Programming in Modern C++

Quick Recap Module QR2: Recap of C/2

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 Recap

- 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

Module Objectives

- Revisit the concepts of C language
- Revisit C Standard Library components
- Revise arrays, structures, unions, and pointers of C
- Revise the concepts of functions and pointers to functions of C
- Revise input/output in C

Quick Recap Module QR2

Partha Pratim Das

Objectives & Outline

Containers and Pointers

Arrays

Structures

Unions

Pointers

Pointer, Array & Structure

Functions

Declaration and Definition

By Value

By Reference

Recursion

Function pointers

Input / Output

stdin & stdout

Files

Strings

Module Summary

Module Outline

1. Containers and Pointers
    - Arrays
    - Structures
    - Unions
    - Pointers
        - Pointer Array Duality and Pointer to Structures
2. Functions
    - Declaration and Definition
    - Call and Return by Value
    - Call by Reference
    - Recursion
    - Function pointers
3. Input / Output
    - `stdin` & `stdout`
    - Files
    - Strings
4. Module Summary

Containers and Pointers

- C supports two types of **containers**:
  - **Array**: Container for one or more elements of the *same type*. This is an *indexed container*
  - **Structure**: Container for one or more members of the *one or more different / same type/s*. This container allows *access by member name*
    - **Union**: It is a special type of structure where *only one out of all the members* can be populated at a time. This is useful to deal with *variant types*
- C supports two types of **addressing**:
  - **Indexed**: This is used in an array
  - **Referential**: This is available as Pointers where the *address of a variable* can be *stored and manipulated as a value*
- Using array, structure, and pointer various **derived containers** can be built in C including **lists**, **trees**, **graphs**, **stack**, and **queue**
- **C Standard Library** has *no additional support* for containers

Arrays

- An **array** is a *collection of data items* of the *same type*, accessed using a *common name*
  - ○ *Declare Arrays*

```
#define SIZE 10
int name[SIZE];     // SIZE must be an integer constant greater than zero
double balance[10]; // Direct use of constant size
```

  - ○ *Initialize Arrays*

```
int primes[]  = {2, 3, 5, 7, 11};                // Size = 5  by initialization
int sizeOfPrimes = sizeof(primes)/sizeof(int); // Size is computed as 5
int primes[5] = {2, 3, 5, 7, 11};                // Size = 5
int primes[5] = {2, 3};                          // Size = 5, last 3 elements set to 0
```

  - ○ *Access Array elements*

```
int primes[5] = {2, 3};
int EvenPrime = primes[0]; // Read 1st element
primes[2] = 5;             // Write 3rd element
```

  - ○ *Multidimensional Arrays*

```
int mat[3][4];  // Array is stored as row-major
for(i = 0; i < 3; ++i)
    for(j = 0; j < 4; ++j)
        mat[i][j] = i + j;
```

Structures

- A **structure** is a *collection of data items* of *different types*. Data items are called *members*. The *size of a structure* is the *sum of the size of its members* or more (to take care of alignment).
  - *Declare Structures*

```
struct Complex { // Complex Number
    double re;   // Real component
    double im;   // Imaginary component
} c;             // c is a variable of struct Complex type
printf("size = %d\n", sizeof(struct Complex)); // Prints: size = 16
typedef struct _Books { // Tag _Books
    char  title[50];    // data member
    char  author[50];   // data member
    int   book_id;      // data member
} Books; // Books is an alias for struct _Books type
```

  - *Initialize Structures*

```
struct Complex x = {2.0, 3.5}; // Initialize both members
struct Complex y = {4.2};      // Initialize only the first member
```

  - *Access Structure members*

```
struct Complex x = {2.0, 3.5};
double norm = sqrt(x.re*x.re + x.im*x.im); // Access using . (dot) operator
Books book;
book.book_id = 6495407;
strcpy(book.title, "C Programming");
```

Unions

- A **union** is a *special structure* that allocates memory *only for the largest data member* and holds *only one member as a time*
  - *Declare Union*

```
typedef union _Packet { // Mixed Data Packet which can be an int, double or char
    int    iData;       // integer data
    double dData;       // floating point data
    char   cData;       // character data
} Packet;
printf("%d\n", sizeof(Packet)); // Prints: 8 = max(sizeof(int), sizeof(double), sizeof(char))
```

  - *Initialize Union*

```
Packet p = {10}; // Initialize only with a value of the type of first member (int)
printf("iData = %d\n", p.iData);    // Prints: iData = 10
```

  - *Access Union members*

```
p.iData = 2;
printf("iData = %d\n", p.iData);    // Prints: iData = 2
p.dData = 2.2;
printf("dData = %lf\n", p.dData);   // Prints: dData = 2.200000
p.cData = 'a';
printf("cData = %c\n", p.cData);    // Prints: cData = a
p.iData = 122;                      // ASCII('z') = 122
printf("iData = %d\n", p.iData);    // Prints: iData = 122. This is correct field
printf("dData = %lf\n", p.dData);   // Prints: dData = 2.199999 as 2.2 is partly changed by 122
printf("cData = %c\n", p.cData);    // Prints: cData = z as chr(122) = 'z'. Incidentally correct
```

Pointers

- A **pointer** is a variable whose *value is a memory address*. The *type of a pointer* is determined by the *type of its pointee*
- *Defining a pointer*

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *pc;    // pointer to a character
void   *pv;    // pointer to unknown / no type - will need a cast before use
```

- *Using a pointer*

```
int main() {
    int  i = 20;   // variable declaration
    int  *ip;      // pointer declaration
    ip = &i;       // store address of i in pointer ip

    printf("Address of variable: %p\n", &i); // Prints: Address of variable : 00A8F73C
    printf("Value of pointer: %p\n", ip);    // Prints: Value of pointer : 00A8F73C
    printf("Value of pointee: %d\n", *ip);   // Prints: Value of pointee : 20
}
```

Pointer Array Duality and Pointer to Structures

- *Pointer-Array Duality*

```
int a[] = {1, 2, 3, 4, 5};
int *p;

p = a;      // base of array a as pointer p
printf("a[0] = %d\n", *p);      // a[0] = 1
printf("a[1] = %d\n", *++p);    // a[1] = 2
printf("a[2] = %d\n", *(p+1)); // a[2] = 3

p = &a[2]; // Pointer to a location in array
*p = -10;
printf("a[2] = %d\n", a[2]);    // a[2] = -10
```

- malloc-free

```
// Allocate and cast void* to int*
int *p = (int *)malloc(sizeof(int));
printf("%X\n", *p);    // 0x8F7E1A2B

unsigned char *q = p; // Little endian: LSB 1st
printf("%X\n", *q++); // 0x2B
printf("%X\n", *q++); // 0x1A
printf("%X\n", *q++); // 0x7E
printf("%X\n", *q++); // 0x8F
free(p);
```

- *Pointer to a structure*

```
struct Complex { // Complex Number
    double re;   // Real component
    double im;   // Imaginary component
} c =  0.0, 0.0 ;

struct Complex *p = &c; // Pointer to structure
(*p).re = 2.5; // Member selection
p->im = 3.6;   // Access by redirection

printf("re = %lf\n", c.re); // re = 2.500000
printf("im = %lf\n", c.im); // im = 3.600000
```

- *Dynamically allocated arrays*

```
// Allocate array p[3] and cast void* to int*
int *p = (int *)malloc(sizeof(int)*3);

p[0] = 1; p[1] = 2; p[2] = 3; // Used as array

// Pointer-Array Duality on dynamic allocation
printf("p[1] = %d\n", *(p+1)); // p[1] = 2
free(p);
```

Functions: Declaration and Definition

- A **function** performs a *specific task* or *computation*
  - Has 0, 1, or more parameters. Every parameters has a type (void for no parameters)
    - If the parameter list is *empty*, the function can be called by *any number of parameters*
    - If the parameter list is void, the function can be called *only without any parameter*
  - May or may not return a result. Return value has a type (void for no result)
    - If the function has return type void, it cannot return any value (void funct(...) { return; }) except void (void funct(...) { return <void>; })
  - Function declaration

```
// Function Prototype / Header / Signature
// Name of the function: funct
// Parameters: x and y. Types of parameters: int
// Return type: int
int funct(int x, int y);
```

  - *Function definition*

```
// Function Implementation
int funct(int x, int y)
// Function Body
{
    return (x + y);
}
```

Functions: Call and Return by Value

- **Call-by-value** mechanism for passing arguments. The value of an *actual parameter* is copied to the *formal parameter*
- **Return-by-value** mechanism to return the value, if any.

```
int funct(int x, int y) {
    ++x; ++y;                   // Formal parameters changed
    return (x + y);
}
int main() { int a = 5, b = 10, z;
    printf("a = %d, b = %d\n", a, b); // Prints: a = 5, b = 10

    z = funct(a, b); // call by value. a copied to x. x becomes 5. b copied to y. y becomes 10
                     // x in funct changes to 6 (++x). y in funct changes to 11 (++y)
                     // return value (x + y) copied to z
    printf("funct = %d\n", z); // Prints: funct = 17

    // Actual parameters do not change on return (call-by-value)
    printf("a = %d, b = %d\n", a, b); // Prints: a = 5, b = 10
}
```

Functions: Call by Reference

- **Call-by-reference** is *not supported* in C in general. However, *arrays are passed by reference*

```
#include <stdio.h>

int arraySum(
    int a[],    // Reference parameter - the base address of array a is passed
    int n) {    // Value parameter
    int sum = 0;
    for(int i = 0; i < n; ++i) {
        sum += a[i];
        a[i] = 0;   // Changes the parameter values
    }
    return sum;
}

int main() {
    int a[3] = {1, 2, 3};
    printf("Sum = %d\n", arraySum(a, 3)); // Prints: Sum = 6 and changes the array a to all 0
    printf("Sum = %d\n", arraySum(a, 3)); // Prints: Sum = 0 as elements of a changed in arraySum()
}
```

Functions: Recursion

- A function may be *recursive (call itself)*
  - Has recursive step/s
  - Has exit condition/s
- Examples:

```
// Factorial of n
unsigned int factorial(unsigned int n) {
    if (n > 0) return n * factorial(n - 1); // Recursive step
    else return 1;                          // Exit condition
}

// Number of 1's in the binary representation of n
unsigned int nOnes(unsigned int n) {
    if (n == 0) return 0; // Exit condition
    else                  // Recursive steps
        if (n % 2 == 0) return nOnes(n / 2); // n is even
        else return nOnes(n / 2) + 1;        // n is odd
}
```

- Two or more functions can be *Co-recursive* - mutually calling each other. Like f() calling g() and g() calling f(). Either f() or g() or both may have exit conditions - at least one is a must

Function pointers: Delegation of function calls

```
#include <stdio.h>
struct GeoObject {
    enum { CIR = 0, REC, TRG } gCode;
    union {
        struct Cir { double x, y, r; } c;
        struct Rec { double x, y, w, h; } r;
        struct Trg { double x, y, b, h; } t;
    };
};
// Function pointer type
typedef void(*DrawFunc) (struct GeoObject);
// Draw functions for callback
void drawCir(struct GeoObject go) {
    printf("Circle: (%lf, %lf, %lf)\n",
        go.c.x, go.c.y, go.c.r); }
void drawRec(struct GeoObject go) {
    printf("Rect: (%lf, %lf, %lf, %lf)\n",
        go.r.x, go.r.y, go.r.w, go.r.h); }
void drawTrg(struct GeoObject go) {
    printf("Triag: (%lf, %lf, %lf, %lf)\n",
        go.t.x, go.t.y, go.t.b, go.t.h); }
```

```
DrawFunc DrawArr[] = { // Array of func. ptrs
   drawCir, drawRec, drawTrg };

int main() {
    struct GeoObject go;

    go.gCode = CIR;
    go.c.x = 2.3; go.c.y = 3.6;
    go.c.r = 1.2;
    DrawArr[go.gCode](go); // Call drawCir() by ptr

    go.gCode = REC;
    go.r.x = 4.5; go.r.y = 1.9;
    go.r.w = 4.2; go.r.h = 3.8;
    DrawArr[go.gCode](go); // Call drawRec() by ptr

    go.gCode = TRG;
    go.t.x = 3.1; go.t.y = 2.8;
    go.t.b = 4.4; go.t.h = 2.7;
    DrawArr[go.gCode](go); // Call drawTrg() by ptr
}
```

```
Circle: (2.300000, 3.600000, 1.200000)
Rect: (4.500000, 1.900000, 4.200000, 3.800000)
Triag: (3.100000, 2.800000, 4.400000, 2.700000)
```

Input / Output: stdin & stdout

- `int printf(const char *format, ...)` writes to `stdout` by the `format` and returns the *number of characters written*. This is a *Variadic* function.
- `int scanf(const char *format, ...)` reads from `stdin` by the `format` and returns the *number of input values that are scanned*. This is a *Variadic* function.
- Use `%s`, `%d`, `%c`, `%lf`, to print/scan *string*, `int`, `char`, and `double`

```
#include <stdio.h>

int main() {

    char str[100];
    int i;

    printf("Enter a value :\n");              // prints a constant string
    scanf("%s %d", str, &i);                  // reads a string and an integer value
    printf("You entered: %s %d\n", str, i);   // prints string and integer
}
```

- Use `stderr` to print errors

Input / Output: Files

- To *write* to or *read* from **file** (`fscanf()` and `fprintf()` are *variadic*):

```
#include <stdio.h>
#include <stdlib.h> // for exit() function

int main() {
    FILE *fp = NULL; // Pointer to handle io using buffers
    int i;

    fp = fopen("Input.dat", "r");   // open in read mode by "r"
    if (!fp) {                      // fp is NULL - open error on file
        fprintf(stderr, "Failed to open Input.dat\n");
        exit(1);
    }
    fscanf(fp, "%d", &i);           // scan from Input.dat
    fclose(fp);                     // clear buffers and close file

    fp = fopen("Output.dat", "w");  // open in write / append mode by "w" / "a"
    if (!fp) {                      // fp is NULL - open error on file
        fprintf(stderr, "Failed to open Output.dat\n");
        exit(1);
    }
    fprintf(fp, "%d^2 = %d\n", i, i*i); // prints to Output.dat
    fclose(fp);                         // write back and clear buffers and close file
}
```

Quick Recap
Module QR2

Partha Pratim
Das

Objectives &
Outline

Containers and
Pointers

Arrays
Structures
Unions
Pointers
Pointer, Array &
Structure

Functions

Declaration and
Definition
By Value
By Reference
Recursion
Function pointers

Input / Output

stdin & stdout
Files
Strings

Module Summary

Input / Output: Strings

- To *write* to or *read* from **string** (`sscanf()` and `sprintf()` are *variadic*):

```
#include <stdio.h>
#include <stdlib.h> // for itoa()

int main() {
    // Parsing a string
    char instring [] = "C++ Programming"; // Input string
    char str1[20], str2[20]; // Parsed strings

    // Read and tokeninze
    sscanf(instring, "%s %s", str1, str2);  // Tokeninze by space
    printf("Input to be parsed = \n\t%s\n", instring);
    printf("Token 1 = %s\n", str1);
    printf("Token 2 = %s\n\n", str2);

    // int to ascii conversion and parsing a number
    int i = 786; char num[10]; // number and array for digits
    sprintf(num, "%d", i); // convert a number (decimal) to string
    printf("Number %d has digits ", i);
    printf("%c %c %c\n\n", num[0], num[1], num[2]);
    printf("itoa(%d) = %s\n", i, itoa(i, num, 10)); // extract digits
}
```

```
OUTPUT
------
Input to be parsed =
        C++ Programming
Token 1 = C++
Token 2 = Programming

Number 786 has digits 7 8 6

itoa(786) = 786

----------------------------

char* itoa( // int to ascii
    int value, // number
    char* str, // ascii array
    int base); // base used

is more versatile; but here
is a quick way for decimal
conversion
```

- `sprintf()` is also useful to nicely edit the output before writing to console or file

Module Summary

- Revised arrays, structures, unions, and pointers of C
- Revised the concepts of functions and pointers to functions of C
- Revised input/output in C