Lecture **10**

Pointers and Dynamic Memory

Fundamentals of Computer and Programming

Instructor: Morteza Zakeri, Ph.D. (m-zakeri@live.com)

Spring 2024

Modified Slides from Dr. *Hossein Zeinali* and *Dr. Bahador Bakhshi*

Computer Engineering Department, Amirkabir University of Technology

What We Will Learn

- Introduction
- Pointers and Functions
- Pointers and Arrays
- Pointers and Strings
- Pointer to Pointer & Pointer to Function
- Dynamic memory allocation

What We Will Learn

- Introduction
- Pointers and Functions
- Pointers and Arrays
- Pointers and Strings
- Pointer to Pointer & Pointer to Function
- Dynamic memory allocation

Pointer: Reference to Memory

- Pointer is a variable that
  - Contains the address of another variable
- Pointer refers to an address
- Examples

```
int i;
int *pi;
i = 20;
pi = &i;
```

Pointer: Declaration and Initialization

- <type> * <identifier>;
- Examples

```
int i, *pi;
pi = &i;
float f;
float *pf = &f;

char c, *pc = &c;


int &ref = i; // Only C++
```

Value of referred memory by a pointer

```
int *pi, *pj, i, j;
```

- **pi** variable contains the memory address
  - If you assign a value to it: **pi = &i;**
    - The address is saved in **pi**
  - If you read it: **pj = pi;**
    - The address is copied from **pi** to **pj**
- ***pi** is the value of referred memory
  - If you read it: **j = *pi;**
    - The value in the referred address is read from **pi**
  - If you assign a value to it: ***pj = i;**
    - The value is saved in the referred address

Using Pointers: Example

```
int i = 10, j;
/* address of i is 100, value of i is 10 */
/* address of j is 200, value of j is ?? */
int *pi;
/* address of pi is 300, value of pi is ?? */
pi = &i;
/* address of pi is 300, value of pi is 100 */
j = *pi;
/* address of j is 200, value of j is 10 */
*pi = 20;
/* address of pi is 300, value of pi is 100 */
/* address of i is 100, value of i is 20 */
```

Using Pointers: Example

```
double d1, d2, *pda, *pdb;
d1 = 10;
d2 = 20;
pda = &d1;
pdb = &d1;
*pda = 15;
d2 = d2 + *pdb;
printf("d2 = %f\n", d2);  // d2 = 35.0
```

Using Pointers: Example

- In C, you can cast between a pointer and an int
- A pointer is just a **32-bit** or **64-bit** number (depending on machine architecture) referring to the aforementioned chunk of memory.

```
#include <stdio.h>
int main () {
  int x = 5;
  int *ref = &x;          // now ref points to x

  printf ("%d\n", x);     // print the value of x // 5
  printf ("%p\n", &x);    // print the address of x // 0x7ffe4b79fb1c
  printf ("%p\n", &ref); // print the address of the pointer variable
   // 0x7ffe4b79fb20
  printf ("%d\n", *ref);   // print the value of the int that ref is
   pointing to // 5
  return 0;
}
```

Pointer: Reference to Memory

- Pointer variable contains an address
- There is a special address
  - NULL
- We can NOT
  - Read any value from NULL
  - Write any value to NULL
- If you try to read/write → Run time error
- NULL is usually used
  - For pointer initialization
  - Check some conditions

What We Will Learn

- Introduction
- Pointers and Functions
- Pointers and Arrays
- Pointers and Strings
- Pointer to Pointer & Pointer to Function
- Dynamic memory allocation

Call by value

```
void func(int y){
    y = 0;
}
void main(void){
    int x = 100;
    func(x);
    printf("%d", x); // 100 not 0
}
```

- Call by value
  - The value of the x is copied to y
  - By changing y, x is not changed

Call by reference

- Call by reference
  - The value of variable is not copied to function
  - If function changes the input parameter → the variable passed to the input is changed
  - Is implemented by pointers in C

```
void func(int *y){
   *y = 0;
}
void main(void){
   int x = 100;
   func(&x);
   printf("%d", x); // 0 ☺
}
```

Pointers in Functions

```
void add(double a, double b, double *res){
  *res = a + b;
  return;
}

int main(void){
  double d1 = 10.1, d2 = 20.2;
  double result = 0;
  add(d1, d2, &result);
  printf("%f\n", result); // 30.3
  return 0;
}
```


What happen?

```
double result = 0;
```

- The address of result is 100, value of result is 0

```
add(d1, d2, &result);
```

- Value of d1, Value of d2 and the address of result is copied to add

```
add(double a, double b, double *res)
```

- Value of a is the value of d1, value of b is the value of d2 and value of res is 100 and the value of *res is 0

```
*res = a + b;
```

- Value of a is added to b and output is saved in the referred address by res (100)
- But the 100 is the address of result. Therefore the value is saved in memory location **result**

Swap function (the wrong version)

```
void swap(double a, double b){
   double temp;
   temp = a;
   a = b;
   b = temp;
   return;
}

int main(void){
   double d1 = 10.1, d2 = 20.2;
   printf("d1 = %f, d2 = %f\n",d1,d2 );
                                          // d1 = 10.1, d2 = 20.2
   swap(d1, d2);
   printf("d1 = %f, d2 = %f\n",d1, d2);
   return 0;                              // d1 = 10.1, d2 = 20.2
}
```

swap function (the correct version)

```
void swap(double *a, double *b){
   double temp;
   temp = *a;
   *a = *b;
   *b = temp;
   return;
}

void main(void){
   double d1 = 10.1, d2 = 20.2;
   printf("d1 = %f, d2 = %f\n", d1, d2);
   swap(&d1, &d2);                          // d1 = 10.1, d2 = 20.2
   printf("d1 = %f, d2 = %f\n", d1, d2);
}                                           // d1 = 20.2, d2 = 10.1
```

Pointer as the function output

- Functions can return a pointer as output
- But, the address pointed by the pointer must be valid after the function finishes
  - The pointed variable must be exist
  - It must not be automatic local variable of the function
  - It can be static local variable, global variable, or the input parameter

Pointer as the function output

```
int gi;
int * func_a(void){
      return &gi;
}
float * func_b(void){
      static float x;
      return &x;
}
```

Pointer to constant: **const <type> ***

- If the input parameter
  - Is a pointer
  - But should not be changed
- Why?
  - We do not want to copy the value of variable
    - Value can be very large (array or struct)
  - We do not allow the function to change the variable

```
void func(const double *a){
   *a = 10.0; //compile error
}
```

Constant pointer: <type> * const

- If a variable is a constant pointer
  - We **cannot** assign a new address to it

```
void func(int * const a){
   int x, y;
   int * const b = &y;

   a = &x; //compile error
   b = &x; //compile error
   *a = 100; // no error
}
```

What We Will Learn

- Introduction
- Pointers and Functions
- Pointers and Arrays
- Pointer and Strings
- Pointer to Pointer & Pointer to Function
- Dynamic memory allocation

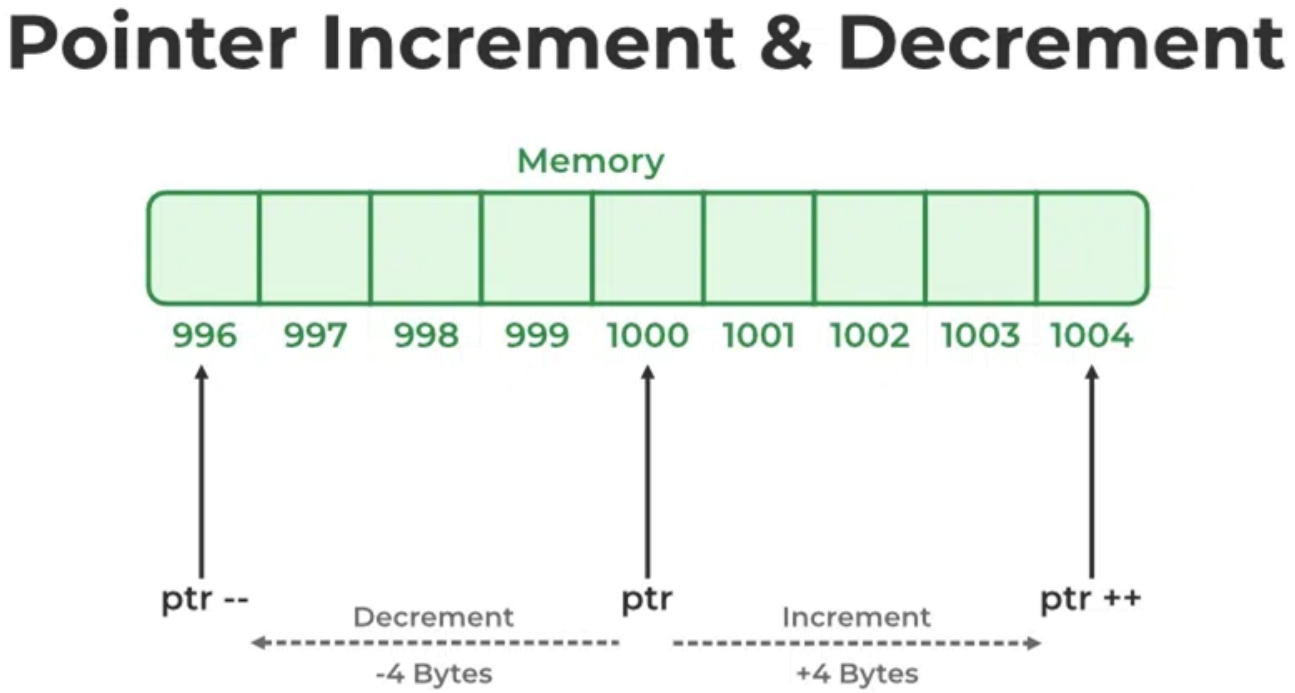
Operations on Pointers

- Arithmetic
  - ☑ <pointer> - or + <integer> (or <pointer> -= or += <integer>)
  - ☑ <pointer>++ or <pointer>--
  - ☑ <pointer> - <pointer> (they must be the same type)
  - ☒ <pointer> + <pointer> **NOT ALLOWED**
- Comparison between pointers

```
int arr[20];
int *pi, *pj, i;
pi = &arr[10];
pj = &arr[15];
i = pj - pi;        // i = 5
i = pi - pj;        // i = -5
if(pi < pj)         // if is True
if(pi == pj)        // if is False
```

Operations on Pointers

- If an integer pointer that stores address 1000 is decremented, then it will decrement by **4** (**size of an int**), and the new address will point to 996.



Operations on Pointers Examples

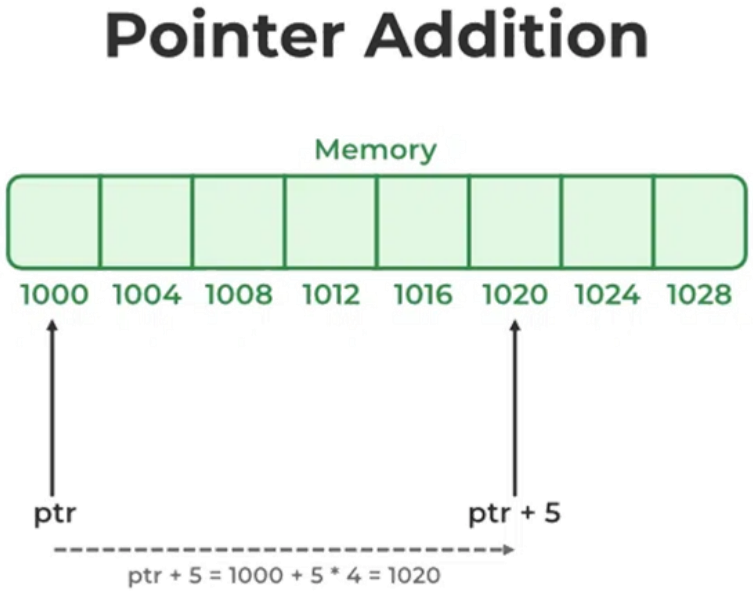
```
int a = 22;
int *p = &a;
printf("p = %u\n", p); //p = 6422288
p++;  printf("p++ = %u\n", p); //p++ = 6422292  +4   // 4 bytes
p--;
printf("p-- = %u\n", p); //p-- = 6422288  -4   // restored to original value

float b = 22.22;
float *q = &b;
printf("q = %u\n", q);  //q = 6422284
q++; printf("q++ = %u\n", q); //q++ = 6422288  +4   // 4 bytes
q--;
printf("q-- = %u\n", q); //q-- = 6422284  -4  // restored to original value

char c = 'a';
char *r = &c;
printf("r = %u\n", r);   //r = 6422283
r++;  printf("r++ = %u\n", r);  //r++ = 6422284  +1   // 1 byte
r--;
printf("r-- = %u\n", r);  //r-- = 6422283  -1  // restored to original value
```

Operations on Pointers

- The **ptr** is an **integer pointer** that stores **1000** as an address.
  - add integer **5** to it using the expression, **ptr = ptr + 5,** then,
  - the final address stored in the **ptr** will be **ptr = 1000 + sizeof(int) * 5 = 1020.**



Operations on Pointers

```
int a = 22;
int *p = &a;
int *q = p + 2;
int *r = q + 2;
int *z1, z2;

printf ("p = %u\n", p); // p = 3131704112
printf ("q = %u\n", q); // q = 3131704120
printf ("r = %u\n", r); // r = 3131704128
z1 = r - p;
printf ("z1 = r - p = %u\n", z1); // z1 = r - p = 4
z2 = r - p;
printf ("z2 = r - p = %u\n", z2); // z2 = r - p = 4
z1 = r + p; // Compiler-time error
```

invalid operands to binary + (have ‘int *’ and ‘int *’)

Operations on Pointers Examples

```
int *pi, *pj, *pk, i, j, k;
char *pa, *pb, *pc, a, b, c;
pi = &i;
pj = pi + 2;
pk = pj + 2;
pa = &a;
pb = pa + 2;
i = pj - pi;          // i = 2
j = pb - pa;          // j = 2
k = pk - pi;          // k = 4
pi = pj + pk; // compile error: No + operation for 2 pointers
pc = pi;          // compile error: Different types
i = pa - pi;      // compile error: Different ptr types
```

Array and Pointers

- Pointer can refer to each element in an array

```
int a[20];
int *pa;
pa = &a[10];  // pa refers to element 10
a[11] = *pa;  // value of pa is saved in element 11
```

- The **name** of array is the pointer to the first element

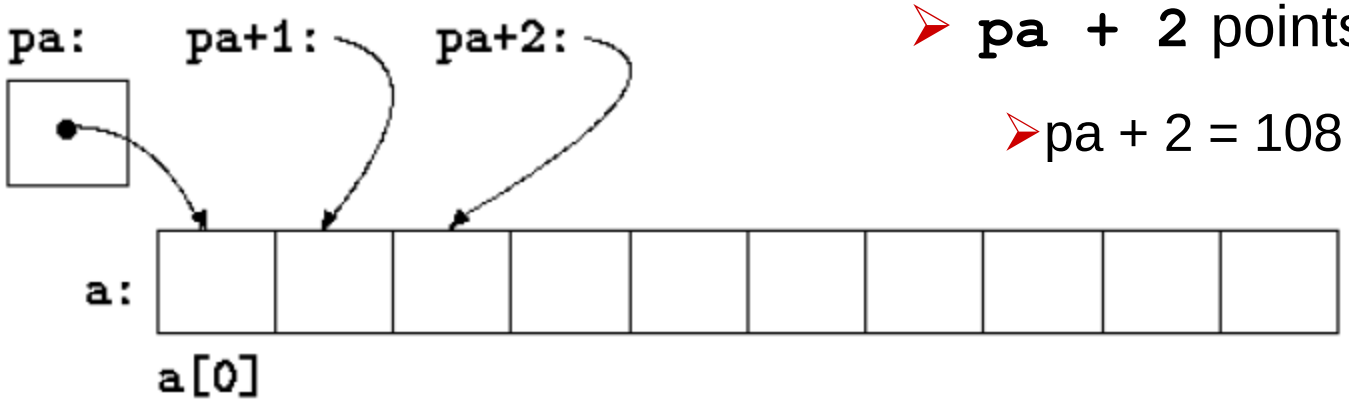
```
pa = &a[0]; //pa refers to element 0
pa = a;     //pa refers to element 0
```

Arrays and Pointers

- Example

```
int a[50];
int *pa;
pa = a;
```

- If address a = 100
  - pa = 100
- **pa+1** points to a[1]
  - pa + 1 = 104
- **pa + 2** points to a[2]
  - pa + 2 = 108



Arrays and Pointers: Similarity

```
int arr[20], *pi, j;
pi = &arr[0];    //pi refers to array
pi = pi + 2;     //pi refers to element 2
pi--;            //pi refers to element 1
j = *(pi+2);     //value of element 3

pi = arr + 2;    //pi refers to element 2
/* arr is used as a pointer */

j = pi[8];       //value of element 10
/* pi is used as array */
```

Arrays and Pointers: Difference

- We can change pointers
  - Assign new value, arithmetic and …
- We cannot change the array variable

```
int arr[20], arr2[20], *pi;
pi = arr;
pi++;
arr2 = pi;     //Compile error
arr2 = arr;    //Compile error
arr++;       //Compile error
```

Arrays in Functions (version 2)

```
int func1(int num[90]){

}

int func2(int num[], int size){

}

int func3(int *num, int size){

}
```

- **`func1`** knows size from [90], **`func2`** and **`func3`** know size from **`int size`**

Copying Arrays

تابعی که یک آرایه را در آرایه دیگر کپی کند.

```
void array_copy_wrong1(int a[], int b[]){
   a = b; //Compile error
}
void array_copy_wrong2(int *a, int *b){
   a = b; //logical error
}
void array_copy1(int dst[], int src[], int size){
   for(int i = 0; i < size; i++)
       dst[i] = src[i];
}
void array_copy2(int *dst, int *src, int size){
   for(int i = 0; i < size; i++)
       dst[i] = src[i];
}
void array_copy3(int *dst, int *src, int size){
   for(int i = 0; i < size; i++)
       *(dst + i) = *(src + i);
}
void array_copy4(int *dst, int *src, int size){
       for(int i = 0; i < size; i++, src++, dst++)
           *dst = *src;
}
```

Copying Arrays (running example)

```
int t1[10]={0}, t2[10]={0}, t3[10]={0},
t4[10]={0}, x[]={1,2,3,4,5,6,7,8,9,10};

array_copy1(t1, x, 10);
→ t1 = {1 2 3 4 5 6 7 8 9 10}
array_copy2(t2, x + 2, 8);
→ t2 = {3 4 5 6 7 8 9 10 0 0}
array_copy3(&(t3[5]), x, 5);
→ t3 = {0 0 0 0 0 1 2 3 4 5}
array_copy4(t4 + 6, &x[8], 2);
→ t4 = {0 0 0 0 0 0 9 10 0 0}
```

Computing arr1 – arr2

برنامه‌ای که تفاضل دو مجموعه را حساب کند.

```
#include <stdio.h>
int search(int *arr, int size, int num){
   int i;
   for(i = 0; i < size; i++)
       if(arr[i] == num)
           return 1;
   return 0;
}
int sub_set(int *arr1, int size_arr1, int *arr2, int
   size_arr2, int *res){
   int i;
   int result_index = 0;
   for(i = 0; i < size_arr1; i++)
       if(search(arr2, size_arr2, arr1[i]) == 0){
           res[result_index] = arr1[i];
           result_index++;
       }
   return result_index;
}
```

Computing arr1 – arr2 (Cont'd)

```
void print_arr(int *arr, int size){
   for(int i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

int main(void){
   int a1[] = {1, 2, 3, 4, 5, 6};
   int a2[] = {4, 8, 6, 11};
   int res[100];
   int result_size;

   result_size = sub_set(a1, sizeof(a1) / sizeof(int), a2,
   sizeof(a2) / sizeof(int), res);
   if(result_size > 0)
      print_arr(res, result_size);
   else
      printf("a1 - a2 = {}\n");
   return 0;
}
```

برنامه‌ای که تفاضل دو مجموعه را حساب کند.

Array of pointers

- Pointer is a type in C
  - We can define pointer variable
  - We can define array of pointer

```
int i = 10, j = 20, k = 30;
int *arr_of_pointers[10];

arr_of_pointers[0] = &i;
arr_of_pointers[1] = &j;
arr_of_pointers[2] = &k;
*arr_of_pointers[1] = *arr_of_pointers[2];
```

→ i = 10, j = 30, k = 30

Call by reference in depth

- Note: The **value of a pointer variable** is actually passed using call by value

```
void array_copy(int *dst, int *src, int size){
    for(int i = 0; i < size; i++, src++, dst++)
        *dst = *src;
    printf("%p\n%p\n", dst, src);
}

int main() {
    int a[] = {1,2,3,4,5}, b[5], *pa, *pb;
    pa = a;
    pb = b;
    printf("%p\n", pa);
    printf("%p\n", pb);
    array_copy(pb, pa, 5);
    printf("%p\n", pa);
    printf("%p\n", pb);
}
```

```
Outputs:
0xffffcc10
0xffffcbf0
0xffffcc04
0xffffcc24
0xffffcc10
0xffffcbf0
```

What We Will Learn

- Introduction
- Pointers and Functions
- Pointers and Arrays
- Pointer and Strings
- Pointer to Pointer & Pointer to Function
- Dynamic memory allocation

Strings and Pointers

- Since strings are array

```
char str1[8] = "program";
char str2[] = "program";
char str3[] = {'p', 'r', 'o', 'g', 'r',
'a', 'm', '\0'};
```

- Because arrays are similar to pointers

```
char *str4 = "program";
```

'p'	'r'	'o'	'g'	'r'	'a'	'm'	'\0'

Strings in C (Cont'd)

- str1, str2, and str3 are array
- str4 is a pointer
- We cannot assign a new value to str1, str2, str3
  - Array is a fix location in memory
  - We can change the elements of array
- We can assign a new value for str4
  - Pointer is not fix location, pointer contains address of memory
  - Content of str4 is constant, you can not change elements

char Array *vs.* char *: Example

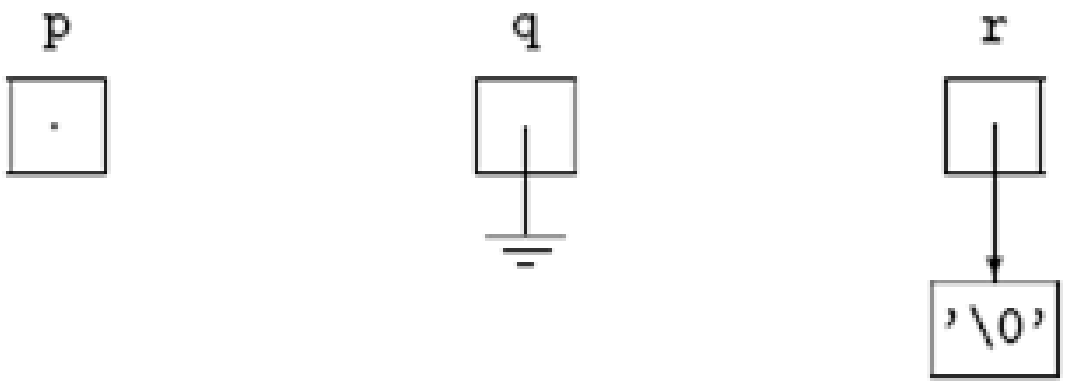
```
char str1[8] = "program";
     //this is array initialization
char *str4 = "program";
     //this is a constant string

str1[6] = 'z';
str4 = "new string";
str1 = "new array";    //Compile Error
str4[1] = 'z';         //Runtime Error
*(str4 + 3) = 'a';     //Runtime Error
```

Empty *vs.* Null

- Empty string `""`
  - Is not null pointer
  - Is not uninitialized pointer

```
char *p;
char *q = NULL;
char *r = "";
```



More String Functions

- char * strchr(const char ***s**, char c)
  - Return the pointer to the first occurrence of c in **s** or NULL

```
char *s="ABZDEZFZ";
char *pc = strchr(s, 'Z');
printf("First index of Z = %d", (pc - s));
First index of Z = 2
```

- char * strstr(const char *s1, cost char *s2)
  - Return pointer to the first occurrence of s2 in s1 or NULL

```
char *s="ABCDxyEFxyGH";
char *pc = strstr(s, "xy");
printf("First index of xy = %d", (pc - s));
First index of xy = 4
```

برنامه‌ای که دو عدد double را تا n رقم بعد از اعشار باهم مقایسه کند.

```
#include <stdio.h>
#include <string.h>

int check_equal(double d1, double d2, int n){
   int dot_index1, dot_index2;
   int search_size;
   char s1[50], s2[50];
   sprintf(s1, "%0.*lf", n, d1);
   sprintf(s2, "%0.*lf", n, d2);
   dot_index1 = strchr(s1, '.') - s1;
   dot_index2 = strchr(s2, '.') - s2;
   if(dot_index1 != dot_index2)
       return 0;

   search_size = dot_index1 + n + 1;

   if(strncmp(s1, s2, search_size) == 0)
       return 1;
   else
       return 0;
}
```

برنامه‌ای که دو عدد double را تا n رقم بعد از اعشار باهم مقایسه کند.

```
int main(void){
   int n;
   double d1, d2;
   printf("Enter numbers d1 and d2: ");
   scanf("%lf %lf", &d1, &d2);
   printf("Enter n: ");
   scanf("%d", &n);

   if(check_equal(d1, d2, n))
      printf("Are equal\n");
   else
      printf("Are Not equal\n");

   return 0;
}
```

String Tokenizer

```
#include <stdio.h>
#include <string.h>
int tokenizer(char *s, char *sep, char result[][100]){
   int res_index = 0;
   char *index;
   while((index = strstr(s, sep)) != NULL){
       int len = index - s;
       if(len > 0){
           strncpy(result[res_index], s, len);
           result[res_index][len] = '\0';
           res_index++;
       }
       s = index + strlen(sep);
   }
   if(strlen(s) > 0){
       strcpy(result[res_index], s); res_index++;
   }
   return res_index;
```

String Tokenizer (Cont'd)

```
int main(void){
   char *s =
   "a123bb123ccc123dddd123eeeee123fffffffffff123";
   char *sep = "123";
   char res[10][100];
   int num = tokenizer(s, sep, res);
   int i;
   for(i = 0; i < num; i++)
      printf("Token %d = %s\n", i + 1, res[i]);

   return 0;
}
```

Token 1 = a
Token 2 = bb
Token 3 = ccc
Token 4 = dddd
Token 5 = eeeee
Token 6 = fffffffffff

What We Will Learn

- Introduction
- Pointers and Functions
- Pointers and Arrays
- Pointer and Strings
- Pointer to Pointer and Pointer to Function
- Dynamic memory allocation

Pointer to Pointer

- Pointer is a variable
  - Has a value: address of other value
  - Has an address
- Pointer to pointer
  - Saving the address of a pointer in another pointer

```
int i, j, *pi, *pj;
int **ppi;
pi = &i;
ppi = &pi;
j = **ppi; pj = *ppi;
```

Pointer to Pointer: Example

```
int i = 10, j = 20, k = 30;
int *pi, *pj, **ppi;
pi = &i;
pj = &j;
ppi = &pi;
printf("%d\n", *pi);
printf("%d\n", **ppi);
ppi = &pj;
**ppi = 100;
printf("%d\n", j);
*ppi = &k;
printf("%d\n", *pj);
```

10

10

100

30

We will see the applications later

Pointer to functions

- Functions are stored in memory
  - Each function has its own address
- We can have pointer to function
  - A pointer that store the address of a function

type (*<identifier>)(<type1>, <type2>, …)

```
int (*pf)(char, float)
```

pf is a pointer to a function that the function return int and its inputs are char and float

Pointer to functions: Example

```
int f1(int x, char c){
   printf("This is f1: x = %d, c = %c\n", x, c); return 0;
}

int f2(int n, char m){
   printf("This is f2: n = %d, m = %c\n", n, m); return 0;
}

int main(void){
   int (*f)(int, char);   // parentheses are required here
   f = f1;                // or  f = &f1;
   (*f)(10, 'a');         // parentheses are optional here
```

This is f1: x = 10, c = a

```
   f = f2;                // or f = &f2
   (*f)(100, 'z');
   return 0;
}
```

This is f2: n = 100, m = z

Pointer to function: Application 1

- Why?
  - To develop **general** functions
    - To change function operation in run-time
- Example: atexit

```
#include <stdlib.h>

int atexit(void (*function)(void));
```

- To do a function, when the program is terminated
  - Normal termination

Pointer to function: Application 1

```
#include <stdio.h>
#include <stdlib.h>

void good_bye(void){ printf("Goooodddd Byeee :-)\n"); }

int main(void){
    int i;
    atexit(good_bye);
    printf("Enter an int: ");
    scanf("%d", &i);
    if(i < 0){
        printf("No negative\n");
        return 0;
    }
    if(i > 7){
        printf("No more than 7\n");
        return 0;
    }
    if(i % 2 == 0)
        printf("Go to class \n");
    else
        printf("Do the homework \n");

     return 0;
}
```

Pointer to function: Application 2

- Why?
  - To develop general functions
    - To change function operation in run-time
- Example: qsort function in <stdlib.h>

```
void qsort(void *arr, int num, int element_size, int
   (*compare)(void *, void *))
```

- To sort array arr with num elements of size element_size.
- The order between elements is specified by the "compare" function

Pointer to function: Application 2

```
#include <stdio.h>
#include <stdlib.h>

int int_cmp_asc(const void *i1, const void *i2){
   int a = *((int *)i1);
   int b = *((int *)i2);

   return (a > b) ? 1 : (a == b) ? 0 : -1;
}

int int_cmp_dsc(const void *i1, const void *i2){
   int a = *((int *)i1);
   int b = *((int *)i2);

   return (a > b) ? -1 : (a == b) ? 0 : 1;
}
```

Pointer to function: Application 2

```
int main(void){
   int i;
   int arr[] = {1, 7, 3, 11, 9};
   qsort(arr, 5, sizeof(int), int_cmp_asc);

   for(i = 0; i < 5; i++)
      printf("%d \n", arr[i]);

   qsort(arr, 5, sizeof(int), int_cmp_dsc);

   for(i = 0; i < 5; i++)
      printf("%d \n", arr[i]);

   return 0;
}
```

What We Will Learn

- Introduction
- Pointers and Functions
- Pointers and Arrays
- Pointer and Strings
- Pointer to Pointer & Pointer to Function
- Dynamic memory allocation

Dynamic Memory Allocation

- Until now we define variables:

```
int i; int a[200]; int x[n]
```

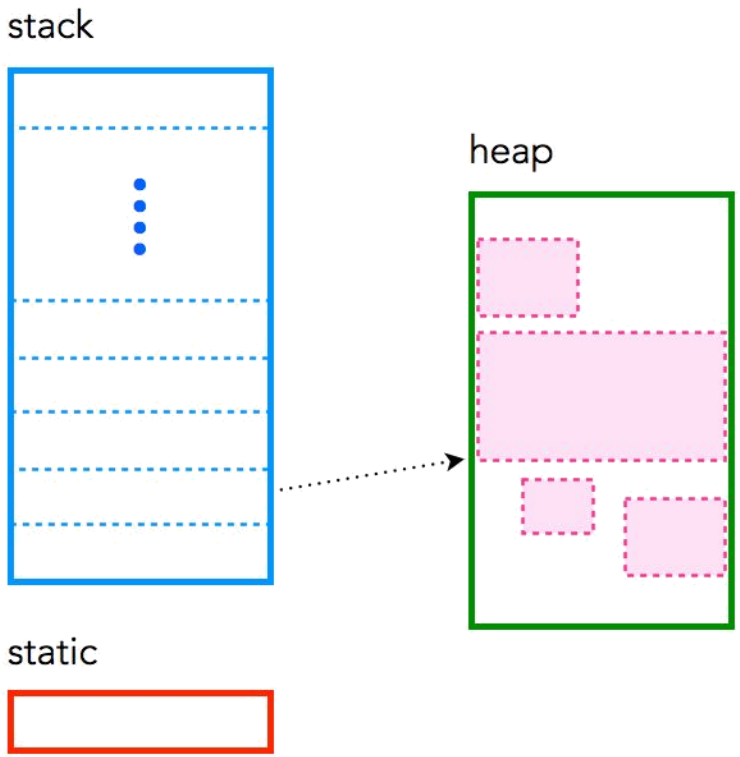
  - Memory is allocated for the variables when the scope starts
  - Allocated memory is released when the scope finishes
- We cannot change the size of the allocated memories
  - We cannot change the size of array
- These variables are in *stack*
- We want to see how to allocate memory in *heap*

Heap

- Memory is compose of a few **logical sections**
  - Stack is one of the logical sections that is used for function calls
    - All automatic variables are allocated in stack
      - Stack is managed by operating system
      - Created by function call and destroyed when function ends
- Another logical section is "Heap"
  - Heap is used for dynamic memory allocation
  - Heap is **managed by programmer** (at least in C)
    - Memory allocation functions and the `free` function

Stack, Heap, and Static Memories

- C has three different pools of memory:
  - **Static**: global variable storage, permanent for the entire run of the program.
  - **Stack**: local variable storage (automatic, continuous memory).
  - **Heap**: dynamic storage (large pool of memory, not allocated in contiguous order).



Dynamic Memory Allocation (cont'd)

- Memory allocation by **`calloc`**

```
#include <stdlib.h>

void * calloc(int num, int size);
```

- **`void *`** is generic pointer, it can be converted to every pointer type

- Allocates a block of memory for an array of num elements, each of them size bytes long, and initializes all its bits to zero.

- If memory is not available **`calloc`** returns **`NULL`**

Dynamic Memory Allocation (cont'd)

- Memory allocation by **`malloc`**

```
#include <stdlib.h>

void * malloc(int size);
```

- **`void *`** is generic pointer, it can be converted to every pointer type.
- Allocates a block of size bytes of memory, returning a pointer to the beginning of the block. Allocated memory is not Initialized.
- If memory is not available **`malloc`** returns **`NULL`**

Dynamic Memory Allocation: Example

```
int *pi;
/* allocate memory, convert it to int * */
pi = (int *) malloc(sizeof(int));
if(pi == NULL){
  printf("cannot allocate\n");
  return -1;
}

double *pd;
pd = (double *) calloc(1,sizeof(double));
```

Free

- In static memory allocation, memory is freed when block/scope is finished

- In dynamic memory allocation, we must free the allocated memory

```
int *pi;
pi = (int *) malloc(sizeof(int));
if(pi != NULL)
  free(pi);
```

برنامه‌ای که n را می‌گیرد، آرایه با اندازه n را تولید و بعد حافظه را آزاد می‌کند.

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
   int i, n;
   int *arr;
   printf("Enter n: ");
   scanf("%d", &n);
   arr = (int *)calloc(n, sizeof(int));
   if(arr == NULL){
      printf("cannot allocate memory\n");
      exit(-1);
    }
   for(i = 0; i < n; i++) /* do you work here */
         arr[i] = i;
   for(i = 0; i < n; i++)
        printf("%d\n", arr[i]);
   free(arr);
   return 0;
}
```

برنامه‌ای که n و m را می‌گیرد، ماتریس nxm را تولید و بعد حافظه را آزاد می‌کند.

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
   int i, j, n, m;
   int **arr;
   printf("Enter n, m: ");
   scanf("%d%d", &n, &m);
   arr = (int **)malloc(n * sizeof(int *));
   for(i = 0; i < n; i++)
      arr[i] = (int *)malloc(m * sizeof(int));
   for(i = 0; i < n; i++)
      for(j = 0; j < m; j++)
         arr[i][j] = i * j;
   for(i = 0; i < n; i++)
      free(arr[i]);
   free(arr);
   return 0;
}
```

Reallocation

- If we need to change the size of allocated memory: Expand or Shrink it

```
void * realloc(void *p, int newsize);
```

- Allocate **`newsize`** bytes for pointer **`p`**
- Previous data of **`p`** does **not** change
- If the new size is larger, the value of the newly allocated portion is indeterminate.

Reallocation

```
int *p;

p = (int *)calloc(2, sizeof(int));

printf("%d\n", *p);        // 0
*p = 500;
printf("%d\n", *(p+1));    // 0
*(p + 1) = 100;

p = (int *)realloc(p, sizeof(int) * 4);

printf("%d\n", *p);        // 500
p++;
printf("%d\n", *p);        // 100
p++;
printf("%d\n", *p);        // ???
p++;
printf("%d\n", *p);        // ???
```

برنامه‌ای که تعدادی عدد (تعداد آن را نمی‌دانیم) که با 1- تمام می‌شود را بگیرد و اعداد کوچکتر از میانگین را چاپ کند.

```
#include <stdio.h>
#include <stdlib.h>

void find_small(double *arr, int size){
	int i;
	double sum = 0, average;
	for(i = 0; i < size; i++)
		sum += arr[i];

	average = sum / size;
	for(i = 0; i < size; i++)
		if(arr[i] < average)
			printf("%f ", arr[i]);
}
```

برنامه‌ای که تعدادی عدد (تعداد آن را نمی‌دانیم) که با 1- تمام می‌شود را بگیرد و اعداد کوچکتر از میانگین را چاپ کند.

```
int main(void){
   double *arr = NULL; int index = 0;
   while(1){
       double num;
       printf("Enter number (-1 to finish): ");
       scanf("%lf", &num);
       if(num == -1)
           break;
       if(arr == NULL)
           arr = (double *)malloc(sizeof(double));
       else
           arr = (double *)realloc(arr, (index + 1) * sizeof(double));
       arr[index] = num;
       index++;
   }
   find_small(arr, index);
   if(arr != NULL)
       free(arr);
   return 0;
}
```

An example of multifunction application (menu-based app)

- برنامه‌ای بنویسید که منوی زیر را به کاربر نشان دهد:

1: New Data

2: Show Data

3: Exit

- اگر کاربر 1 وارد کند، برنامه عدد n را می‌گیرد، آرایه‌ای به طول n ایجاد می‌کند. سپس، n عدد را از کاربر می‌گیرد و آنها را در آرایه نگه می‌دارد.
- اگر کاربر 2 وارد کند اطلاعات وارد شده نشان داده می‌شود.
- اگر کار بر 3 وارد کند از برنامه خارج می‌شویم.

An example of multifunction application (menu-based app)

```
#include <stdio.h>
#include <stdlib.h>

void show(){
   printf("1: New Data\n");
   printf("2: Show Data\n");
   printf("3: Exit\n");
}

int main(void){
   int n;
   int *arr = NULL;
   while(1){
       int code;
       show();
       scanf("%d", &code);
       if(code == 1){
          printf("Enter size: ");
          scanf("%d", &n);
          printf("Enter data: \n");
          if(arr == NULL)
            arr = (int *)malloc(n * sizeof(int));
          else
            arr = (int *)realloc(arr, n *
sizeof(int));
          int i;
          for(i = 0; i < n; i++)
            scanf("%d", &(arr[i]));
       }
```

An example of multifunction application (menu-based app)

```
        else if(code == 2){
                printf("Your data: ");
                int i;
                for(i = 0; i < n; i++)
                        printf("%d ", arr[i]);

                printf("\n");
        }
        else if(code == 3){
                if(arr != NULL)
                        free(arr);
                exit(0);
        }
        else{
                printf("Unknown input ...\n");
        }
    }
}
```

What We Will Learn

- Introduction
- Pointers and Functions
- Pointers and Arrays
- Pointer and Strings
- Pointer to Pointer & Pointer to Function
- Dynamic memory allocation
- Common Bugs

Common Bugs

- Be very very careful about pointers
  - Invalid type of value assigned to pointer

```
int i, *pi = &i;
*pi = 29.090; // No warning in some compilers!!!
```

  - Invalid usage of pointers

```
int *pi, i;
pi = i;
i = pi;
```

- We cannot change constant string
  - `char *s = "abc";`
  - `*(s + 1) = 'z'; // Run Time Error`

Reference

- Reading Assignment: Chapter 7 of “C How to Program”