BCA (Honours)

Fundamentals of Programming Using C

Module 2

C Character Set

In C, a character set refers to the valid characters recognized by the compiler. These are the building blocks for writing programs.

- **Letters**: Uppercase (A-Z) and lowercase (a-z)
- **Digits**: Numbers (0-9)
- **Special Characters**: Include symbols such as `+ - * / = % ! # & ( ) { } ; : " ' < > [ ] , . ? | ^ ~ \`
- **Whitespace**: Blank spaces, tabs (`\t`), newlines (`\n`)

Delimiters in C

Delimiters in C are symbols that help separate different parts of the code. Common delimiters include:

- **Semicolon (;)**: Ends statements
- **Comma (,)**: Separates elements in a list
- **Braces ({})**: Define blocks of code (e.g., functions, loops)
- **Parentheses (())**: Used in functions and control statements
- **Square Brackets ([])**: Used in array declarations and access

Tokens in C

Tokens are the smallest units in a C program and can be classified into several categories:

1. **Keywords**: Reserved words with special meanings (e.g., `int`, `return`, `if`)
2. **Identifiers**: User-defined names for variables, functions, and other entities

3. **Constants**: Fixed values that do not change during program execution
4. **Operators**: Symbols representing operations (e.g., `+`, `-`, `*`, `/`)
5. **Strings**: Text enclosed in double quotes (e.g., `"Hello, World!"`)
6. **Special Symbols**: Punctuation marks and other symbols (e.g., `;`, `{}`, `()`, `[]`)

Keywords

C has a set of reserved keywords, which have predefined meanings and cannot be used as identifiers (variable or function names). Here are some common C keywords:

- **Data Types**: `int`, `float`, `char`, `double`, `void`
- **Control Statements**: `if`, `else`, `switch`, `case`, `for`, `while`, `do`, `break`, `continue`
- **Modifiers**: `long`, `short`, `signed`, `unsigned`, `const`, `volatile`
- **Others**: `return`, `sizeof`, `goto`, `static`, `enum`, `struct`, `union`, `typedef`, `extern`

Identifiers

Identifiers are names given by the programmer to elements like variables, functions, arrays, etc. Rules for defining identifiers include:

- They must start with a letter (uppercase or lowercase) or an underscore (`_`)
- They can contain letters, digits (0-9), and underscores (`_`)
- They are case-sensitive (e.g., `Var` and `var` are different)
- Cannot be a keyword (e.g., `int`, `return`)

Example: Valid identifiers include `count`, `_total`, `sum1`, while invalid ones are `1sum` (begins with a digit) and `float` (keyword).

Constants

Constants are values that do not change throughout the program. Types of constants in C include:

- **Integer Constants**: Whole numbers without decimal points (e.g., `10`, `-25`)
- **Floating-point Constants**: Numbers with decimal points (e.g., `3.14`, `-0.001`)

- **Single Character Constants**: Single characters enclosed in single quotes (e.g., `'A'`, `'$'`)
- **String Constants**: Sequence of characters enclosed in double quotes (e.g., `"Hello"`)

Variables

Variables are containers for storing data values in a program. In C, variables must be declared before use, specifying their type (e.g., `int`, `float`, `char`).

Declaration Example:

int count; // Integer variable

float price; // Floating-point variable

char grade; // Character variable

Rules for Defining Variables

When defining variables, follow these rules to ensure valid variable names:

1. Variable names must begin with a letter or an underscore (_).
2. Names can include letters, digits, and underscores but no spaces or special characters.
3. Variables are case-sensitive.
4. Reserved keywords cannot be used as variable names.
5. Only the first 31 characters are significant.

Data Types

Data types specify the type of data that a variable can store. They are essential as they determine how much memory is allocated for the variable and how the data is interpreted.

In C, data types are generally classified as follows:

- **Primary (or Basic) Data Types**: `int`, `float`, `double`, `char`
- **Derived Data Types**: Arrays, Pointers, Structures, Unions.

- **Void Data Type**: `void`

Primary Data Types

Primary data types are the fundamental data types provided by the C language to handle various kinds of data. They are:

Integer (`int`):

  - Used to store whole numbers without decimal points (e.g., `5`, `-23`).
  - Size: Typically 2 or 4 bytes, depending on the system.
  - **Example:** int age = 25;

Character (`char`):

- Used to store single characters (e.g., `'A'`, `'b'`).
- Size: 1 byte.
- Holds ASCII values, so each character internally maps to an integer.
- **Example**: char grade = 'A';

Floating-point (`float`):

- Used for decimal values or fractional numbers (e.g., `3.14`, `-0.001`).
- Size: 4 bytes.
- Precision: Up to 6-7 decimal places.
- **Example:** float price = 19.99;

Double (`double`):

- Used for high-precision decimal numbers.
- Size: 8 bytes.
- Precision: Up to 15-16 decimal places, making it suitable for scientific calculations.
- **Example:** double largeNumber = 12345.6789;

Void (`void`):

- Represents the absence of a value.
- Primarily used with functions that do not return any value or with pointers.
- **Example:**

```
void displayMessage()
{
    printf("Hello, World!");
}
```

Derived Data Types

Derived data types are built using primary data types and include structures that allow more complex data handling. Derived data types in C are:

Arrays:

- A collection of elements of the same data type stored in contiguous memory locations.
- Allows handling of large amounts of data efficiently.
- **Syntax**: `data_type array_name[array_size];`
- Example: int scores[5] = {90, 85, 88, 92, 80}; // Integer array with 5 elements

Pointers:

- Stores the memory address of another variable.
- Useful for dynamic memory allocation, arrays, and function calls.
- **Syntax**: `data_type *pointer_name;`
- Example:

  int num = 10;

  int *ptr = &num; // Pointer to an integer

Strings:

- An array of characters, ending with a null character ('\0').
- C does not have a built-in string data type, but strings are created as char arrays.
- **Example:** char name[] = "Alice"; // Character array representing a string

Functions:

- Encapsulate reusable code blocks.
- A function can take inputs, process them, and return outputs.

- Example:

```
int add(int a, int b) {
   return a + b;
}
```

User-Defined Data Types

User-defined data types allow users to create custom data structures that fit specific needs. Common user-defined data types in C include:

Structures (`struct`):

- Used to group different data types under a single unit.
- Allows storing related information together in a more organized way.
- **Syntax:**

```
struct structure_name {
   data_type member1;
   data_type member2;
   // more members...
};
```

Unions (`union`):

- Similar to structures, but members share the same memory location.
- Only one member can hold a value at any given time.
- Useful for memory-saving when only one of several variables is needed.
- **Syntax**:

```
union union_name {
   data_type member1;
   data_type member2;
   // more members...
```

};

Enumerations (`enum`):

- Allows defining a set of named integer constants, improving code readability.
- **Syntax:**

  enum enum_name { constant1, constant2, constant3, ... };

`typedef`:

- Used to give a new name (alias) to an existing data type.
- Improves readability and code maintenance, especially for complex data types.
- **Syntax:** `typedef existing_type new_name;`

Declaring and Initializing Variables

Variables must be declared with a specific data type, and optionally initialized at the time of declaration.

Syntax for Declaration: **data_type variable_name;**

Syntax for Initialization: **data_type variable_name = value;**

Variables can also be initialized later in the code after they have been declared.

Type Modifiers

Type modifiers in C allow you to alter the properties of basic data types, affecting their size or range. Common type modifiers include:

- **`signed`**: Allows a variable to hold both positive and negative values (e.g., `signed int x = -5;`).
- **`unsigned`**: Restricts a variable to only positive values (e.g., `unsigned int x = 5;`).
- **`short`**: Reduces the storage size of an integer (e.g., `short int x = 10;`).
- **`long`**: Increases the storage size of an integer (e.g., `long int x = 100000;`).

Type Conversion

Type conversion refers to changing one data type to another. There are two main types:

1. **Implicit Type Conversion (Automatic Conversion)**:
    - The compiler automatically converts a smaller data type to a larger data type when mixed types are used in an expression.
    - Example: `int x = 10; float y = 20.5; float result = x + y;` (x is implicitly converted to float)
2. **Explicit Type Conversion (Type Casting)**:
    - Conversion is manually specified by the programmer using casting.
    - Syntax: `(data_type) expression`
    - Example: `int x = 10; float y = (float)x / 3;` (x is explicitly cast to float)

Operators

Operators in C are symbols used to perform operations on variables and values. Operators are used in programs to manipulate data and variables. They are used with operands to build expressions.

C language has a rich set of operators which can be classified as follows:

Arithmetic Operators: Used for mathematical calculations.

- `+` (Addition), `-` (Subtraction), `*` (Multiplication), `/` (Division), `%` (Modulus)
- Example: `int sum = a + b;`

Relational Operators: used to compare two values. They evaluate either `true` (1) or `false` (0).

- **Equal to (`==`)**: Checks if two operands are equal.
    - Example: `a == b`
- **Not equal to (`!=`)**: Checks if two operands are not equal.
    - Example: `a != b`
- **Greater than (`>`)**: Checks if the first operand is greater than the second.
    - Example: `a > b`
- **Less than (`<`)**: Checks if the first operand is less than the second.

- Example: `a < b`
- **Greater than or equal to (`>=`)**: Checks if the first operand is greater than or equal to the second.
  - Example: `a >= b`
- **Less than or equal to (`<=`)**: Checks if the first operand is less than or equal to the second.
  - Example: `a <= b`

Logical Operators: used to combine multiple conditions or expressions. They are mostly used in decision-making statements.

- **Logical AND (`&&`)**: Returns true if both operands are true.
  - Example: `(a > 0 && b > 0)`
- **Logical OR (`||`)**: Returns true if at least one operand is true.
  - Example: `(a > 0 || b > 0)`
- **Logical NOT (`!`)**: Reverses the truth value of the operand.
  - Example: `!(a > 0)`

Assignment Operators: assign values to variables. The basic assignment operator is `=`, but there are compound assignment operators that combine assignment with another operation.

- **Simple Assignment (`=`)**: Assigns a value to a variable.
  - Example: `a = 5;`
- **Add and Assign (`+=`)**: Adds the right operand to the left operand and assigns the result to the left operand.
  - Example: `a += 5;` (equivalent to `a = a + 5`)
- **Subtract and Assign (`-=`)**: Subtracts the right operand from the left operand and assigns the result to the left operand.
  - Example: `a -= 5;` (equivalent to `a = a - 5`)
- **Multiply and Assign (`*=`)**: Multiplies the left operand by the right operand and assigns the result to the left operand.
  - Example: `a *= 5;` (equivalent to `a = a * 5`)
- **Divide and Assign (`/=`)**: Divides the left operand by the right operand and assigns the result to the left operand.
  - Example: `a /= 5;` (equivalent to `a = a / 5`)

- **Modulus and Assign (%=)**: Takes modulus using the two operands and assigns the result to the left operand.
  - Example: `a %= 5;` (equivalent to `a = a % 5`)

Increment and Decrement Operators: These operators increase or decrease the value of a variable by one.

- **Increment (++)**: Adds 1 to the operand. It has two forms:
  - **Pre-increment** (`++a`): Increases `a` by 1, then returns the new value.
  - **Post-increment** (`a++`): Returns the current value of `a`, then increases `a` by 1.
- **Decrement (--)**: Subtracts 1 from the operand. It also has two forms:
  - **Pre-decrement** (`--a`): Decreases `a` by 1, then returns the new value.
  - **Post-decrement** (`a--`): Returns the current value of `a`, then decreases `a` by 1.

Bitwise Operators: Bitwise operators work on the binary representations of numbers. They are useful for low-level programming.

- **AND (&)**: Performs a binary AND operation.
  - Example: `a & b`
- **OR (|)**: Performs a binary OR operation.
  - Example: `a | b`
- **XOR (^)**: Performs a binary XOR operation.
  - Example: `a ^ b`
- **NOT (~)**: Inverts the bits of the operand.
  - Example: `~a`
- **Left Shift (<<)**: Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
  - Example: `a << 2`
- **Right Shift (>>)**: Shifts the bits of the left operand to the right by the number of positions specified by the right operand.
  - Example: `a >> 2`

Conditional (Ternary) Operator: Used as a shorthand for `if-else` conditions.

- Syntax: `condition?expression_if_true:expression_if_false;`
- Example: `int result = (a > b) ? a : b;`

Special Operators: unique operators that perform specific tasks not covered by standard arithmetic, logical, or bitwise operators. Here's a detailed explanation of the special operators available in C:

Sizeof Operator

- **Description**: The `sizeof` operator determines the size (in bytes) of a data type or variable in memory.
- **Syntax**: `sizeof(data_type or variable)`

  Example:
  ```
  int a;

  printf("Size of int: %zu\n", sizeof(int)); // Output: 4 (on most systems)

  printf("Size of variable a: %zu\n", sizeof(a)); // Output: 4 (if a is an int)
  ```
- **Use Cases**:
  - Checking data type sizes for portability.
  - Dynamic memory allocation using functions like `malloc`.

Comma Operator (`,`)

- **Description**: The comma operator allows multiple expressions to be evaluated sequentially, with the final value of the expression being the result of the last expression.
- **Syntax**: `expr1, expr2, ..., exprN`

  Example:
  ```
  for (int i = 0, j = 10; i < j; i++, j--)

  {

      printf("%d %d\n", i, j);

  }
  ```

- **Use Cases**:
  - Combining multiple expressions in a single statement.
  - Used in loops for multiple initialization or increment steps

Dot Operator (.)

- **Description:** Used to access members of a structure or union using a structure /union variable.

Arrow Operator (->)

- **Description:** Used to access members of a structure or union using a pointer to a structure/union.

Expressions in C

An expression is any valid combination of operators, constants, variables, and functions that produce a result. Expressions are evaluated to yield a value.

Types of Expressions

1. **Constant Expressions**: Contain only constant values and are evaluated at compile time.
   - Example: 10 + 5
2. **Variable Expressions**: Involve variables and yield a result when evaluated.
   - Example: a + b
3. **Arithmetic Expressions**: Combine variables and constants using arithmetic operators.
   - Example: a + b - c * d / e
4. **Relational Expressions**: Use relational operators to compare values, returning true or false.
   - Example: a > b
5. **Logical Expressions**: Combine multiple relational expressions using logical operators, returning true or false.
   - Example: (a > b) && (b < c)
6. **Bitwise Expressions**: Use bitwise operators on integer values.
   - Example: a & b

Evaluation of Expressions

The evaluation of expressions follows the **operator precedence** and **associativity rules** in C:

- **Precedence**: Determines which operator is evaluated first in expressions with multiple operators. Operators with higher precedence are evaluated before operators with lower precedence.
- **Associativity**: Defines the direction (left-to-right or right-to-left) in which operators with the same precedence level are evaluated.

Priority (Precedence) of Operators in C

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

- High priority: * / %
- Low Priority: + -

The basic evaluation procedure includes "two" left-to-right passes through the expression. During the first pass, the high priority operators are applied as they are encountered. During the second pass, the low priority operators are applied as they are encountered.

Examples to Understand Precedence and Associativity

```
int result = 10 + 5 * 2;
```

- **Explanation**:
  1. Multiplication (`*`) is evaluated first: `5 * 2 = 10`.
  2. Addition (`+`) is next: `10 + 10 = 20`.
- **Result**: `20`.