Pointers

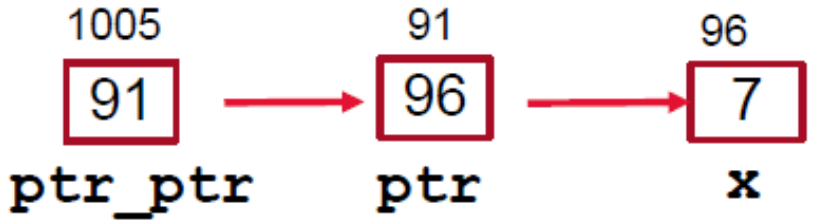
EECS 2031

Song Wang
wangsong@eecs.yorku.ca
eecs.yorku.ca/~wangsong/

Acknowledgement

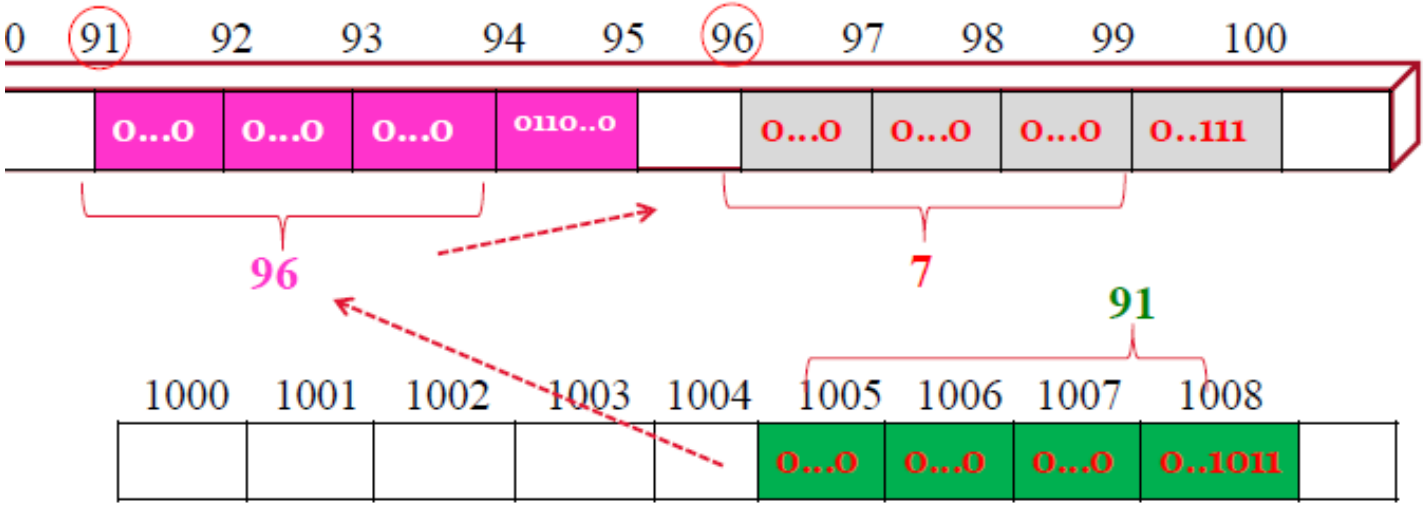
- Some of the covered materials are based on previous EECS2031 offerings:
  - Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

Pointer to pointers

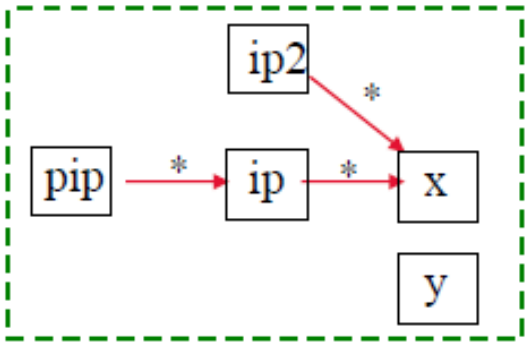



```
int x = 7;
int * ptr = &x;
```

```
int ** ptr_ptr      mnemonic:   // a pointer to pointer
ptr_ptr = &ptr;                 // ptr_ptr value is 91
** ptr_ptr = 20;                // ** access x, set x to 20
```



More Examples




```
int x = 1, y = 2;
int *ip, *ip2;

ip = &x;

int **pip;      // I am a pointer to pointer
pip = &ip;      // pip points to pointer ip

y = **pip;
(**pip)--;

ip2 = ip;
*ip2 += 10;
```

```
ip = &y;
(**pip)--;

printf("%d %d\n", x,  y);
```

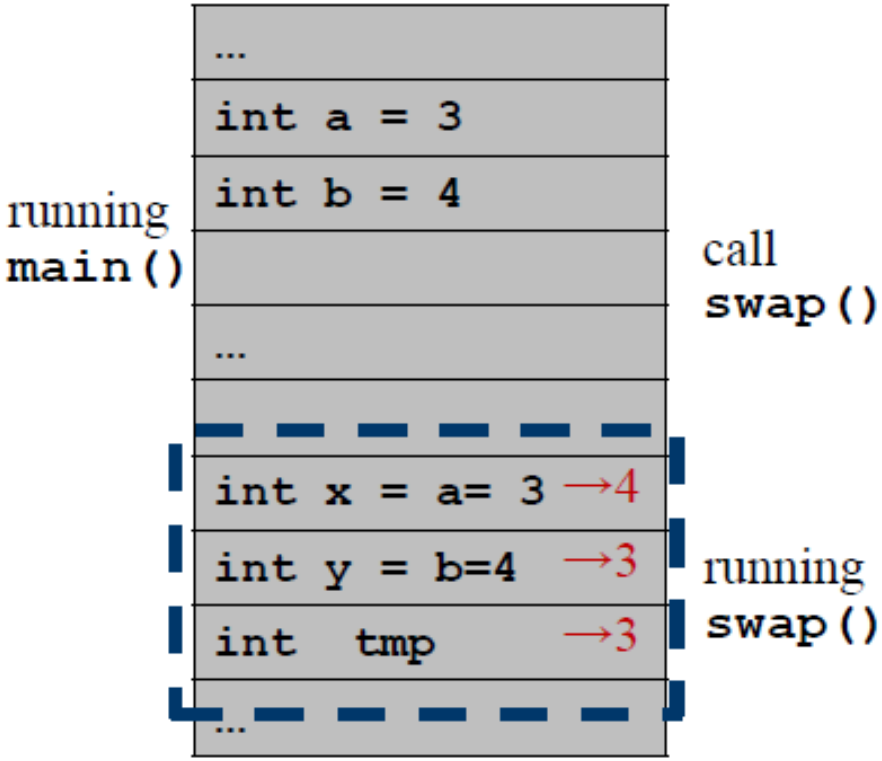
?

```c
#include <stdio.h>

int main () {

   int  var;
   int  *ptr;
   int  **pptr;
   int  *ptr2;
   int  **pptr3;

   var = 5;


   ptr = &var;


   pptr = &ptr;
   ptr2 = *pptr;
   pptr3 = &ptr2;

   /* take the values */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
   printf("Value available at *ptr2 = %d\n", *ptr2);
   printf("Value available at **pptr3 = %d\n", **pptr3);

   return 0;
}
```

Calling by Value

- In C, all functions are **called by value**
  - Value of the arguments are passed to functions, but not the arguments themselves (i.e., not ~~call by reference~~)

```
void swap (int x, int y)
{
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}
main(){
   int a=3, b=4;
   swap(a,b);
}
```



Pointers and function arguments

- In C, all functions are **called by value**
  - Value of the arguments are passed to functions, but not the arguments themselves (i.e., not ~~call by reference~~)
  - How to modify the arguments? `increment()` `swap()`
  - How to pass a structure such as array?

- Modify an actual argument by passing its address/pointer
  - Possibly modify passed arguments via their address!
  - Efficient.

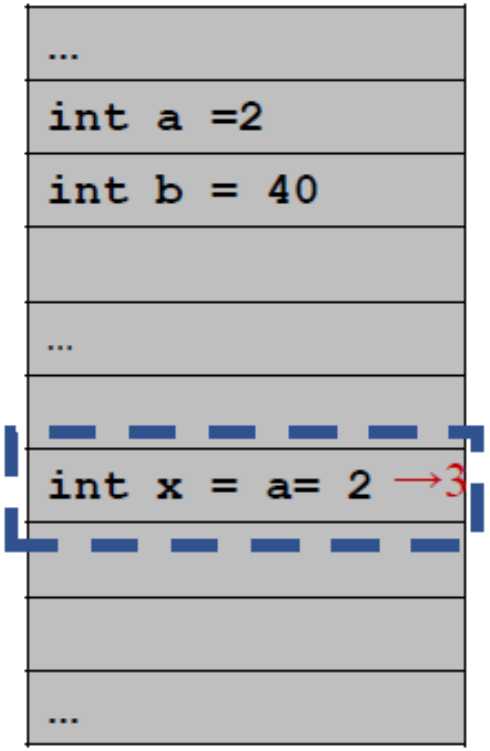
An example. Not working.


```
void increment(int x)
{

   x++;

}

void main( ) {
   int a=2;

   increment(a);
   printf("%d", a);
}
```

Pass by value !!!

x = a

running main()



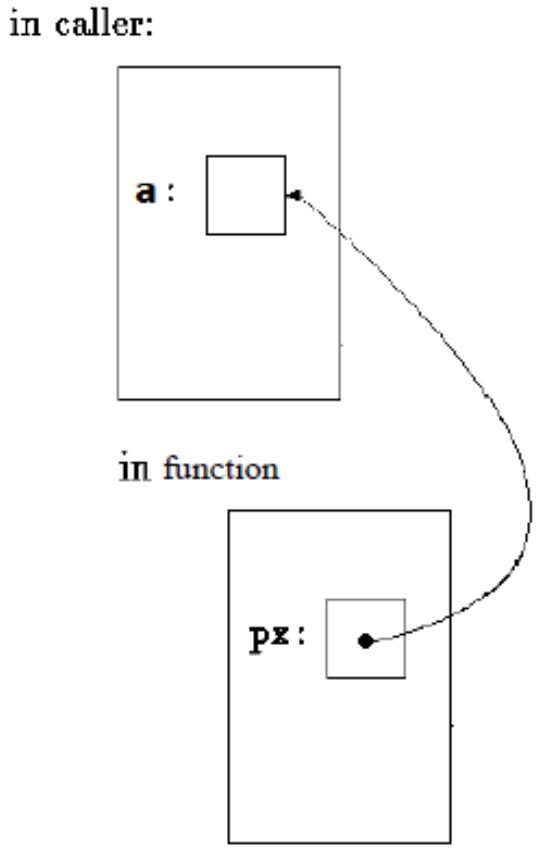
The Correct Version

I am expecting int pointers

```
void increment(int *px)
{



}

void main( ) {
   int a=2;

   increment(&a);
   printf("%d", a);
}
```

?

Pass address/pointer



Now understand scanf() -- more or less

```
int x=1;  int y = 2;
swap(&x,&y);  increment(&x,&y);
```

```
int x;
scanf ("%d", &x);
scanf ("%d  %d", &x, &y);
printf("%d", x);   // printf("%d", &x);
```

```
int x;
int *px = &x;
scanf("%d", px);
printf("%d",*px);
```

But why array name is used directly

```
scanf ("%d %s", &x, arrName)
fgets (arrName, 5,stdin);
```

explain shortly

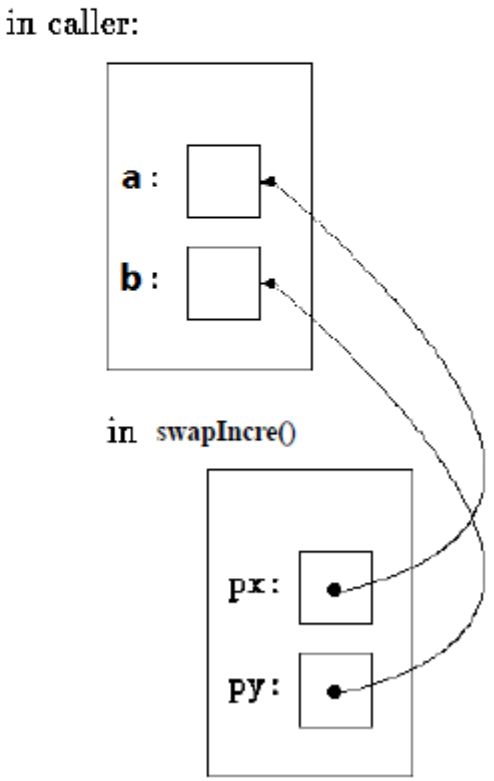
Another example

```
void swapIncre(int *px, int *py)
{
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
   increment( ? , ? );
}
```

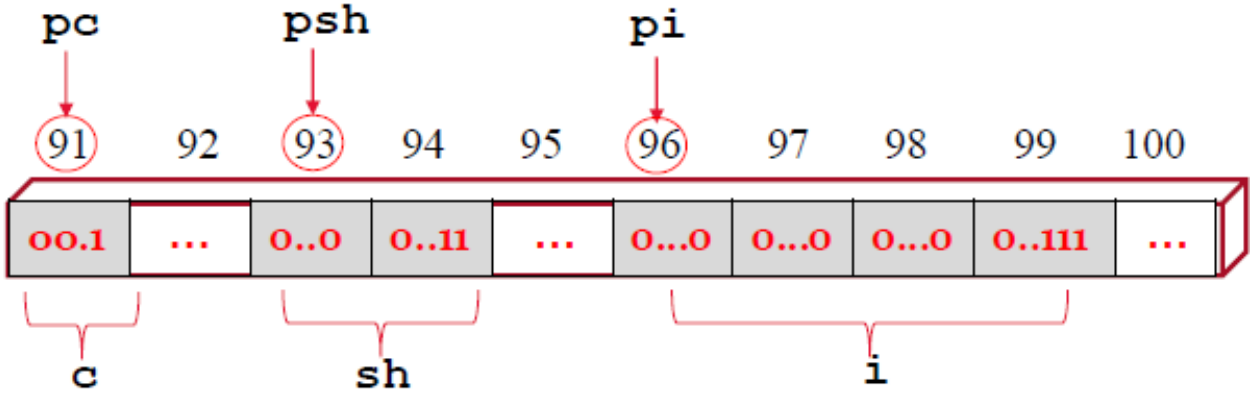
increment(,);

```
void increment(int *px2, int *py2)
{
    (*px2) ++;
    (*py2) += 10;
}
```

```
void main( ) {
    int a=2, b=40;

    swapIncre(&a, &b);
    printf("%d %d", a, b);
}
```



Pointer arithmetic



- So far deference * & Also limited math on a pointer
- Four arithmetic operators that can be applied

  + - ++ --

  Result is a pointer (address)

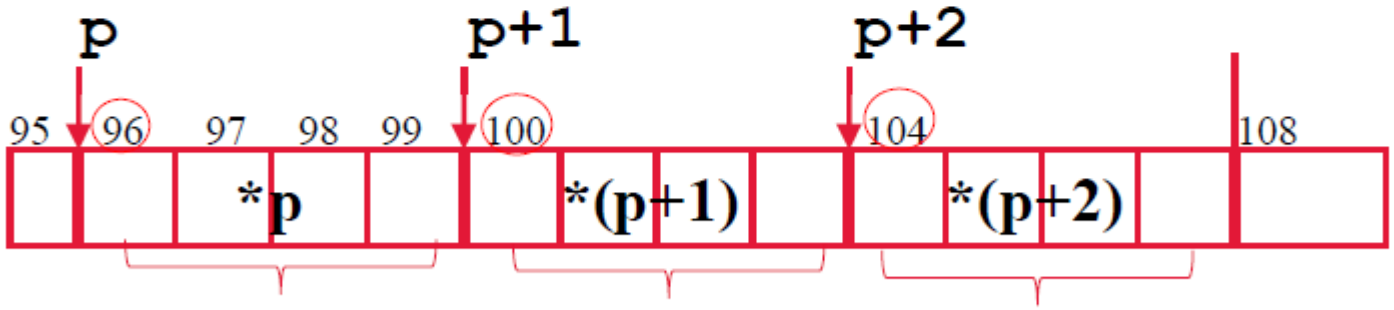
```
int* pi=&i;//96   char* pc=&c;//91  short* psh=&sh;//93

pi  + 1    97?    pi + 2  98?
psh + 1    94?    psh + 3  96?
pi++     pi = pi+1;
pc++  psh++
```

?

Pointer arithmetic –scaled

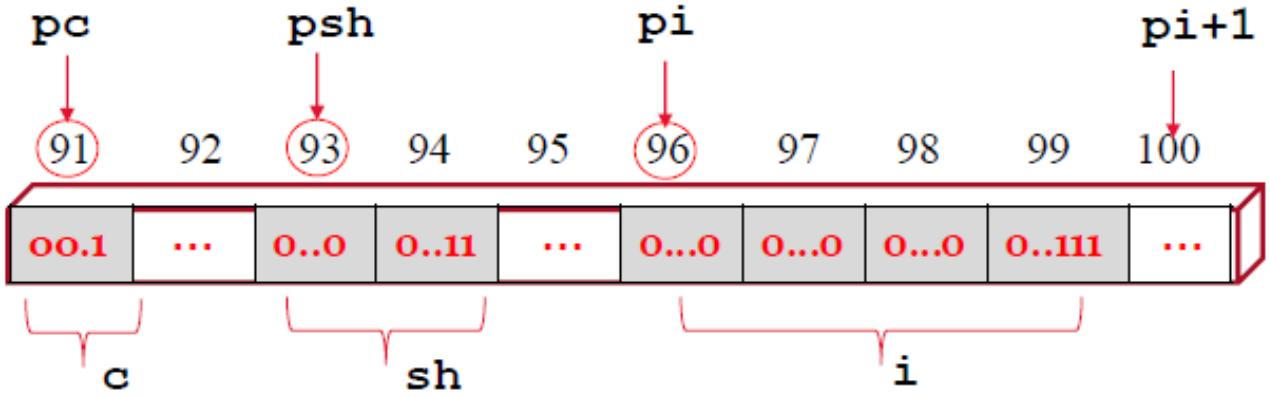
- Incrementing / decrementing a pointer by *n* moves it *n* units bytes
  p ± *n* ➔ p ± *n* × unit byte
  - value of a "unit" is based upon the size of the pointee type
    - p ± *n* ➔ p ± *n* × pointee-type-size

  - If p points to an integer (4 bytes), value of unit is 4
    p + n advances by n*4 bytes:
    p + 1 = 96 + 1*4 = 100 p + 2 = 96 + 2*4 = 104



- Why would we need to move pointer? p+1; p++
- Why designed this way? *"p+1 is p+4"*

Pointer arithmetic – scaled. *"p+1 is p+4"*




```
int* pi=&i;//96     char *pc=&c;//91   short* psh=&sh;//93
```

```
pi + 1      address 96+ 1*4 =100
pi + 2      address 96+ 2*4 =104
```

?

!!!

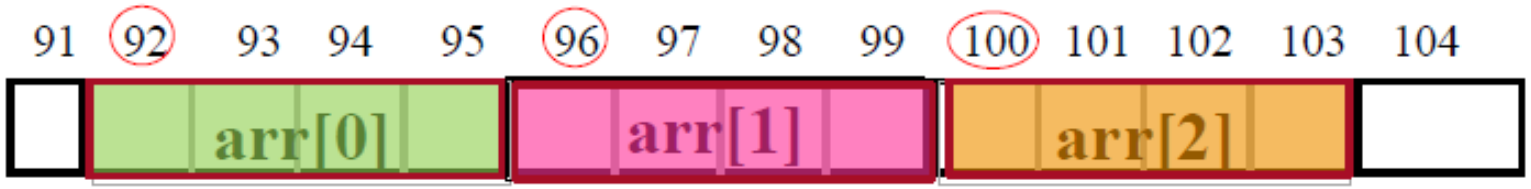
```
psh +1      address  93+ 1*2 =95
psh +3      address  93+ 3*2 =99   (other area)
```

assume
int 4 bytes
short 2 bytes
char 1 byte

```
pi++?  pc++?  psh++?
pi = 96 + 4     pc = 91 + 1    psh = 93 + 2
```

```
i = ++ * ptr      i= * ptr;   *ptr = *ptr + 1
i = * ++ ptr      ptr = ptr +1;    i = *ptr;

(* ptr) ++        * ptr;   * ptr = * ptr + 1
i = * ptr ++      i = * ptr;   ptr = ptr +1
```

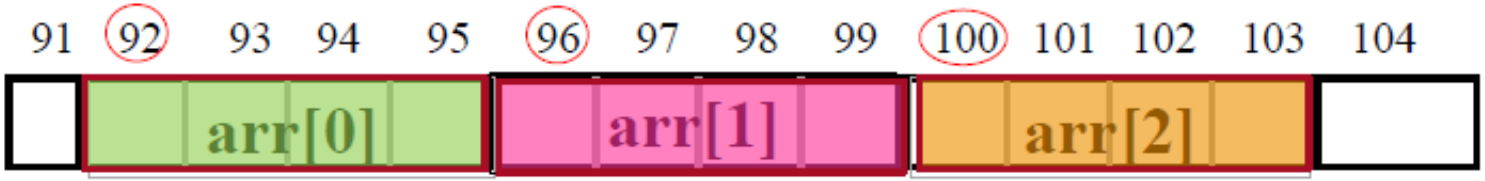
- Array members are next to each other in memory
  - `arr[0]` always occupies in the lowest address



arrayElementSize2021.c

```
int i[10], x;
float f[10];
double d[10];
char c[10];

main()
{

    /* Print the addresses of each array element. */
    printf("\n================================");

    for (x = 0; x < 10; x++)
        printf("\nElement [%d]: %p  %p %p %p", x, &c[x], &i[x],&f[x], &d[x]);

    printf("\n================================");
```

- Array members are next to each other in memory
  - `arr[0]` always occupies in the lowest address

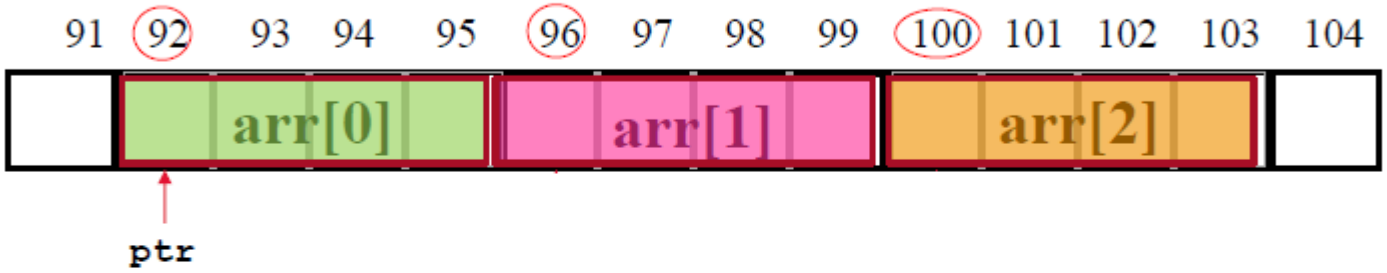



```
for (x = 0; x < 10; x++)
   printf("\nAddress [%d]: %p %p %p %p", x, &c[x], &i[x], &f[x], &d[x]);
```

```
indigo 322 % a.out
                char[]          int[]           float[]         double[]
================================================================================
Address [0]:    0x4040e8        0x404100        0x4040c0        0x404060
Address [1]:    0x4040e9        0x404104        0x4040c4        0x404068
Address [2]:    0x4040ea        0x404108        0x4040c8        0x404070
Address [3]:    0x4040eb        0x40410c        0x4040cc        0x404078
Address [4]:    0x4040ec        0x404110        0x4040d0        0x404080
Address [5]:    0x4040ed        0x404114        0x4040d4        0x404088
Address [6]:    0x4040ee        0x404118        0x4040d8        0x404090
Address [7]:    0x4040ef        0x40411c        0x4040dc        0x404098
Address [8]:    0x4040f0        0x404120        0x4040e0        0x4040a0
Address [9]:    0x4040f1        0x404124        0x4040e4        0x4040a8
================================================================================
```

Pointers and Arrays

- *"p+1 is p+4"*
- Array members are next to each other in memory
  - `arr[0]` always occupies in the lowest address




```
int arr[3]; int *ptr;
ptr = &arr[0];   // 92

ptr + 1
ptr + 2
*(ptr+2)
```

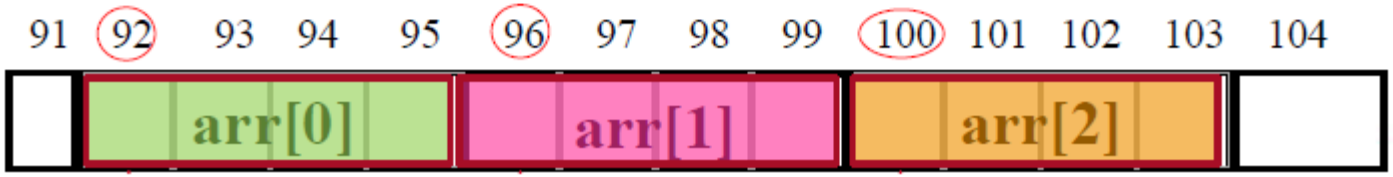
Pointers and Arrays

- There is special relationship between pointers and arrays
- When you use array, you are using pointers!

```
int i, arr[20], char c;
scanf("%d  %c  %s", &i, &c, arr); // &arr is wrong
```

- Identifier (name) of an array is equivalent to the address of its 1st element. **`arr == &arr[0]`**
  - Array name can be used 'like' a pointer. Follow pointer arithmetic rule!

```
arr + 1?
arr + 2?
```



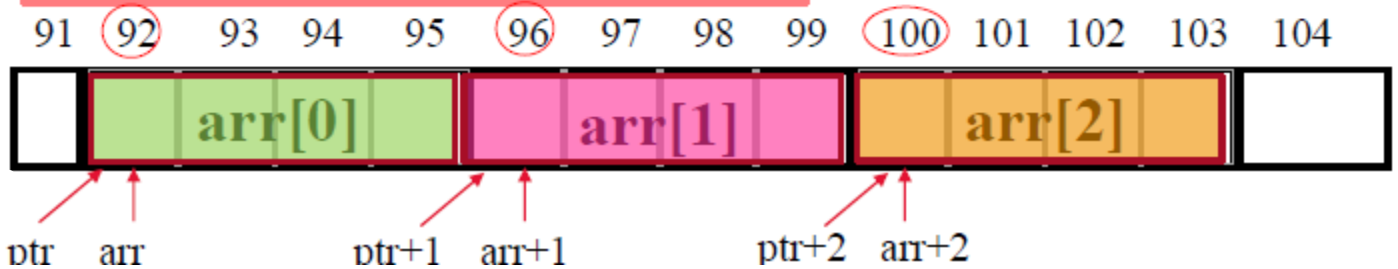
Pointers and Arrays

- There is special relationship between pointers and arrays
- Identifier (name) of an array is equivalent to the address of its 1st element. **arr == &arr[0]**

```
*arr == *(&arr[0]) == arr[0]
arr + i == &arr[i]
*(arr + i) == *(&arr[i]) == arr[i]
```

```
int arr[3];
int * ptr;
ptr = &arr[0];   // 92
```

```
ptr + i == &arr[i]
*(ptr + i) == arr[i]
```

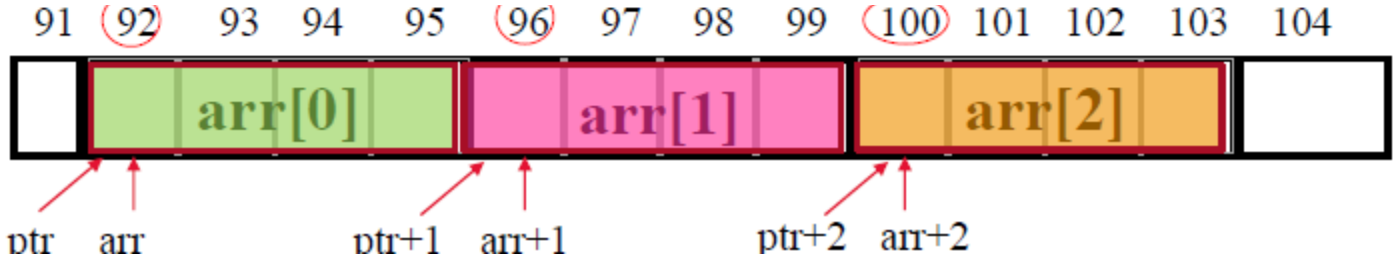


Pointers and Arrays

- There is special relationship between pointers and arrays
- Identifier (name) of an array is equivalent to the address of its 1st element. **`arr == &arr[0]`**

```
int arr[3]; int * ptr;
ptr = arr;    /*    ptr = &arr[0]   */
```

```
arr+i == &arr[i]
ptr+i == &arr[i]
```

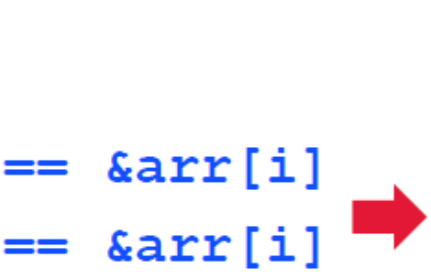


Pointers and Arrays

- There is special relationship between pointers and arrays
- Identifier (name) of an array is equivalent to the address of its 1st element. **`arr == &arr[0]`**

```
int arr[3]; int * ptr;
ptr = arr;    /*    ptr = &arr[0]   */
```

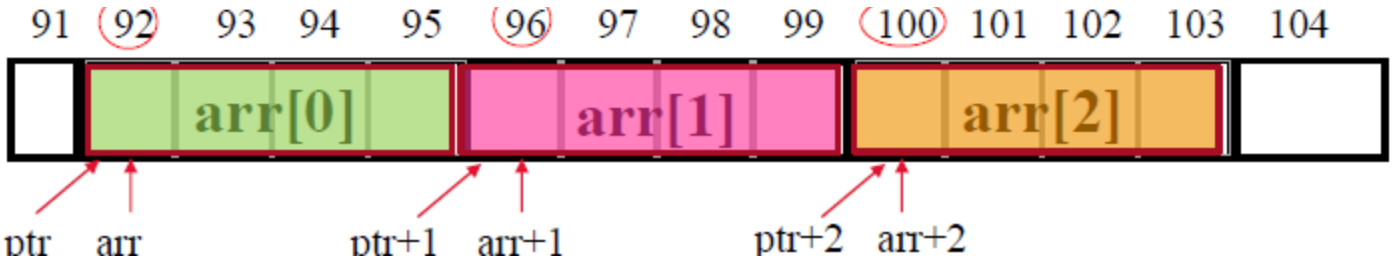
```
arr+i == &arr[i]
ptr+i == &arr[i]
```



```
arr[i]
*(ptr + i)
*(arr + i)
ptr[i];
```

Compiler converts arr[2] to *(arr+2)

equivalent



Attention: Array name can be used as a pointer, but is not a pointer variable!

```
int arr[20];
int * p = arr;
```

- **p** and **arr** are equivalent in that they have the same properties: **&arr[0]**

- Difference: **p** is a pointer variable, **arr** is a pointer constant
  - we could assign another value to **p**
  - **arr** will always point to the first of the 20 integer numbers of type int. Cannot change **arr** (point to somewhere else)

```
p = arr;   /*valid*/        arr = p;   /*invalid*/
p++;       /*valid*/        arr++;     /*invalid*/
```

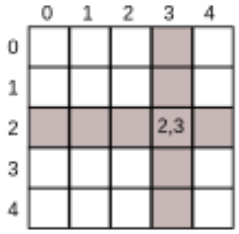

```
char arr[10] = "hello";  char c;
char * p;
p = arr;        // p=&arr[0]

arr = p;
arr = &i;
arr = arr +1;
arr++;

p++;
p = &c;
```

```
p = arr+2;
*(arr + 1)='x';
c = *(arr+2);
```

now points to others*/

```
strlen(arr);
strlen(p);
```

```
sizeof arr ?   10
sizeof p ?     8
```



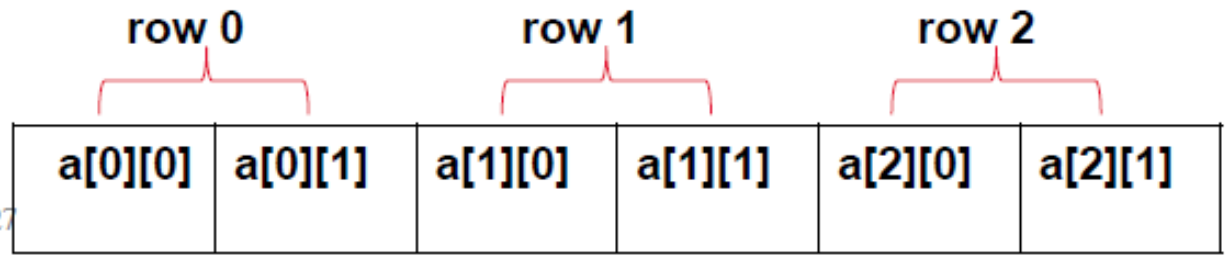
Multi-dimension array how are they stored

- `int arr1D[10];`

  - Size: type bytes * number of element
    - 4 * 10 = 40 bytes

- `int arr2D[3][2]`
  - Size: type bytes * row * column
    - 4 * 3 * 2 = 24 bytes

	col 0	col 1
row 0	a[0][0]	a[0][1]
row 1	a[1][0]	a[1][1]
row 2	a[2][0]	a[2][1]



Example

- **Find the minimum value in the given Array, use float variables**

float A[6] = {1.0, 0.2, 1.5, 0.5, 3.0, 2.0};

Example

- **Find the minimum value in the given Array, do not use float variables (use pointer to float)**

float A[6] = {1.0, 0.2, 1.5, 0.5, 3.0, 2.0};

Example

- **Find the minimum value in the given Array, you are not allowed to use float variables**

```
float A[6] = {1.0, 0.2, 1.5, 0.5, 3.0, 2.0};

float *theMin = &(A[0]);
float *walker = &(A[1]);

while (walker < &(A[6])) {
  if (*walker < *theMin)
    theMin = walker;
  walker = walker + 1;
}
printf("%.1f\n",*theMin);
```