Lecture 18-20

Pointers

Outline

- Defining and using Pointers
- Operations on pointers
  - Arithmetic
  - Logical
- Pointers and Arrays
- Memory Management for Pointers

Pointer Fundamentals

- When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable
  - int x;

00000000
00000000
00000000
00000011

- When a value is assigned to a variable, the value is actually placed to the memory that was allocated
  - x=3;

Recall Variables

```
int x1=1;

int x2=7;
```

name	address	Memory - content
	0	
	1	
x1	2	1 = 00000001
	3	
	4	
x2	5	7 = 00000111
	6	
	...	

Recall Variables

- Recall a variable is nothing more than a convenient name for a memory location.
  - The type of the variable (*e.g.,* **int**) defines
    - how the bits inside that memory location will be interpreted, and
    - what operations are permitted on this variable.
- Every variable has an address.
- Every variable has a value.

The Real Variable Name is its Address!

- There are 4 billion (2^{32}) different addresses, and hence 4 billion different memory locations.
  - Each memory location is a variable (whether your program uses it or not).
  - Your program will probably only create names for a small subset of these “potential variables”.
  - Some variables are guarded by the operating system and cannot be accessed.
- When your program uses a variable the compiler inserts machine code that calculates the address of the variable.
  - **Only by knowing the address can the variables be accessed**.

Pointers

- When the value of a variable is used, the contents in the memory are used
  - y=x;   will read the contents in the 4 bytes of memory, and then assign it to variable y
- &x can get the address of x (referencing operator &)
- The address can be passed to a function:
  - scanf("%d", &x);
- The address can also be stored in a variable ……

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;
```

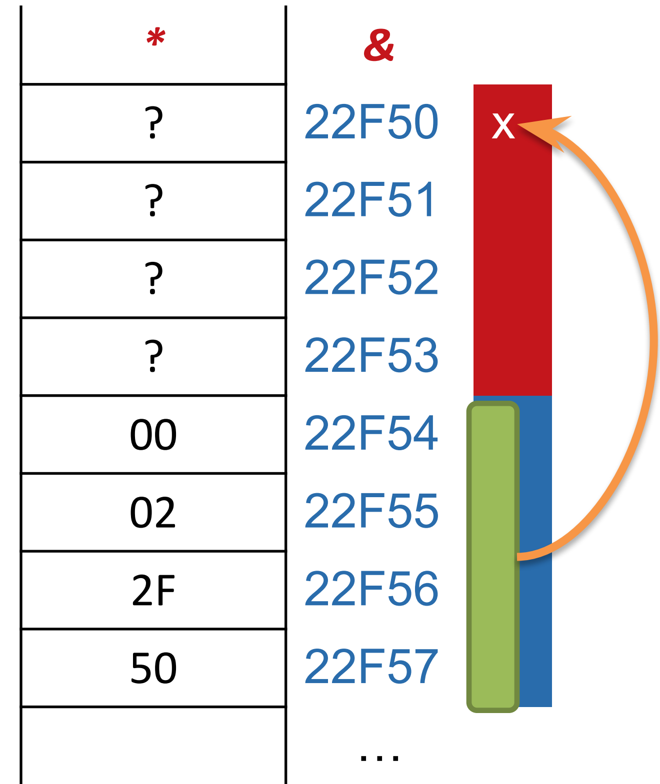
Pointers

- To declare a pointer variable

  type * PointerName;

- For example:

```
int     x;
int     * p;      //p is a int pointer
// char   *p2;
p = &x;          /* Initializing p */
```

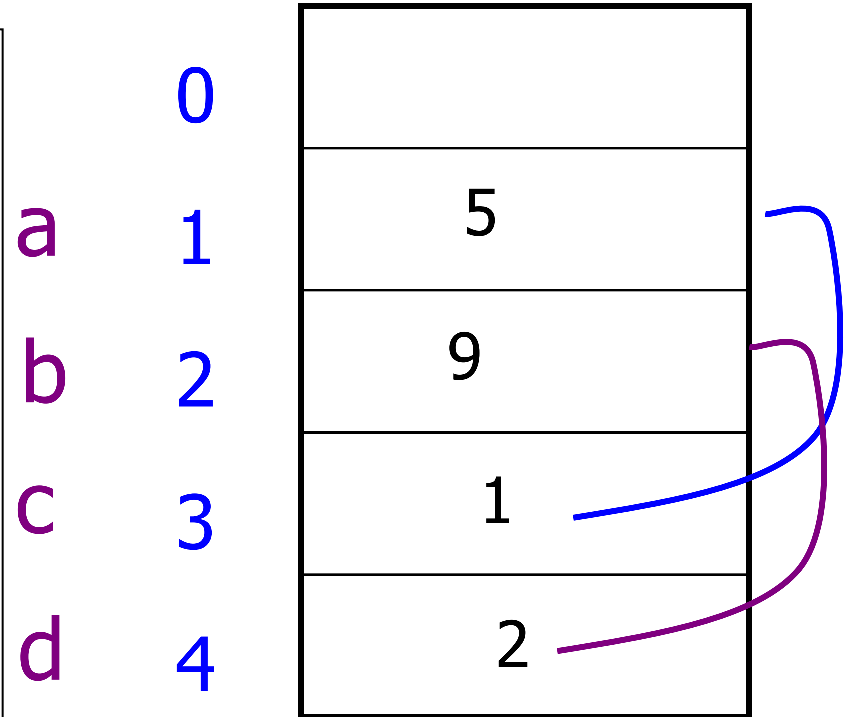


Pointer: Declaration and Initialization

```
int i, *pi;

pi = &i;

float f;

float *pf = &f;

char c, *pc = &c;
```

Addresses and Pointers

```
int a, b;
int *c, *d;
a = 5;
c = &a;
d = &b;
*d = 9;
printf(…,c, *c,&c)
printf(…,a, b)
```



Addresses and Pointers

- A pointer variable is a variable!
  - It is stored in memory somewhere and has an address.
  - It is a string of bits (just like any other variable).
  - Pointers are 32 bits long on most systems.

Using Pointers

- You can use pointers to access the values of other variables, *i.e.* the contents of the memory for other variables

- To do this, use the * operator (**dereferencing operator**)
  - Depending on different context, * has different meanings

* has different meanings in different contexts

```
a = x * y;
```
 multiplication

```
int *ptr;
```
 declare a pointer

* is also used as **indirection** or **de-referencing** operator in C statements.

```
ptr = &y;
a = x *  *ptr;
```

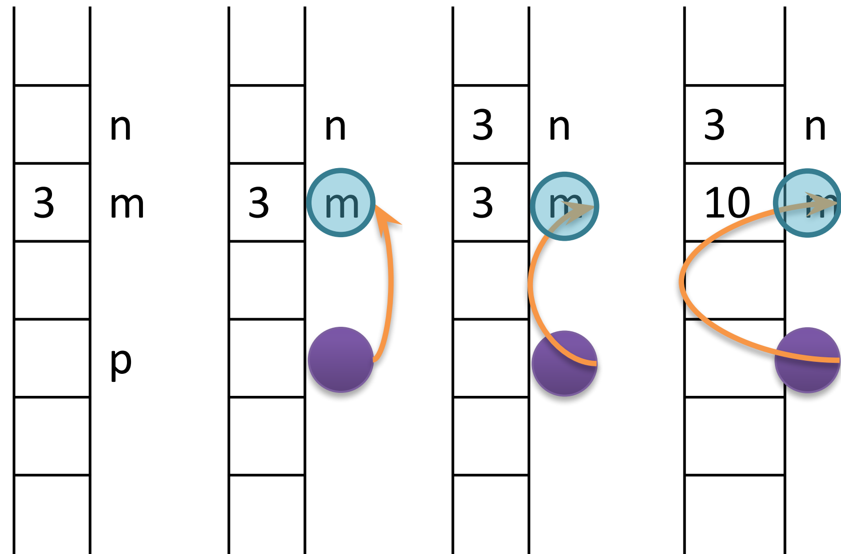
Using Pointers

- You can use pointers to access the values of other variables, *i.e.* the contents of the memory for other variables

- To do this, use the * operator (**dereferencing operator**)
  - Depending on different context, * has different meanings

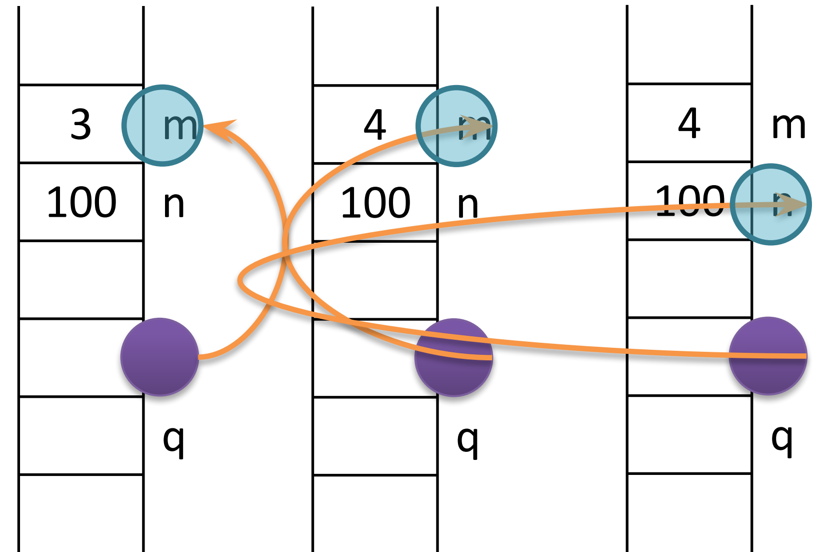
- For example:

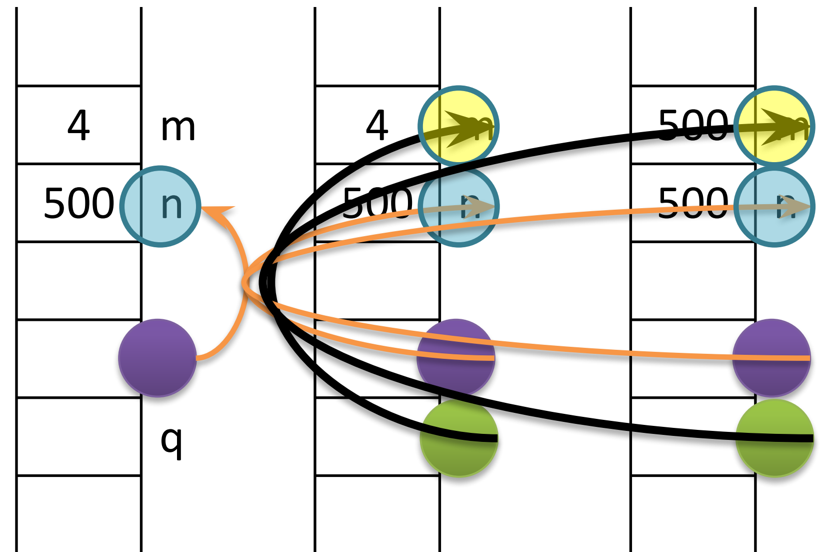
```
int n, m = 3, *p;
p = &m; // Initializing
n = *p;
printf("%d\n",  n);  // 3
printf("%d\n", *p); // 3
*p = 10;
printf("%d\n",  n);  // 3
printf("%d\n", *p); // 10
```



An Example

```
int m = 3, n = 100, *p, *q;
p = &m;
printf("m is %d\n", *p); // 3
m++;
printf("now m is %d\n", *p); // 4

p = &n;
printf("n is %d\n", *p); // 100
*p = 500;
printf("now n is %d\n", n); // 500

q = &m;
*q = *p;
printf("now m is %d\n", m); // 500
```

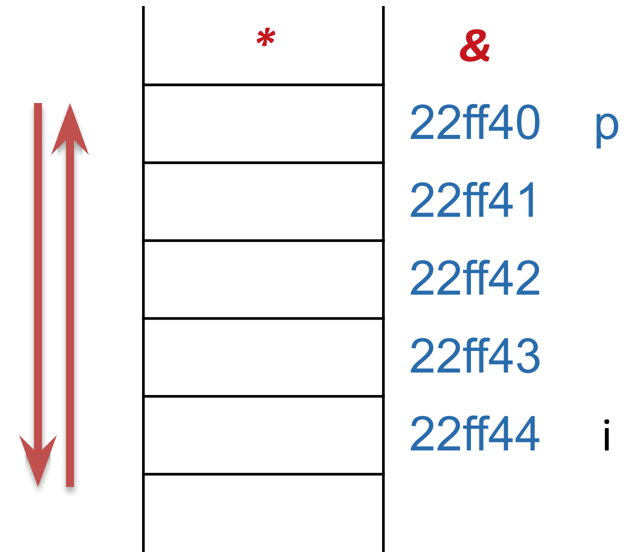




An Example

```
int i = 25;
int *p;
p = &i;
```

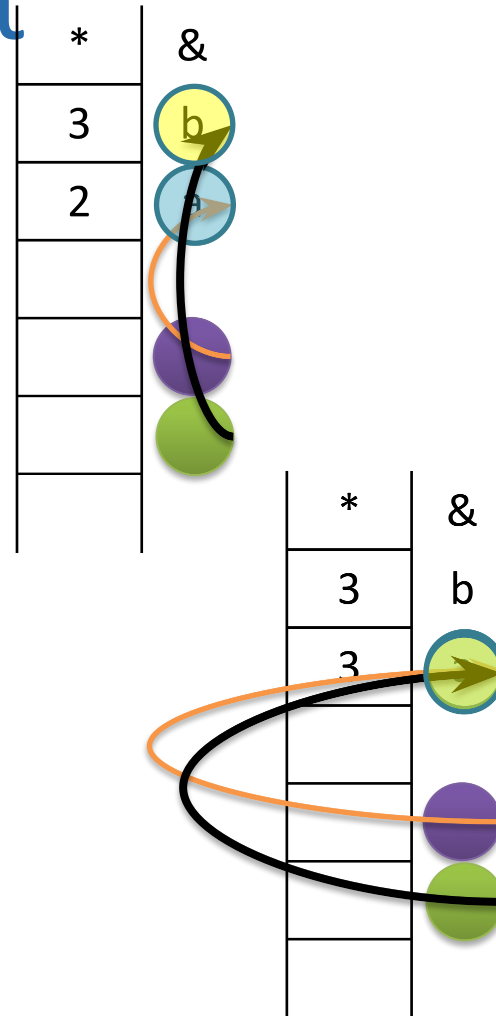
Flow of address is complier dependent




```
printf("%x %x", &p, &i);  // 22ff40   22ff44
printf("%x %p", p, p);  // 22ff44   0022ff44
printf("%d %d", i, *p);  // 25 25
```

Pointer Assignment

```
int a = 2, b = 3;
int  *p1, *p2;
p1 = &a;
p2 = &b;
printf("%p %p", p1 ,p2);

*p1 = *p2;
printf("%d %d", *p1,  *p2);

p2 = p1;
printf("%p %p", p1, p2);
printf("%p %p", &p1, &p2);
```



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

Exercise: Trace the following code

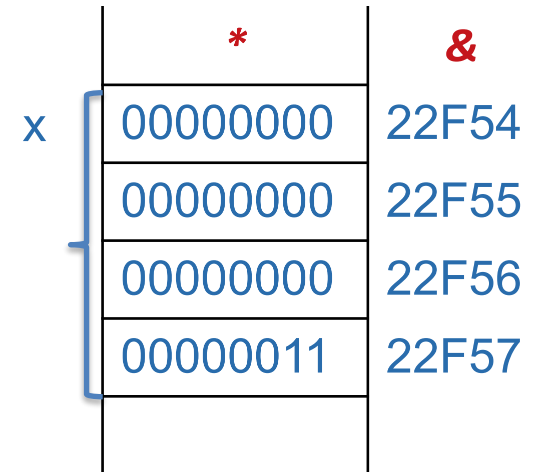
```
int x, y;
int *p1, *p2;
x = 3 + 4;
Y = x / 2 + 5;
p1 = &y;
p2 = &x;
*p1 = x + *p2;
*p2 = *p1 + y;
printf(…,p1,*p1,&p1)
printf(…,x,&x,y,&y)
```

name	address	memory
	510	
x	511	?
y	512	?
p1	513	?
p2	514	?

```
60ff04 14 60ff00
28 60ff08 14 60ff04
```

Pointer Fundamentals

- When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable
  - int x;   // &x = 22f54;
  - &x = 22f54;   // Error



- When a value is assigned to a variable, the value is actually placed to the memory that was allocated
  - x = 3;   // * (&x) = 3;
  - *x = 3;   //   Error

Allocating Memory for a Pointer

```
// The following program is wrong!
#include <stdio.h>
int main()
{
    int *p;
    scanf("%d", p);
    return 0;
}
```

```
// This one is correct:
#include <stdio.h>
int main()
{
    int *p;
    int a;
    p = &a;
    scanf("%d", p);
    return 0;
}
```

Characteristics of Pointers

- We've seen that pointers can be initialized and assigned (like any variable can).
  - They can be local or global variables (or parameters)
  - You can have an array of pointers
  - etc., just like any other kind of variable.
- We've also seen the dereference operator (*).
  - This is the operation that really makes pointers special (pointers are the only type of variable that can be dereferenced).

Pointer "Size"

- Note: Pointers are all the same size. On most computers, a pointer variable is four bytes (32 bits).
  - However, the variable that a pointer points to can be arbitrary sizes.
  - A `char*` pointer points at variables that are one byte long. A `double*` pointer points at variables that are eight bytes long.
- When pointer arithmetic is performed, the actual address stored in the pointer is computed based on the size of the variables being pointed at.

Constant Pointers

- A pointer to const data ***does not allow modification of the data through the pointer***

```
const  int  a = 10, b = 20;
a = 5; // Error
const  int  *p;
int  *q;
p = &a; // or p=&b;
*p = 100; // Error :  p is (const int *)
p = &b;
q = &a;
*q = 100; // OK !!!
```

Constant Pointers

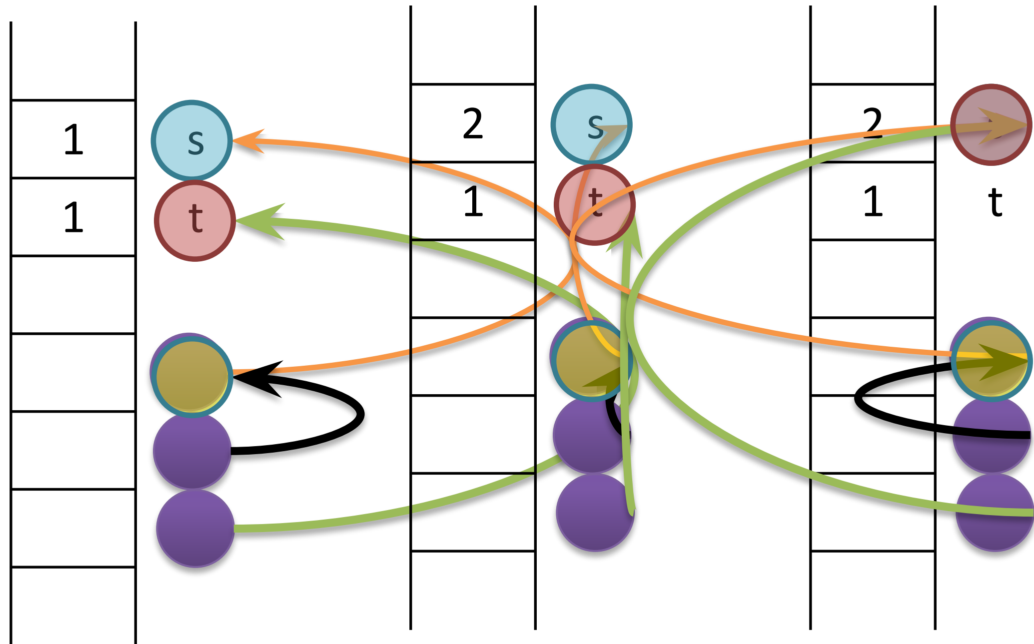
```
int x; /* define x */
int y; /* define y */

/*ptr is a constant pointer to an integer that can be
  modified through ptr, but ptr always points to the
  same memory location */
int * const ptr = &x;

*ptr = 7; /* allowed: *ptr is not const */
ptr = &y; /* error: cannot assign new address */
```

Constant Pointers

```
int x = 5; /*  initialize x */
int y;     /* define y */

/*ptr is a constant pointer to a constant integer. ptr
  always points to the same location; the integer at
  that location cannot be modified */
const  int * const ptr = &x;

*ptr = 7; /* error: cannot assign new value */
ptr = &y; /* error: cannot assign new address */
```

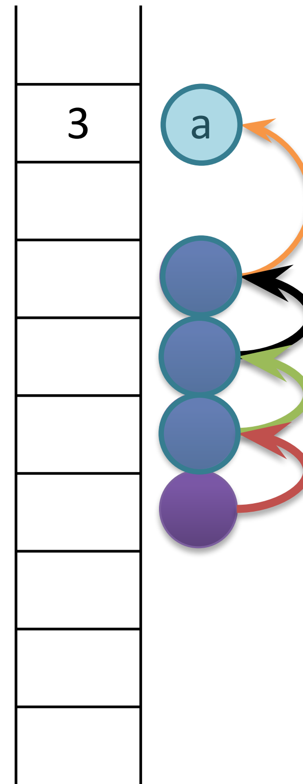
Pointer to pointer

```
int main(void)
{
        int s = 1;
        int t = 1;
        int *ps = &s;
        int **pps = &ps;
        int *pt = &t;
        **pps = 2;
        pt = ps;
        *pt = 3;
        return 0;
}
```



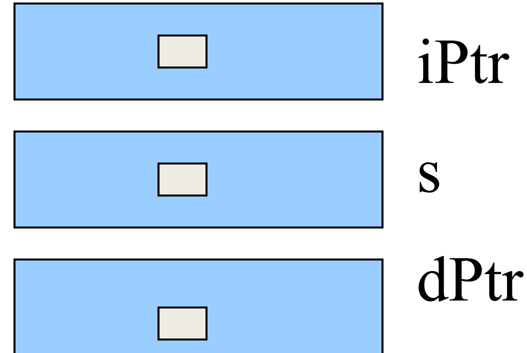
Multiple indirection

```
int a = 3;
int *b = &a;
int **c = &b;
int ***d = &c;
int ****f = &d;
```



Pointer Initialization

```
int *iPtr=0;
char *s=0;
double *dPtr=NULL;
```



!!! When we assign a value to a pointer during it is declaration, we mean to put that value into pointer variable (no indirection)!!!

`int *iPtr=0;` is same as

```
int *iPtr;
iPtr=0; /* not like *iPtr = 0; */
```

NULL Pointer

- Special constant pointer **NULL**
  - Points to no data
  - Dereferencing illegal
  - To define, include **<stdio.h>**

  - **int *q = NULL;**

NULL Pointer

- 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

NULL Pointer

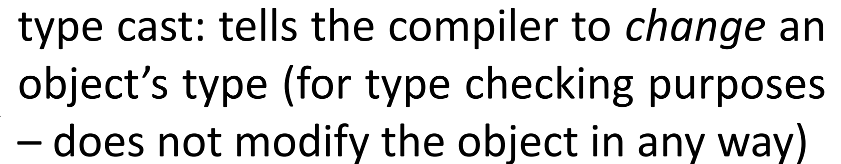
- Often used as the return type of functions that return a pointer to indicate function failure

```
int *myPtr;
myPtr = myFunction( );
if (myPtr == NULL){
   /* something bad happened */
}
```

- Dereferencing a pointer whose value is NULL will result in program termination.

Generic Pointers: void *

- **void ***: a pointer to ***anything***




```
void   *p;
int     i;
char    c;
p = &i;
p = &c;
putchar(*(char *)p);
```

- Lose all information about what type of thing is pointed to
  - Reduces effectiveness of compiler's type-checking
  - ***Can't use pointer arithmetic***

Operations on Pointers

- Arithmetic

  <pointer> - or + <integer> (or <pointer> -= or += <integer>)

  <pointer> - <pointer> (they must be the same type)

  <pointer>++ or <pointer>--

- 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
```

Arithmetic Operations

- When an *integer* is *added* to or *subtracted* from a pointer, the new pointer value is changed by the *integer times the number of bytes* in the data variable the pointer is pointing to

  – For example, if the pointer *p* contains the address of a double precision variable and that address is 234567870, then the statement:

    ```
    p = p + 2; // 234567870 + 2 * sizeof(double)
    ```

    would change *p* to 234567886

Operations on Pointers

```
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;
j = pb - pa;
k = pk - pi;
```

i = 2
j = 2
k = 4

```
pi = pj + pk;   // compile error: No + for 2 pointers
pc = pi;        // compile error: Different types
i = pa - pi;    // compile error: Different ptr types
```

Arithmetic Operations

- A pointer may be *incremented* or *decremented*
  - An *integer* may be *added* to or *subtracted* from a pointer.
  - *Pointer variables* may be *subtracted* from *one another*

```
int a, b;
int *p = &a, *q = &b;
p = p + q ;  // Error
p = p * q;   // Error
p = p / q;   // Error
p = p - q;   // OK
p  = p + 3;
p += 1.6; // Error
p %= q; // Error
```

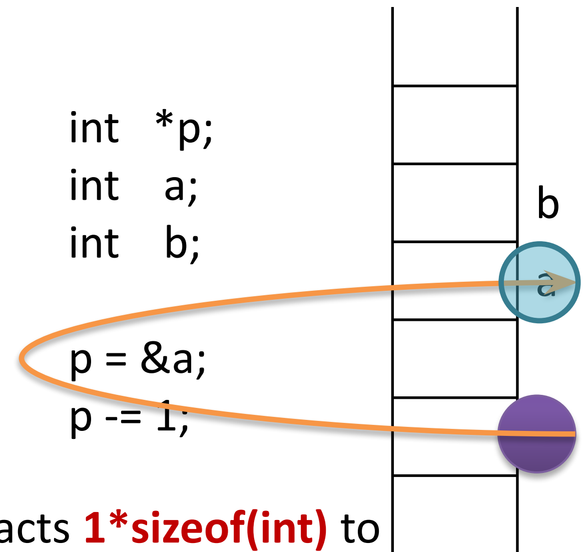
Arithmetic Operations

pointer + number *pointer – number*

```
char   *p;
char    a;
char    b;

p = &a;
p -= 1;
```

In each, p now points to b !!!
(<u>complier dependent</u>)

```
int   *p;
int    a;
int    b;

p = &a;
p -= 1;
```



subtracts **1*sizeof(char)** to the memory address

subtracts **1*sizeof(int)** to the memory address

Pointer arithmetic should be used cautiously

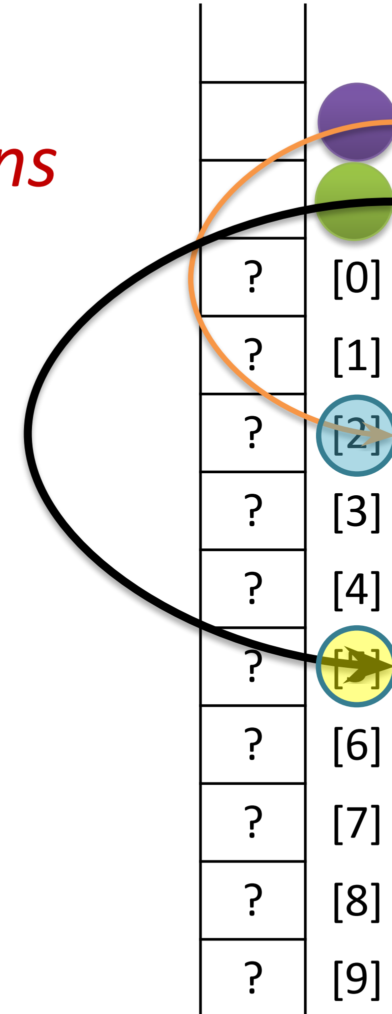
Comparing Pointers

- Pointers can also be compared using $==$, $!=$, $<$, $>$, $<=$, and $>=$
  - Two pointers are "equal" if they point to the same variable (*i.e.,* the pointers have the same value!)
  - A pointer *p* is "less than" some other pointer *q* if the address currently stored in *p* is smaller than the address currently stored in *q.*
  - **It is rarely useful to compare pointers with <** unless both *p* and *q* "point" to variables in the same array.

Logical Operations

- Pointers can be used in *comparisons*

```
int a[10], *p, *q , i;
p = &a[2];
q = &a[5];
i = q - p;        /* i is 3*/
i = p - q;        /* i is -3 */
a[2] = a[5] = 0;
i = *p - *q;      // i = a[2] – a[5]
if (p < q) ...;           /* true */
if (p == q)...;           /* false */
if (p != q) ...;          /* true */
```



Pointers and Arrays

- the value of an *array name* is also an *address*
- In fact, *pointers* and *array names* can be used *interchangeably* in many (*but not all*) cases

- The major differences are:
  - Array names come with *valid spaces* where they "point" to. And you cannot ***"point" the names to other places***.
  - Pointers do not point to valid space when they are created. ***You*** have to point them ***to some valid space*** (initialization)

Pointers and Arrays

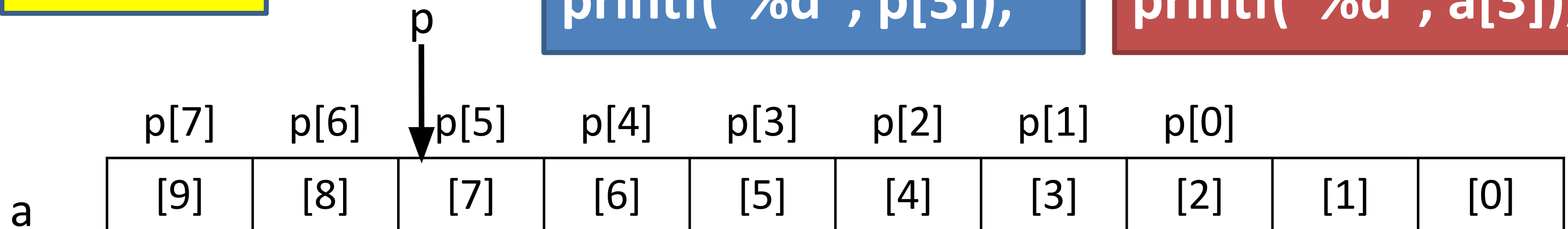
Array ≈ pointer to the initial (0th) array element

a ≡ &a[0]
a[i] ≡ *(a+i)
&a[i] ≡ a + i

<u>Example:</u>
```
int   a, *p;
p=&a;
*p = 1;
p[0] = 1;
```

```
int a[ 10 ], *p;
p = &a[2];
p[0] = 10;
p[1] = 10;
printf("%d", p[3]);
```

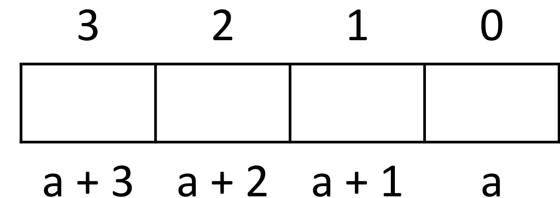
```
int a[ 10 ], *p;

a[2] = 10;
a[3] = 10;
printf("%d", a[3]);
```



Pointers and Arrays

Array ≈ pointer to the initial (0th) array element

a ≡ &a[0]
a[i] ≡ *(a+i)
&a[i] ≡ a + i




```
int  i;
int  array[10];

for (i = 0; i < 10; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

These two blocks of code are functionally equivalent

An Array Name is Like a Constant Pointer

- Array name is like a **constant pointer** which points to the first element of the array

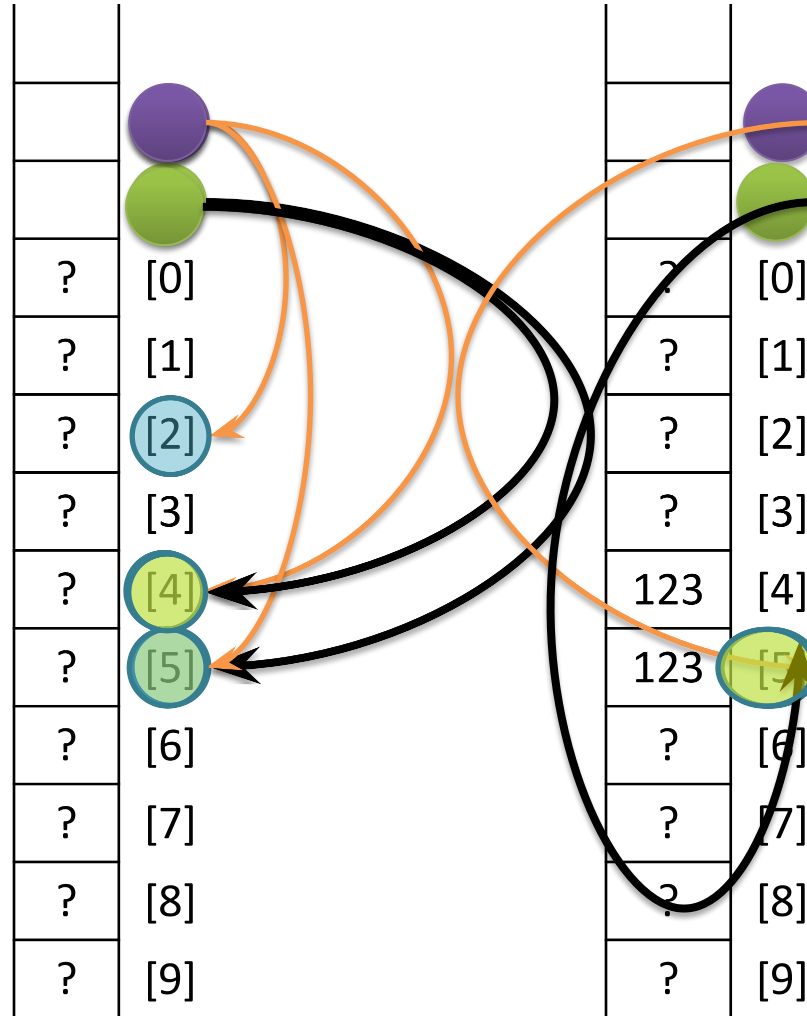
```
int a[10], *p, *q;
p = a;        /* p = &a[0] */
q = a + 3;    /* q = &a[0] + 3 */
a ++;         /*  Error !!! */
```

Example

```
int a[10], i;
int *p = a;  // int *p = &a[0];

for (i = 0; i < 10; i++)
    scanf("%d", a + i); // scanf("%d", &a[i]);

for (i = 9; i >= 0; --i)
    printf("%d", *(p + i));
    // printf("%d", a[i]);
    //printf("%d", p[i]);

for (p = a; p < &a[10]; p++)
    printf("%d", *p);
```

An example

```
int a[10], *p, *q;
p = &a[2];
q = p + 3;
p = q – 1;
p++;
q--;
*p = 123;
*q = *p;
q = p;
printf("%d", *q);   /* printf("%d", a[5])  */
```



An Example

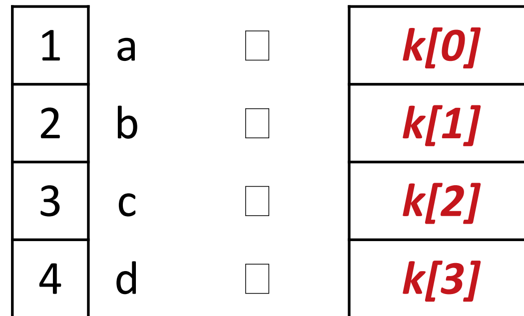
```
int a[10], *p;
a++; //Error
a--; // Error
a += 3; //Error
p = a; // p = &a[0];
p ++; //OK
p--; // Ok
P +=3; // Ok
```

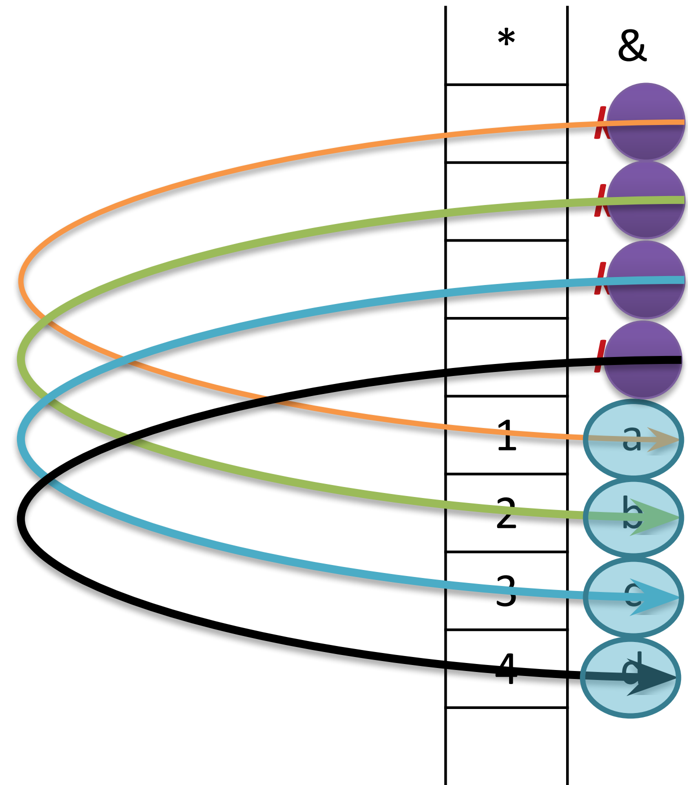
Array Example Using a Pointer

```
int x[4] = {12, 20, 39, 43}, *y;
y = &x[0];              // y points to the beginning of the array
printf("%d\n", x[0]);       // outputs 12
printf("%d\n", *y);         // also outputs 12
printf("%d\n", *y+1);       // outputs 13 (12 + 1)
printf("%d\n", (*y)+1);     // also outputs 13
printf("%d\n", *(y+1));     // outputs x[1] or 20
y+=2;                   // y now points to x[2]
printf("%d\n", *y);         // prints out 39
*y = 38;                // changes x[2] to 38
printf("%d\n", *y-1);       // prints out x[2] - 1 or 37
printf("%d\n", *y++);       // prints out x[2]  and sets y to point
                      //at the next array element
printf("%d\n", *y);         // outputs x[3] (43)
```

Array of Pointers

```
int a=1, b=2, c=3, d=4;
int *k[4] = {&a, &b, &c, &d};
```






```
printf("%d %d %d %d", *k[0], *k[1],*k[2],*k[3]);
```

Strings

- In C, strings are just an array of characters
  - Terminated with '\0' character
  - Arrays for bounded-length strings
  - Pointer for constant strings (or ***unknown length***)

```
char  str1[15] = "Hello, world!";
```

```
char  str1[]   = "Hello, world!";
char *str2     = "Hello, world!";
```

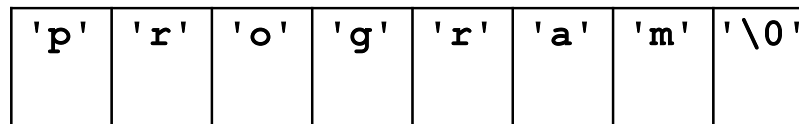
Strings & Pointers

- 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";
```



Strings in C (cont'd)

- str1,str2 and str3 are array
- str4 is a pointer
- We can not 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

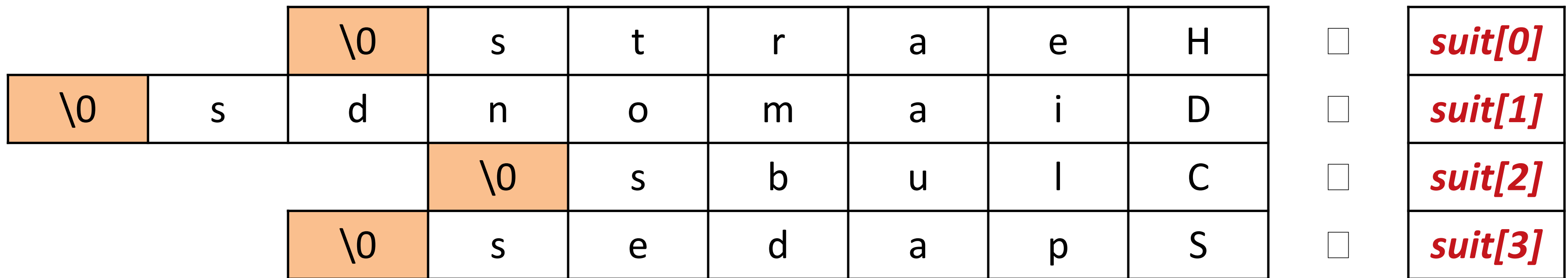
```
char str1[8] = "program";
     //this is array initialization
char *str4 = "program";
     //this is a constant string

str1[6] = 'z';
str4 = "new string";

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

An Example

```
char *str, s[] = "ALIREZA";
printf("%s", s); // ALIREZA
printf(s); // ALIREZA
printf("%s", s + 3); // REZA

scanf("%s", s);
scanf("%s", &s[0]);

str = s;
while(* str)
    putchar(*str++); // *s++  : Error
```

Array of Pointers

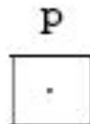
```
char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs",
   "Spades" };
```

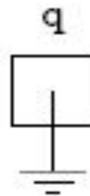


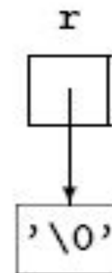
Empty vs. Null

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

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







Multi-Dimensional Arrays

int a[row][col];

a[row][col] ≡ *(a[row] + col)

a ≡ a[0][0] ≡ a[0]

```
scanf(" %d ", &a[0][0]) ≡ scanf(" %d ", a[0])

printf (" %d ", a[0][0]) ≡ printf(" %d ", *a[0])

scanf(" %d ", &a[2][2]) ≡ scanf(" %d ", a[2]+ 2)

printf (" %d ", a[2][2]) ≡ printf(" %d ", *(a[2] + 2))
```

a[0] + 2

[0][9]	[0][8]	[0][7]	[0][6]	[0][5]	[0][4]	[0][3]	[0][2]	[0][1]	[0][0]	□	*a[0]*
[1][9]	[1][8]	[1][7]	[1][6]	[1][5]	[1][4]	[1][3]	[1][2]	[1][1]	[1][0]	□	*a[1]*
[2][9]	[2][8]	[2][7]	[2][6]	[2][5]	[2][4]	[2][3]	[2][2]	[2][1]	[2][0]	□	*a[2]*
[3][9]	[3][8]	[3][7]	[3][6]	[3][5]	[3][4]	[3][3]	[3][2]	[3][1]	[3][0]	□	*a[3]*
[4][9]	[4][8]	[4][7]	[4][6]	[4][5]	[4][4]	[4][3]	[4][2]	[4][1]	[4][0]	□	*a[4]*

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;
}
```

Swap function (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);
   printf("d1 = %f, d2 = %f\n", d1, d2);
}
```

d1 = 10.1, d2 = 20.1

d1 = 20.2, d2 = 10.1

Now we can get more than one value from a function

- Write a function to compute the roots of quadratic equation ax^2+bx+c=0. How to return two roots?

```
void comproots(int a,int b,int c,
         double *dptr1, double *dptr2)
{
 *dptr1 = (-b - sqrt(b*b-4*a*c))/(2.0*a);
 *dptr2 = (-b + sqrt(b*b-4*a*c))/(2.0*a);
 return;
}
```

Trace a program

```
main()
{
  int x, y;
  max_min(4, 3, 5, &x, &y);
  printf(" First: %d  %d", x, y);
  max_min(x, y, 2, &x, &y);
  printf("Second: %d  %d", x, y);
}
void max_min(int a, int b, int c,
             int *max, int *min)
{
   *max = a;
   *min = a;
   if (b > *max) *max = b;
   if (c > *max) *max = c;
   if (b < *min) *min = b;
   if (c < *min) *min = c;
   printf("F: %d  %d\n", max, *max);
}
```

name	Addr	Value
x	1	
y	2	
	3	
	4	
	5	
a	6	
b	7	
c	8	
max	9	
min	10	

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 don’t want to copy the value of variable
    - Value can be very large (array or struct)
  - We don’t 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
}
```

Passing Arrays to Functions

```
#include <stdio.h>
void display(int a)
{
    printf("%d",a);
}
int main()
{
   int c[] = {2,3,4};
   display(c[2]);  //Passing array element c[2] only
   return 0;
}
```

Arrays in Functions

```
int func1(int num[],      size){
                  int
}

int func2(int *num, int size){

}
```

- **func1** and **func2** know size from **int size**

Passing Arrays to Functions

```
#include <stdio.h>
float average(float a[], int count); // float average(float  *a, int count)
int main(){
    float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
    avg=average(c, 6);   /* Only name of array is passed as argument */
    printf("Average age=%.2f", avg);
    return 0;
 }
float average(float a[], int count){ // float average(float  *a
    int I;  float avg, sum = 0.0;
    for(I = 0;I < count; ++i)  sum += a[i];
    avg = (sum / 6);
    return avg;
}
```

Passing Arrays to Functions

```
#include <stdio.h>
void f1(float *a) { a[1] = 100;}
void f2(float a[]){ a[2] = 200;}
void printArray(float a[])
{
    int i = 0;
    for(; i < 6; i++) printf("%g ", a[i]);
}
int main(){
    float c[]={23.4, 55, 22.6, 3, 40.5, 18};
    f1(c);
    printArray(c);
    puts("");
    f2(c);
    printArray(c);
    return 0;
}
```

Passing Array By Reference

18	40.5	3	22.6	55	23.4

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 Function

```
#include <stdio.h>
void f1(float a){  printf("F1 %g", a);}
void f2(float a){  printf("F2 %g", a);}

int main(){
    void (*ptrF)(float a);
    ptrF = f1;
    ptrF(12.5);
    ptrF = f2;
    ptrF(12.5);
    getch();
    return 0;
}
```

A function pointer is defined in the same way as a function prototype, but the function name is replaced by the pointer name prefixed with an asterisk and encapsulated with parenthesis

Example:

```
int (*fptr)(int, char)
fptr = some_function;

(*ftpr)(3,'A');
some_function(3,'A');
```

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);
   f = f1;  // or  f = &f1;
   (*f)(10, 'a');

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

This is f1: x = 10, c = a

This is f2: n = 100, m = z

Pointer to function

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

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

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

int int_cmp_asc(void *i1, void *i2){
   int a = *((int *)i1);
   int b = *((int *)i2);
   return (a > b) ? 1 : (a == b) ? 0 : -1;
}

int int_cmp_dsc(void *i1, void *i2){
   int a = *((int *)i1);
   int b = *((int *)i2);
   return (a > b) ? -1 : (a == b) ? 0 : 1;
}
```

```
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;
```

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
- Dynamically allocated memory is determined at *runtime*

Dynamic Memory Allocation

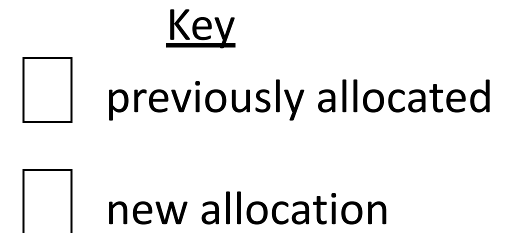
- Memory is allocated using the:
  - malloc function (memory allocation)
  - calloc function (cleared memory allocation)
- Memory is released using the:
  - free function
    - note: memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up
- The size of memory requested by malloc or calloc can be changed using the:
  - realloc function

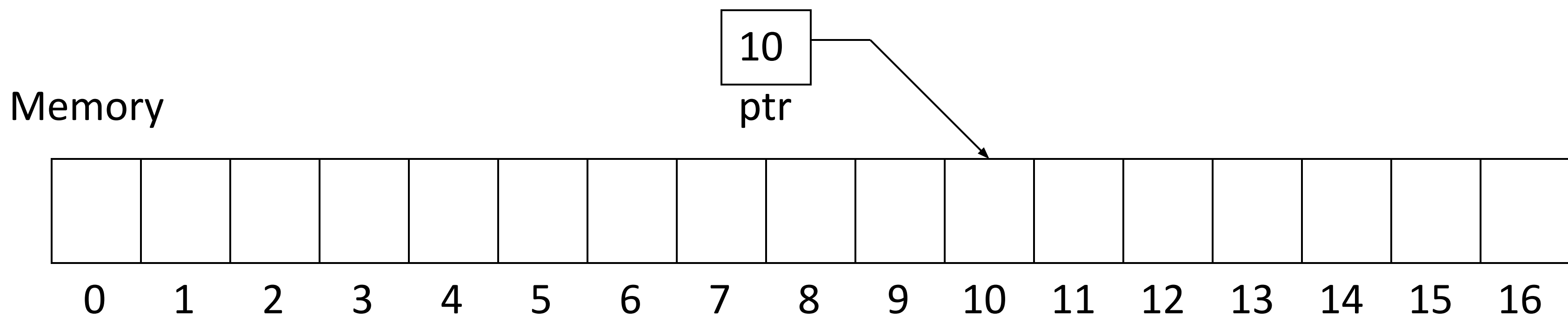
malloc

#include <stdlib.h>

- Prototype: ***void *malloc(size_t size);***
  - function returns the address of the first byte
  - programmers responsibility to not lose the pointer
- Example:

```
int *ptr;
ptr = (int *)malloc(sizeof(int));  // new allocation
```





calloc

- Memory allocation by **`calloc`**

```
#include <stdlib.h>

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

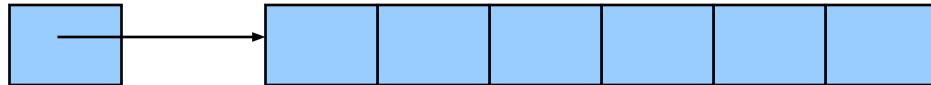
- **`void *`** is generic pointer, it can be converted to every pointer type

- Initializes allocated memory to zero

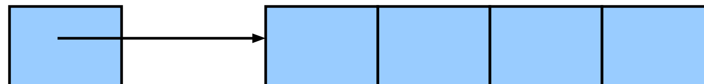
- If memory is not available **`calloc`** returns **`NULL`**

Example of malloc and calloc

```
int n = 6, m = 4;
double *x;
int *p;
```




```
/*  Allocate memory for 6 doubles.  */
x = (double *)malloc(n*sizeof(double));
```




```
/*  Allocate memory for 4 integers.  */
p = (int *)calloc(m,sizeof(int));
```

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));
```

malloc and calloc

- Both functions return a pointer to the newly allocated memory
- If memory can not be allocated, the value returned will be a **NULL** value
- The pointer returned by these functions is declared to be a **void pointer**
- A cast operator should be used with the returned pointer value to coerce it to the proper pointer type
- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

malloc vs. calloc

- The number of arguments:
  - malloc() takes a single argument (memory required in bytes), while calloc() needs two arguments.
- Initialization:
  - malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to ZERO.

Free

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

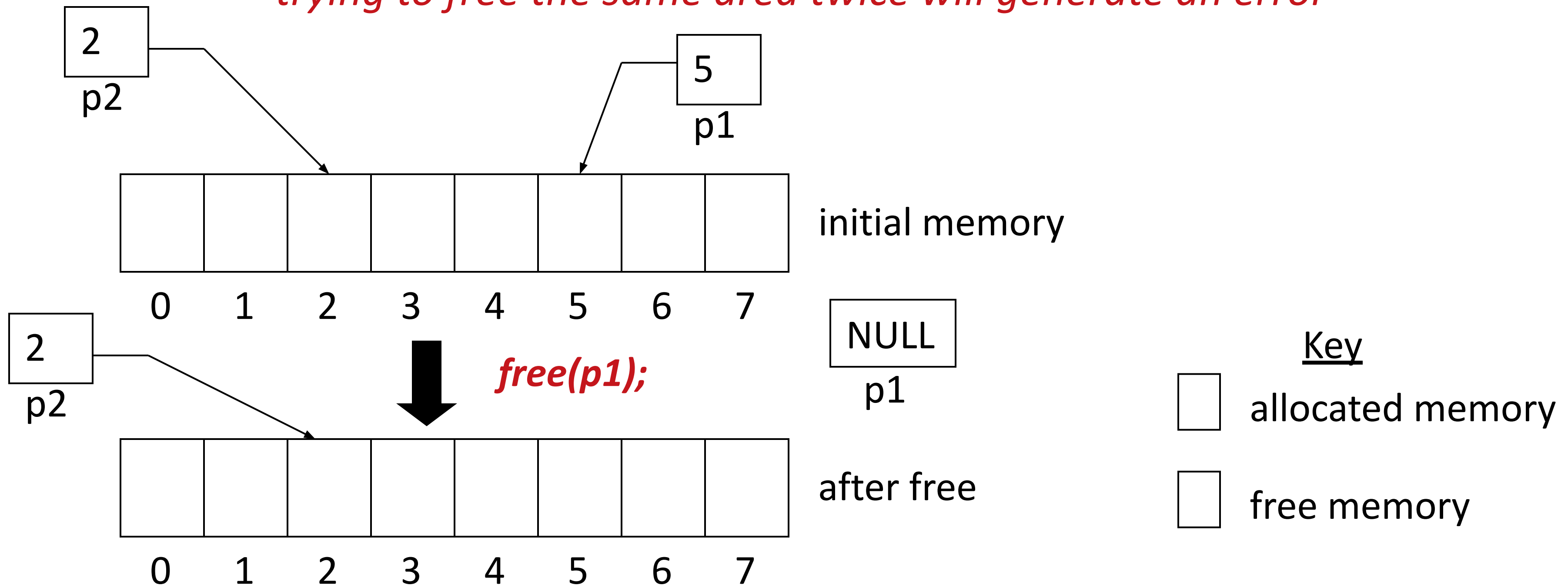
- In dynamic memory allocation, we must free the allocated memory

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

free

#include <stdlib.h>

- Prototype: ***void free(void *ptr)***
  - releases the area pointed to by ptr
  - ptr must not be null
    - *trying to free the same area twice will generate an error*



```
#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                work here */
     }    you  arr[i] = i;
   for(i = 0; i < n; i++)
         printf("%d\n",
         arr[i]);

   free(arr);
}  return 0;
```

هك n دريگيم ار،
n يا‌همانرب و ديلوت ار
هزادنا اب هيارآ دنكيم

دعب هظفاح ار دازآ

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

هک n و m دريگيم ار،
يلاهمانرب دعب و ديلوت ار
سيرتام دنكيم دازآ nxm

ار هظفاح

1. ارایه ای از اشاره گر ها اختصاص دهیم.
2. 2. هر سطر را با یک فراخوانی مجزا به malloc تخصیص دهیم.

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

realloc Example

```
float *nums;
int I;

nums = (float *) calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */

for (I = 0; I < 5; I++)
  nums[I] = 2.0 * I;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */

nums = (float *) realloc(nums,10 * sizeof(float));
/* An array of 10 floating point values is allocated, the
   first 5 floats from the old nums are copied as the first
   5 floats of the new nums, then the old nums is released
   */
```

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


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

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

printf("%d\n", *p);
p++;

printf("%d\n", *p);

p++;
printf("%d\n", *p);
p++;
printf("%d\n", *p);
```

0

0

500

100

0

0

Allocating Memory for a Pointer

- There is another way to allocate memory so the pointer can point to something:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *p;
    p = (int *) malloc( sizeof(int) );   /* Allocate 4 bytes */
    scanf("%d", p);
    printf("%d", *p);
    // ....
    free(p);      /* This returns the memory to the system*/
              /* Important !!! */
}
```

Allocating Memory for a Pointer

- You can use *malloc* and *free* to dynamically allocate and release the memory

```
int *p;
p = (int *) malloc(1000 * sizeof(int) );
for(i=0; i<1000; i++)
    p[i] = i;

p[999]=3;
free(p);
p[0]=5;        /* Error! */
```

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

void find_small(doubl *arr, int size)
e 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]);
}
```

ار نآ دادعت) ددع يدادعت هک
ياهمانرب
دريگب ار دوشيم مامت 1- اب هک
ميناديمن)

دنک پاچ ار نيگنايم زا رتکچوک ياهدادعا
و

```
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;
}
```

برنامه ای بنویسید که منوی زیر را به کاربر نشان دهد.

1: New Data

2: Show Data

3: Exit

ددع همانرب ،دنک دراو 1 n لوط هب يارآ ،دريگيم ار n دعب .دنكيم داجيا
ربراک رگا

درادیم هگن هیارآ رد ار اهنآ و دریگیم ربراک زا
nار ددع

دوشیم هداد ناشن هدشدراو تاعلاطا دنک دراو 2

ربراک رگا میوشیم جراخ همانرب زا دنک

```
#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]));
}
```

```
    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");
    }
  }
}
```