct point
  {
    int x, y;
  };

struct point first_point;

first_point.x = 0;
first_point.y = 5;
```

For nested structures, can use multiple member access operators in a line.

```
struct rectangle
  {
    struct point top_left, bottom_right;
  };

struct rectangle my_rectangle;

my_rectangle.top_left.x = 0;
my_rectangle.top_left.y = 5;

my_rectangle.bottom_right.x = 10;
my_rectangle.bottom_right.y = 0;
```

Arrays

Used for consecutive storage of data in memory, and is indexed from 0 to n-1, where n-1 is the size of the array.

1. Declaring arrays

Specify the variable type first, then the array name followed by the number of consecutive such variables to be stored in square brackets without any space.
Example : `int arr[10];`

Number of elements in the array can also be a variable; in that case, we just mention the variable name inside square brackets.
Example :

```
int n = 5;
int arr[n];
```

2. Initializing arrays

Here are some ways in which initialization of arrays is supported.

a) Can define and initialize the values at the same time.
   `int arr[5] = {0, 1, 2, 3, 4};`
b) If we don't initialize all the values, then the remaining values will be initialized to zero by default.
   `int arr[5] = {0, 1, 2};` is equivalent to `int arr[5] = {0, 1, 2, 0, 0};`

c) If you initialize every element of an array, then you do not have to specify its size; its size is determined by the number of elements you initialize.

```
int arr[] = {0, 1, 2, 3, 4};
```

3. Accessing arrays

To access any element in an array, specify the name of the array followed by the index number of the element inside square braces without spacing.

```
arr[0] = 2;
```
/ this initializes the first element of array with 2/

4. Arrays of structures

You can create an array of a structure type just as you can an array of a primitive data type.

```
struct point
  {
    int x, y;
  };
struct point coordinates[3];
```

That example creates a 3-element array of struct point variables called point_array. You can also initialize the elements of a structure array:

```
struct point coordinates[3] = {{2, 3}, {4, 5}, {6, 7}};
```

Pointers

Pointers hold memory addresses of stored constants or variables.

1. Declaring Pointers

To declare a pointer, data type must be mentioned along with the pointer name, separated by a space and an asterisk (*).

```
int *ptr;
```

2. Initializing Pointers

We need to declare the pointer first and then initialize it with an address of existing variable.

```
int n;
int *ptr = &n;
```

Here, the **&** operator is used to refer to the address of the variable.
The following piece of code shows the valid ways of using pointers.

```
int i, j;
int *ptr = &i;       /* ‘ip’ now holds the address of ‘i’. */
```

```
ptr = &j;            /* ‘ip’ now holds the address of ‘j’. */
*ptr = &i;           /* ‘j’ now holds the address of ‘i’. */
```

3. Pointers to structures

You can create a pointer to a structure type just as you can a pointer to a primitive data type.

```
struct point
  {
    float x, y;
  };
struct point p1 = {10.5, 13.2};
struct point *target = &p1;
```

That example creates a new structure type, struct point, and declares (and initializes) a variable of that type named target. Finally, it declares a pointer to the type struct point, and gives it the address of p1.

Access the variables inside the struct to which our pointer points to using the following syntax.

```
target->x = 10.5;
target->y = 13.2;
```

Type Qualifiers

Constant variables can only be read, not written during execution. Volatile variables can be both read and written during execution.

Constant variables are declared using an additional keyword ‘const’. If no such keyword is used, then the variable is considered to be a volatile variable by default.

```
const float pi = 3.14;
```

Typedef

Typedef statement is used to rename a specific portion of the code and give it another name. Using that name will imply that the original piece of code is substituted with that name.

Expressions and Operators

Expressions

An *expression* consists of at least one operand and zero or more operators.

Operands

Operands are typed objects such as constants, variables, and function calls that return values. Here are some examples:

10
sine(3.14)

Operators

An *operator* specifies an operation to be performed on its operand(s). Operators may have one, two, or three operands, depending on the operator.

Using Parentheses to assign priority to operations

Innermost parentheses have the highest priority and the expression inside the innermost parentheses is dealt with first. When only one operand is within the parenthesis, it can be omitted. Then the priority goes to the next innermost parentheses, and so on.

(2 * ((3 + 10) - (2 * 6))) = (2* (13 - 12)) = 2*1 = 2

Assignment Operators

Assignment operators are used to store values in variables. Here is a list of assignment operators supported by the compiler. Binary operators have one operand to the left and one to the right.

Let the left operand be ‘**x**’ and the right operand be ‘**y**’.

- x += y is equivalent to x = x + y
- x -= y is equivalent to x = x - y
- x *= y is equivalent to x = x * y
- x /= y is equivalent to x = x / y
- x %= y is equivalent to x = x % y
- x <<= y is equivalent to x = x << y

- x >>= y is equivalent to x = x >> y
- x &= y is equivalent to x = x & y
- x ^= y is equivalent to x = x ^ y
- x |= y is equivalent to x = x | y

Increment and Decrement Operators

Increment operator adds 1 to the operand, whereas decrement operator subtracts 1 from the operand.
Note that the operand must be either a variable of one of the primitive data types, a pointer, or an enumeration variable.
Prefix increment operator adds 1 before the operand is evaluated. Suffix increment operator adds 1 after the operand is evaluated. Same logic applies to the prefix and suffix decrement operators.
Here are a few examples of usage of these operators.

```
char w = 'a';
int x = 5;
float z = 1.3;
int *p = &x;

++w;   /* w is now the character ‘b’. */
x++;   /* x is now 6. */
z++;   /* z is now 2.3. */
++p;   /* p is now &x + sizeof(int). */
```

Arithmetic Operators

Here are the examples of the arithmetic operations and their usage.

Addition

```
int x = 2 + 3;              /* int x = 5 */
float y = 2.5 + 1.5;        /* float y = 4.0 */
```

Subtraction

```
int x = 5 - 4;              /* int x = 1 */
float y = 5.1 - 3.9;        /* float y = 1.2 */
```

Multiplication

```
int x = 2 * 3;              /* int x = 6 */
float y = 2.0 * 1.5;        /* float y = 3.0 */
```

Integer division

```
int x = 7 / 3;              /* int x = 2 */
```

Float division

```
float y = 3.6 / 3.0;        /* float y = 1.2 */
```

Modular division

```
int x = 7 % 3;              /* int x = 1 */
```

Negation

```
int x = -3;
float y = -3.6;
```

Note : Trivially, you can also apply a positive operator to a numeric expression:

```
    int x = +42;
```

Comparison Operators

When you use any of the comparison operators, the result is either 1 or 0, meaning true or false, respectively.

The equal-to operator == tests its two operands for equality.
The not-equal-to operator != tests its two operands for inequality
Less than operator < tests if left side operand is less than right side operand
Less than or equal to <= tests if left side operand is less than or equal right side operand
Greater than operator > tests if left side operand is greater than right side operand
Greater than or equal to >= tests if left side operand is greater than or equal right side operand

Logical Operators

Logical operators evaluate the truth value of a pair of operands. The && operator tests if both the expressions are true, if the first is false then the second is not evaluated. The || operator checks if either one of the two expressions is true. If the first expression is true, then the second expression is not evaluated. The ! operator can be prepended to an expression to negate the expression.

Bitwise Logical Operators

`&` for bitwise AND
`|` for bitwise OR
`^` for bitwise XOR
`~` for bitwise NEGATION

Pointer Operators

You can use the address operator & to obtain the memory address of an object.

```
int x = 5;
int *pointer_to_x = &x;
```

Sizeof Operator

This is used to obtain the size of data type of its operand. The operand may be an actual type specifier (such as int or float), as well as any valid expression. When the operand is a type name, it must be enclosed in parentheses

Type Casts

A type cast consists of a type specifier enclosed in parentheses, followed by an expression

Array Subscripts

You can access array elements by specifying the name of the array, and the array subscript (or index, or element number) enclosed in brackets.
Ex: `my_array[4]=5;`

Conditional Expressions

Syntax is

```
a ? b : c;
```

Here if expression a is true b is evaluated else c is evaluated.

Operator Precedence

Same as that of GCC compiler.

Statements

Labels

Labels can be used to identify different sections of the source code which can later be used with `goto` statements. The label consists of an identifier along with a colon. Example of label is:

```
printf("Hello World");
end_of_the_program:
```

Expression Statements

Any expression can be turned into a statement by adding a semicolon, due to which they get evaluated. Examples are:

```
    x++;
    y=x+25
```

`if` Statement

`if` statements can be used when the execution of certain statements is based on the result of an expression. The general syntax is

```
    if (test){
        statement1;
    }
    else {
        statement2;
    }
```

In the given syntax if the *test* evaluates to true then *statement1* gets executed and the rest is ignored, however if condition evaluates to false the *statement2* gets executed and the rest is ignored. `else` statement is optional and it can also be written as:

```
    if (test){
        statement1;
    }
```

Example of `if` statement:

```
    if (x > 0) {
        printf("x is positive");
    }
    else {
        printf("x is negative");
```

```
      }
```

Multiple if statements can be used for testing multiple conditions. Example

```
      if (x > 0) {
                printf("x is positive");}
      else if (x == 0) {
                printf("x is zero");
      }
      else if (x < 0) {
           printf("x is negative");
      }
      else {
           printf("Hello World");
      }
```

`switch` Statement

`Switch` statement compares the given expression with others, and then executes the statements according to the result of the comparison. The general syntax is:

```
      switch (test){
           case condition1:
                statement1;break;
           case condition2:
                statement2;break;
           ....
           default:
                statementn;break;
```

The `switch` statement compares *expression* with *condition1, condition2 … so on* until it finds the *condition* which matches the *test* expression. Then the statements corresponding to the case will get executed. The *default* case in the end is optional, which gets executed when the test does not match any of the conditions prior to the *default* case. Here, the `break` statement is also optional, but if not present at the end of each case it will not only execute the statements of the matching *case* but also the statements coming after them.

Example of `switch` statement:

```
      switch (2){
           case 0:
                printf("x is 0");break;
           case 1:
                printf("x is 1");break;
           case 2:
                printf("x is 2");break;
           case 3:
                printf("x is 3");break;
```

```
          default:
               printf("x is an integer");break;
```
The output of the example will be
```
     x is 2
```
But in the case `break` statement was not used at the end of every case the output would be:
```
     x is 2
     x is 3
     x is integer
```

`while` Statement

The while statement is a looping statement with an exit condition at the start of the loop. The general syntax is:
```
     while (test) {
          statement
     }
```
The `while` statement first evaluates the *test* expression, if true then the *statement* gets executed, then the *test* expression is evaluated once again repeating the process. *statement* gets executed again and again until the *test* evaluates to true after each execution. Example of `while` statement:
```
     int counter = 0;
     while (counter  <  5) {
          printf("%d", counter++);
     }
```

`do` Statement

The do statement is a loop statement similar to the while statement but the exit condition is at the end of the loop. The general syntax is:
```
     do {
          statement;
     }while(test);
```
The `do` statement first executes the *statement*, then evaluates the *test* expression if true then executes the *statement* again, then again the *test* expression is evaluated repeating the process. *statement* gets executed again and again until the *test* evaluates to true after each execution. Example of `do` statement:
```
     int counter = 0;
     do {
          printf("%d", counter++);
     }while(counter  <  5)
```

`for` Statement

For statement is a loop statement whose syntax contains variable initialization, exit condition and variable modification. The general syntax is:

```
    for (initialize; test; step){
        statement;
    }
```

The `for` statement first evaluates the `initialize` expression, then it evaluates the *test* expression, if the result is false the loop breaks but if it is true then the statement gets executed. In the end, *step* is executed and the next iteration of the loop begins by evaluating the *test* expression. The process continues until the test is evaluated false. Example of for statement is:

```
    for( x=0; x < 10; x++){
        printf(“%d”, x);
    }
```

Multiple test conditions can be put by using && or || operator.

Blocks

A `block` is a set of zero or more statements enclosed in braces. Often, a block is used as the body of an `if` statement or a loop statement, to group statements together.

You can declare variables inside a block such variables are local to that block.

Null Statement

It’s just a semicolon alone.

```
    ;
```

`goto` Statement

`goto` statements are used to perform unconditional jumps to a different place in a program. The general syntax is:

```
    goto label;
```

When the `goto` statement is executed the program control jumps to the specified label.

Example of goto statement:

```
goto end_of_the_program;
....
end_of_the_program
```

The `goto` statement can jump to any label as long as it is in the same function.

`break` Statement

You can use the `break` statement to terminate a `while, do, for, or switch` statement.

If you put a `break` statement inside of a loop or `switch` statement which itself is inside of a loop or switch statement, the break only terminates the innermost loop or switch statement.

`continue` Statement

You can use the `continue` statement in loops to terminate an iteration of the loop and begin the next iteration.

If you put a `continue` statement inside a loop which itself is inside a loop, then it affects only the innermost loop.

`return` Statement

You can use the return statement to end the execution of a function and return program control to the function that called it.
Here is the general form of the `return` statement:

```
      return return-value;
```

If the return type is void then `return;` will be ok.
If the function's return type is not the same as the type of return-value, and automatic type conversion cannot be performed, then returning return-value is invalid.

`typedef` Statement

You can use the typedef statement to create new names for data types.

General form,

```
      typedef old-type-name new-type-name
```

`old-type-name` is the existing name for the type, and may consist of more than one token (e.g., `unsigned long int`).

`new-type-name` is the resulting new name for the type, and must be a single identifier.
Creating this new name for the type does not cause the old name to cease to exist.

You can use `typedef` to make a new name for the structure while defining the type:

```
      typedef struct fish
      {
        float weight;
```

```
    float length;
    float probability_of_being_caught;
  } fish_type;
```

Functions

Function Declarations

General form of a function is,

```
return-type function-name ( parameters-list);
```

`return-type` Indicates the data type of the value returned by the function. You can declare a function that doesn’t return anything by using the return type `void`.

`function-name` can be any valid identifier.

`parameters-list` can consist of zero or more parameters separated by commas. A typical parameter consists of a data type and an optional name for the parameter

Example of a function declaration with 2 parameters:

```
int foo(int, char);
```

If we wish to include the name of the parameter, it will be followed immediately by the data type. An example is,

```
int foo(int x, char ch);
```

Parameter names can be any identifier. Two parameters cannot take the same name within a single declaration. The parameter names in declaration need not match the names in definition. You should write the function declaration above the first use of the function

Function Definitions

This specifies what a function has to do. A function definition consists of information regarding the function’s name, return type, and types and names of parameters, along with the body of the function. The function body is a series of statements enclosed in braces, in fact it is simply a block.

General form of a function definition:

```
return-type function-name (parameter-list)
{
    function-body
}
```

`return-type` and `function-name` are the same as what you use in the function declaration

`parameter-list` is the same as the parameter list used in the function declaration except you ***must*** include names for the parameters in a function definition.

A simple example of function definition that takes two integers as its parameters and returns the sum of them as its return value,

```
int add_values ( int x, int y )
{
      return x + y;
}
```

If you wish to write just the function definition without declaring then it has to be written above the first use of the function. But we strongly recommend first declaring and then defining it.

Calling Functions

You can call a function by using its name and supplying any needed parameters separated by commas. Here is the general form of a function call:

```
function-name (parameters)
```

Example of function call used as statement,

```
foo(5);
```

Example of function call used as subexpression,

```
a = add(5,10);
```

Function Parameters

Function parameters can be any expression, a literal value, a value stored in variables, an address of memory or combination of these.

Within the function body parameter is a local copy of the value passed into the function; you cannot change the value passed in by changing the local copy.

Example of calling a function with variable names as parameters,

```
int a=5,b=4,c;
int add(int x, int y);
…
```

```
    int add(int x, int y)
    {
          return x + y;
    }
    …
    c = add( a , b );
```
Example of calling a function with a pointer parameter,
```
    void foo(int *x)
    {
       *x = *x + 42;
    }
    …
    int a=15;
    foo(&a);
```

The main Function

Every program requires at least one function, called ‘main’. This is where the program begins executing. Return type of main is always `int`.

Reaching the } at the end of main without a return, or executing a return statement with no value (that is, `return;`) are both equivalent and the effect is equivalent to `return 0;`

To accept command line parameters, you need to have two parameters in the main function, `int argc` followed by `char *argv[]`. You can change the names of those parameters, but they must have those data types—`int` and array of pointers to `char`. `argv[0]`, the first element in the array is the name of the program as typed at the command line.

Recursive Functions

A function that calls itself is a recursive.
Example,

```
int factorial (int x)
{
  if (x < 1)
    return 1;
  else
    return (x * factorial (x - 1));
}
```

Static Functions

You can define a function to be static if you want it to be callable only within the source file where it is defined.

```
static int foo (int x)
{
  return x + 42;
}
```

Scope

It declares what parts of the code can see a declared variable. A scope usually confines the visibility of a variable to a particular function, a particular file or maybe accessible through all the files by using extern declaration.

Unless explicitly stated otherwise, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, including from within functions, but are not visible outside of the file.

Declarations made within functions are visible only within those functions.

A declaration is not visible to declarations that came before it.