Programming in C

Chapter – 7

Pointers and Arrays

by, Santa Basnet

Pointers

Pointers

- A pointer is a variable that contains the address of a variable.
- Pointers are powerful but dangerous as well.
- Example:

```
int x;          // An integer.
int *xp;        // Pointer to integer.
int **xpp;      // Point to int pointer.
```

- Pointer usually lead to more compact and efficient code but the programmer must be extremely careful.

Pointers

Pointers: storage

- All the variables are stored in memory.
- Think of memory as a very large array. Every location in memory has an address and the type of address is in integer.
- In C, a memory address is called a pointer and C programming language lets you access memory locations directly.

Pointers

Pointers: Operators

- □ & (“**address of**”) operator.
  - Returns the address of its argument. (returns a pointer to its argument.)
  - The argument must be a variable name.
- □ * (“**dereference**”) in operator.
  - Returns the value stored at the given memory address.
  - The argument must be a pointer.

Pointers

Pointers: Declaration

```
int i;    // Integer i
int *p;   // Pointer to integer
int **m;  // Pointer to int pointer

p = &i;   // p now points to i
printf("%p", p);// Prints the address
                 of i (in p)

m = &p;   // m now points to p
printf("%p", m); // Prints the address
                    of p (in m)
```

Pointers

Pointers: Declaration

Class Work:

```
int a = 0;
int b = 0;
int *p;

a = 10;
p = &a;
*p = 20; // a = ? b = ?

p = &b;
*p = 10; // a = ? b = ?
a = *p;   // a = ? b = ?
```

Pointers

Passing Pointer arguments

- □ C allows you the alter values with by passing pointer arguments to a function and the Example is:

```c
/*Passing pointers to a Function to swap values*/
#include<stdio.h>

void swap(int *a, int *b){
        int t = *a;
        *a = *b;
        *b = t;
}
int main(){
        int a = 5, b = 3;
        printf("Before swap: a = %d b = %d\n", a, b);
        swap(&a, &b);
        printf("After swap: a = %d b = %d\n", a, b);
        return(1);
}
```

Pointers

Multiple initializations

- C allows you to initialize multiple values by passing pointer arguments to a function.

```c
/*Passing pointers to a Function to swap values*/
#include<stdio.h>

void initialize(int *a, char *b){
        *a = 10;
        *b = ‘t’;
}
int main(){
        int a, b;
        initialize(&a, &b);
        printf("Now, a = %d b = %c\n", a, b);
        return(1);
}
```

Pointers

Pointers are dangerous

□ What does this code ?

```
int main(){
        char *x;
        *x = ‘a’;
        return(1);
}
```

□ What about this code ?

```
int main(){
        char a = ‘x’;;
        char *p = &x;
        p++;
        printf("%c\n", *a);
        return(1);
}
```

Pointers

Arrays are pointers

- □ Pointers and arrays are closely related.
- □ Variables of array types are the addresses/pointers to the first element.
- □ You are allowed to do address arithmetic in array variables.

```
int main(){
        const char greet[20] = “Hello World”;
        for(int i=0;i<strlen(greet); i++){
                printf(“greet[i] = %c\n”, greet[i]);
                printf(“*(greet+i) = %c\n”, *(greet+i));
        }
        return(1);
}
```

Pointers

Arrays are pointers

- Pointers and arrays are closely related.
- Variables of array types are the addresses/pointers to the first element.
- You are allowed to do address arithmetic in array variables.

```
int a[5] = {3, 7, -1, 4, 6}; // Fixed size.

int a[] = {3, 7, -1, 4, 6};  // Let the compiler
                               calculate the size.
```

Pointers

Arrays are pointers

- Pointers and arrays are closely related.
- Variables of array types are the addresses/pointers to the first element.
- You are allowed to do address arithmetic in array variables.

```
>> a[0] is the same as *a.
>> a[1] is the same as *(a + 1).
>> a[2] is the same as *(a + 2).
```

Pointers

- Pointer arithmetic:

```
int x, *b, a[] = {5, 10, 15, 20, 25};

The variable x is an integer,  a and b
are pointer to an integer.

If we initialize, b = a, both a and b
points to the same address.

The statement:
     x = a[0] is identical to x = *a,
     x = a[1] is identical to x = *(a+1)
```

Pointers

Arrays are pointers

- □ Example:

```c
#include<stdio.h>
int main(){
        int a[] = {3, 7, -1, 4, 6};
        int i;
        double mean = 0.0f;
        // compute mean of values in a
        for (i = 0; i < 5; ++i){
                mean += *(a + i);
        }
        mean /= 5.0f;
        printf("Mean = %.2f\n", mean);
        return (0);
}
```

Pointers

Arrays are pointers

- Pointers and Arrays summary:

  If pa points to a particular element of an array, (pa + 1) always points to the next element, (pa + i) points i elements after pa and (pa - i) points i elements before.

  The difference is:

  >> A pointer is a variable, so pa = a and pa++ is legal.

  >> An array name is not a variable, so a = pa and a++ is illegal.

Pointers

- Passing array to a function:
  - It is possible to pass part of an array to a function, by passing a pointer to the beginning of the sub-array.
  - Example: `fun(&a[2])` or `fun(a+2)`, `a` is an array.
  - Function Definition:
  - `fun(int arr[]) { ... }`
  - `fun(int *arr) { ... }`

Pointers

Pointers, Arrays and Functions

- □ Example:

```
int strlen(char *s)
{
    int n = 0;
    for(; *s != ‘\0’; s++){
        n++;
    }
    return(n);
}

/* Calling a function, strlen */
char *p = "hello, world";
strlen(p);
strlen(p + 7);
```

Pointers

Dynamic Memory Allocation

- □ malloc: Allocates contiguous memory dynamically(i.e. at runtime).
- □ free: Deallocates the memory.
- □ Always **make sure** that malloc and free are paired.

- `int *p = (int*) malloc(n * sizeof(int));`
- An array of size **n**.
- Defined in <stdlib.h>
- `free(p);`

Pointers

Dynamic Memory Allocation

- □ Example: **n** sized dynamic array.

```c
#include<stdio.h>
#include<stdlib.h>
int main() {
    int n, *arr;
    printf("\nHow many items: ");
    scanf("%d", &n);
    arr = (int *) malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        printf("Data [%i]: ", i);
        scanf("%i", (arr+i));
    }
    printf("\nOutput: \t");
    for(int i=0;i<n;i++) printf("%d\t", *(arr+i));
    return 0;
}
```

Pointers & Structures

Self Referential Structure

- Sometimes we need to include one member i.e. a pointer to its parent structure type.

```
/**
A self referential structure of
Customer, the data part is id.
**/
struct Customer {
    int id;
    struct Customer *next;
};
```

```
/** Creating a data item. **/
struct Customer *createCustomer(int data){
    struct Customer *customer;
    customer = (struct Customer *) malloc
(sizeof(struct Customer));
    customer -> id = data;
    customer -> next = NULL;
    return customer;
}
```

```
enum Boolean {FALSE, TRUE};
/** Search a customer. **/
enum Boolean findById(int id, struct Customer *start){
    while(start != NULL) {
        if(start->id == id) return TRUE;
        start = start -> next;
    }
    return FALSE;
}
```

Pointers & Structures

Self Referential Structure

- Appending and cleaning up the allocated memory, needs to be careful.

```c
/**
A self referential structure of
Customer, the data part is id.
**/
struct Customer {
    int id;
    struct Customer *next;
};
```

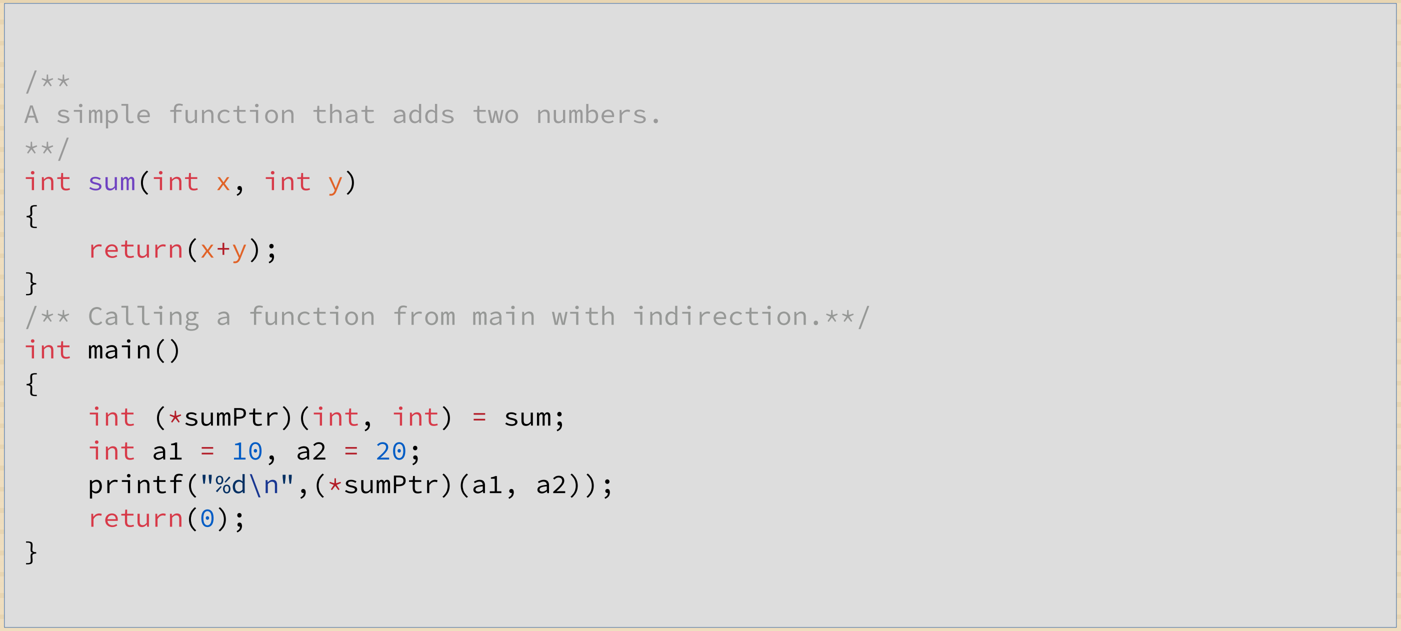
```c
/** Cleaning up the allocated memory **/
void cleanUp(struct Customer *start){
    while(start != NULL){
        struct Customer *t = start;
        start = start -> next;
        free(t);
    }
    printf("\nMemory cleaned-up successfully!\n");
}
```

```c
/** Appending a new customer at last position.**/
struct Customer *appendCustomer(struct Customer *start, int data) {
    if (start == NULL) start = createCustomer(data);
    else {
        struct Customer *t = start;
        while (t->next != NULL) t = t->next;
        t->next = createCustomer(data);
    }
    return start;
}
```

Pointers & Function

Pointer to Functions

- Function also have address and C language allows you to define a variable to a function.



```c
/**
A simple function that adds two numbers.
**/
int sum(int x, int y)
{
    return(x+y);
}
/** Calling a function from main with indirection.**/
int main()
{
    int (*sumPtr)(int, int) = sum;
    int a1 = 10, a2 = 20;
    printf("%d\n",(*sumPtr)(a1, a2));
    return(0);
}
```

Pointers & Functions

Pointer to Functions

- Passing a function as an argument, i.e. a higher order function in C.

```c
/** Adds two numbers **/
int add (int x, int y)
{
    return (x + y);
}
```

```c
/** Multiplies two numbers **/
int multiply (int x, int y)
{
    return(x*y);
}
```

```c
/** Defines operation taking function as an arguments **/
int operation(int a, int b, int (*callOper)(int, int))
{
    return ((*callOper)(a, b));
}
int main()
{
    int  a = 100, b = 45;
    int  (*plus) (int, int) = add;
    int (*cross) (int, int) = multiply;
    printf("Add = %d\n",operation(a, b, plus));
    printf("Multiply = %d\n",operation(a, b, cross));
    return 0;
}
```

Pointers & Function

Pointer to Functions

- We can even have typedef of a function pointer.

```c
/**
A simple function that adds two numbers.
**/
int sum(int x, int y)
{
    return(x+y);
}
/** Calling a function from main with indirection.**/
int main()
{
    typedef int (*SumPtr)(int, int);
    SumPtr sumObj = &sum;
    int a1 = 10, a2 = 20;
    printf("Sum (x, y) = %d\n",sumObj(a1, a2));
    return(0);
}
```

Thank you.

Questions ?