**C Training**

**Data Types**

A screenshot of a computer

Description automatically generated

A screenshot of a white table with numbers and text

Description automatically generated

A white grid with black numbers

Description automatically generated

**Types of C Literals**

In C, Literals are the Constant values that are assigned to the constant variables.

There are 4 types of literals in C:

* **Integer Literal: decimals, octal-literals (045), hex-literals(0x23A), binary-literals(0b101)**
* **Float Literal**
* **Character Literal**
* **String Literal (**char stringVal[] = "GeeksforGeeks"**)**

# **Escape Sequence in C**

The escape sequence in C is the characters or the sequence of characters that can be used inside the string literal.

A screenshot of a phone

Description automatically generatedA screenshot of a computer

Description automatically generated

## Boolean in C

In C, the bool data type is not a built-in data type. It can be implemented in C using different methods as mentioned below:

1. Using header file “stdbool.h”
2. Using Enumeration type
3. Using define to declare boolean values

Boolean in C has the size of 1 byte as it needs only two values 0 and 1.

# **Integer Promotions in C**

Some data types like *char*, *short int*take a smaller number of bytes than *int*, these data types are automatically promoted to *int*or *unsigned int* when an operation is performed on them. This is called **integer promotion**. For example, no arithmetic calculation happens on smaller types like *char*, *short*and *enum*. They are first converted to *int*or *unsigned int*, and then arithmetic is done on them. If an *int*can represent all values of the original type, the value is converted to an *int*. Otherwise, it is converted to an *unsigned int.*

# **Character Arithmetic in C**

Character arithmetic is used to implement arithmetic operations like addition, subtraction, multiplication, and division on characters in C language. In character arithmetic character converts into an integer value to perform the task. For this ASCII value is used. It is used to perform actions on the strings.

# **Type Conversion in C**

## 1. Implicit Type Conversion

**-** Done by the compiler on its own, without any external trigger from the user.

**-** Generally, takes place when in an expression more than one data type is present. In such conditions type conversion (type promotion) takes place to avoid loss of data.

- bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

## 2. Explicit Type Conversion

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

# **Basic Input and Output in C scanf(). printf()**

### **printf**("\n\nEnter the Sentence: "); **scanf**("%[^\n]\ns", str); // input with spaces

C language does not provide a format specifier for binary numbers.

## List of Format Specifiers in C

A screenshot of a document

Description automatically generatedA screenshot of a computer

Description automatically generated

A white background with black text

Description automatically generated

# **Operators in C**

A diagram of numbers and symbols

Description automatically generated

## ****Bitwise Operators in C****

A screenshot of a computer

Description automatically generatedA white grid with black text

Description automatically generated

**Logical operators**

*The precedence of logical operators is: NOT, AND, OR.*

# **Increment and Decrement Operators in C**

### **Pre-Increment**

result = ++var1 is the same as var = var + 1; result = var

### **Post-Increment**

result = var1++ is the same as result = var; var = var + 1

## Syntax of Conditional/Ternary Operator in C

## The ternary operator in C is a conditional operator that works on three operands. It works similarly to the if-else statement and executes the code based on the specified condition. It is also called conditional Operator

*variable = Expression1* **?** *Expression2* **:** *Expression3****;***

*(condition)* **?** *(variable = Expression2)* **:** *(variable = Expression3)*;

# **sizeof operator in C**

sizeof() is a compile-time operator

**To find out the number of elements in an array:**

#include <stdio.h>

int main()

{

int arr[] = { 1, 2, 3, 4, 7, 98, 0, 12, 35, 99, 14 };

printf("Number of elements:%lu ",

sizeof(arr) / sizeof(arr[0]));

return 0;

}

**Data structures**

# **C Arrays**

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. The C arrays are static in nature, i.e., they are allocated memory at the compile time.

Array declaration: data\_type array\_name [size];

Array initialization: data\_type array\_name [size] = {value1, value2, ... valueN};

### **Array of Characters (Strings)**

In C, we store the words, i.e., a sequence of characters in the form of an array of characters terminated by a NULL character. These are called strings in C language.

**int** main()

{

    // creating array of character

**char** arr[6] = { 'G', 'e', 'e', 'k', 's', '\0' };

    // printing string

**int** i = 0;

**while** (arr[i]) {

**printf**("%c", arr[i++]);

    }

**return** 0;

}

## Properties of Arrays in C

### 1. Fixed Size

### 2. Homogeneous Collection

We can only store one type of element in an array. There is no restriction on the number of elements but the type of all of these elements must be the same.

### 3. Indexing in Array

The array index always starts with 0 in C language. It means that the index of the first element of the array will be 0 and the last element will be N – 1.

### 4. Dimensions of an Array

A dimension of an array is the number of indexes required to refer to an element in the array. It is the number of directions in which you can grow the array size.

### 5. Contiguous Storage

All the elements in the array are stored continuously one after another in the memory. It is one of the defining properties of the array in C which is also the reason why random access is possible in the array.

### 6. Random Access

The array in C provides random access to its element i.e we can get to a random element at any index of the array in constant time complexity just by using its index number.

### 7. No Index Out of Bounds Checking

There is no index out-of-bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

# **Pass Array to Functions in C**

*Arrays in C are always passed to the function as pointers pointing to the first element of the array.*

# **Bitmasking In C**

A bit is the smallest unit of data which can either store a 0 or 1 inside it

1. Setting a Bit

number **|** (1 **<<** bit\_position\_to\_set)

1. Clearing a Bit

number **& ~**(1 **<<** bit\_position\_to\_clear)

1. Flipping a bit

number ^ (1 << bit\_position\_to\_flip)

**Bitwise operators**

1. The **& (bitwise AND)** in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. The **| (bitwise OR)** in C takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. The **^ (bitwise XOR)** in C takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
4. The **<< (left shift)** in C takes two numbers, the left shifts the bits of the first operand, and the second operand decides the number of places to shift.
5. The **>> (right shift)** in C takes two numbers, right shifts the bits of the first operand, and the second operand decides the number of places to shift.
6. The **~ (bitwise NOT)** in C takes one number and inverts all bits of it.

**Stack memory**

* Stack memory could be considered the “default” memory as in most of the time you will be using the stack for your variables
* A stack is a linear data structure – with faster access than heap memory
* The amount of memory for a variable cannot be resized and is fixed from compile time
* Any allocation/deallocation is done for you, based on variable scope
* Once scope is lost, the variable name and its space are gone
  + this happens implicitly – the user has no control over it
* All of the variables inside the function following will be stored on the stack:

Int main (void) {

int i32x;

unsigned char u8y = 245;

short i16arr[5] = {0};

char stackString[] = “On the stack”;

return 0;

}

**Heap memory**

* Heap memory is useful when you want to change the size of your memory dynamically or when you want to assign large amounts of memory
* Heap Space may not be used as efficiently as the stack. Memory can become fragmented as blocks of memory first allocated and then freed.
* Variables can be resized as and when requested and size can be set dynamically at runtime
* Any allocation/deallocation must be explicitly performed
  + If memory is allocated but freed, it will continue to exist, resulting in memory leaks
  + Important in embedded use case to be aware of any potential memory leaks, not always as easy as turning a system off/on again to resolve memory issues
* Memory management is difficult, nobody is a perfect programmer!
* The malloc family of functions is used to allocate memory on the heap (though other functions may indirectly call malloc)
  + E.g. int \* ptr = malloc(sizeof(int))

**Uninitialized data segment (.bss)**

* This is where global or static memory that has not yet been assigned a value is stored.
* The value of these will always be 0 so you can save space in the compiled binary as the value of these variables doesn’t need to be stored (global, static inside a function)
* The following will both be stored in the bss:

unsigned long gu64y;

int main(void) {

static int i32x;

return 0;

}

**Initialized data segment (.data)**

* This is where global or static variables sit once they have been initialised with a value
* Any data here will take up space in the compiled binary unlike if initialised to the bss
* All of the following would be found here:

unsigned long gu64y = 3988L;

static int i32z = 900;

int main(void) {

static int i32x = 45;

return 0;

}

**Code/text segment (.text)**

* This is where the instructions for the program sit hence the name.
* However, in order to save space C also uses this space to save the values of string literals
* This means that if you take the address of a string in C it will be pointing to the relevant place in the code.
* Important: This space is READ ONLY, meaning that if your string is here, you cannot modify it.

**Pointers in C**

A pointer is simply a variable that contains a memory location.

* Variables are allocated at *addresses* in computer
* Name of the variable is a reference to that memory address
* A pointer variable contains a representation of an address of another
* The address (&) operator can be used in front of any variable to get the location in memory of the variable
* The indirection (\*) operator can be used to refer to the *contents* of the variable that the pointer points to