Quick Recap of C

Intructors: Abir Das and Sourangshu Bhattacharya

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

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides heavily lifted from Programming in Modern C++ NPTEL Course

by Prof. Partha Pratim Das

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

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);
```

Note on Endian-ness: Link

- *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()
}
```