# Programming in Modern C++

## Quick Recap Module QR1: Recap of C/1

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ac.in*

*All url's in this module have been accessed in September, 2021 and found to be functional*

## Module Objectives

- Revisit the concepts of C language
- Revisit C Standard Library components
- Revise the concept of variables and literals in C
- Revise the various data types, operators, expressions, and statements of C
- Revise the control constructs of C

## Module Outline

## First C program

- Print "Hello World"

**Source Program**

```
#include <stdio.h>

int main() {

    printf("Hello World");
    printf("\n");

    return 0;
}
```

- stdio.h header included for input / output
- main function is used to start execution
- printf function is used to print the string "Hello World"

## Data Types

Data types in C are used for declaring variables and deciding on storage and computations:

- **Built-in** / **Basic** data types are used to define raw data
  - `char`
  - `int`
  - `float`
  - `double`

  Additionally, `C89` defines:
  - `_Bool`

  All data items of a given type has the same size (in bytes). The size is *implementation-defined*
- **Enumerated Type** data are internally of `int` type and operates on a select subset.

## Data Types

Data types in C further include:

- **void**: The type specifier `void` indicates *no type*
- **Derived** data types include:
  - *Array*
  - *Structure* – `struct` & `union`
  - *Pointer*
  - *Function*
  - *String* – C-Strings are really not a type; but can be made to behave as such using functions from `<string.h>` in standard library
- **Type modifiers** include:
  - `short`
  - `long`
  - `signed`
  - `unsigned`

## Variables

- A **variable** is a name given to a *storage area*
- *Declaration of Variables*
  - Each *variable* in C has a *specific type*, which determines the size and layout of the storage (memory) for the variable
  - The *name of a variable* can be composed of *letters*, *digits*, and the *underscore character*. It *must begin* with either a *letter* or an *underscore*

```
int           i, noOfData;
char          c, endOfSession;
float         f, velocity;
double        d, dist_in_light_years;
unsigned int  i, nPeople;
short int     i, nCount;
unsigned char c, ascii_char;
int           a[10], ;
```

# Variables

- *Initialization of Variables*
  - *Initialization* is setting an *initial value* to a *variable at its declaration*
  - C variables declared can be initialized with the help of `operator '='`
  - Multiple variables can be initialized in a single statement by single value

```
int        i = 10, j = 20, numberOfWorkDays = 22;
char       c = 'x';
float      weight = 4.5;
double     density = 0.0;
const int nElements = 100;   // const must always be initialized
char*      name[] = {"Partha", "Pratim", "Das"}; // Array size is 3
```

- *Definition of Variables*
  - A *variable is defined* when a value is written to it using
    - ▷ assignment `operator '='`
    - ▷ pointer aliasing

```
int    i = 10; // Array size is 3
int*   p = &i; // Address of i set to p
i = 20;  // Assignment
*p = 30; // Pointer aliasing
```

## Literals

- *Literals* refer to *fixed values* of a *built-in type*
- *Literals* can be of any of the basic data types

```
212     // (int) Decimal literal
0173    // (int) Octal literal
0b1010  // (int) Binary literal
0xF2    // (int) Hexadecimal literal
3.14    // (double) Floating-point literal
'x'     // (char) Character literal
"Hello" // (char *) String literal
```

- In C*9, literals are constant values having `const` types as:

```
212     // (const int) Decimal literal
0173    // (const int) Octal literal
0b1010  // (const int) Binary literal
0xF2    // (const int) Hexadecimal literal
3.14    // (const double) Floating-point literal
'x'     // (const char) Character literal
"Hello" // (const char *) String literal
```

## Operators

- An **operator** denotes a *specific operation*. C has the following types of operators:
  - *Arithmetic Operators*: `+ - * / % ++ --`
  - *Relational Operators*: `== != > < >= <=`
  - *Logical Operators*: `&& || !`
  - *Bit-wise Operators*: `& | ~ << >>`
  - *Assignment Operators*: `= += -= *= /=` · · ·
  - *Miscellaneous Operators*: `. , sizeof & * ?:`
- **Arity of Operators**: Number of operand(s) for an operator
  - `+, -, *, &` operators can be *unary* (1 operand) or *binary* (2 operands)
  - `==, !=, >, <, >=, <=, &&, ||, +=, -=, *=, =, /=, &, |, <<, >>` can work only as *binary* (2 operands) operators
  - `sizeof ! ~ ++ --` can work only as *unary* (1 operand) operators
  - `?:` works as *ternary* (3 operands) operator. The condition is the first operand and the `if` true logic and `if` false logic corresponds to the other two operands.

## Operators

- **Operator Precedence**: Determines which operator will be performed first in a chain of different operators
  - The precedence of all operators are defined in the following order: (left to right – Highest to lowest precedence)
  - `(), [], ++, --, + (unary), - (unary), !, ∼, *, &, sizeof, *, /, %, +, -, <<, >>, ==, !=, *=, =, /=, &, |, &&, ||, ?:, =, +=, -=, *=, /=, <<=, >>=`
- **Operator Associativity** Indicates in what order operators of equal precedence in an expression are applied
- Consider the expression $a @ b @ c$. If the operator @ has left associativity, this expression would be interpreted as $(a @ b) @ c$. If the operator has right associativity, the expression would be interpreted as $a @ (b @ c)$
  - *Right-to-Left*: `?:, =, +=, -=, *=, /=, <<=, >>=, -, +-, !, ∼, *, &, sizeof`
  - *Left-to-Right*: `*, /, %, +, -, <<, >>, ==, !=, *=, =, /=, &, |, &&, ||`

## Expressions

- Every **expression** has a **value**
  - A *literal* is an expression
  - A *variable* is an expression
  - 1, 2 or 3 *expression/s* connected by an *operator* (of appropriate arity) is an expression
  - A *function call* is an expression
- Examples:
  - For

```
int i = 10, j = 20, k;
int f(int x, int y) { return x + y; }
```

  - Expression are:

```
10                // Value 10
i                 // Value 10
-i                // Value -10
i - j             // Value -10
k = 5             // Value 5
f(i, j)           // Value 30
i + j == i * 3 // Value true
(i == j)? 1: 2 // Value 2
```

# Statement

- A **statement** is a command for a specific action. It has *no value*
    - A ; (*semicolon*) is a (null) statement
    - An *expression terminated by a ;* (semicolon) is a statement
    - A list of *one or more statements* enclosed within a *pair of curly braces { and } or block* is a *compound statement*
    - *Control constructs* like `if`, `if-else`, `switch`, `for`, `while`, `do-while`, `goto`, `continue`, `break`, `return` are statements
- Example: *Expression statements*

| **Expressions** | **Statements** |
|---|---|
| `i + j` | `i + j;` |
| `k = i + j` | `k = i + j;` |
| `funct(i,j)` | `funct(i,j);` |
| `k = funct(i,j)` | `k = funct(i,j);` |

- Example: *Compound statements*

```
{
    int i = 2, j = 3, t;

    t = i;
    i = j;
    j = t;
}
```

## Control Constructs

- These statements control the flow based on conditions:
  - *Selection-statement*: `if`, `if-else`, `switch`
  - *Labeled-statement*: Statements labeled with identifier, `case`, or `default`
  - *Iteration-statement*: `for`, `while`, `do-while`
  - *Jump-statement*: `goto`, `continue`, `break`, `return`
- Examples:

```
if (a < b) {
    int t;

    t = a;
    a = b;
    b = t;
}
```

```
if (x < 5)
    x = x + 1;
else {
    x = x + 2;
    --y;
}
```

```
switch (i) {
    case 1: x = 5;
            break;
    case 3: x = 10;
    default: x = 15;
}
```

```
int sum = 0;
for(i = 0; i < 5; ++i) {
    int j = i * i;
    sum += j;
}
```

```
while (n) {
    sum += n;
    if (sum > 20)
        break;
    --n;
}
```

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

## Module Summary

- Revised the concept of variables and literals in C
- Revised the various data types, operators, expressions, and statements of C
- Revised the control constructs of C