Lab 3. Pointers

Programming Lab (Using C)

XU Silei

slxu@cse.cuhk.edu.hk

Outline

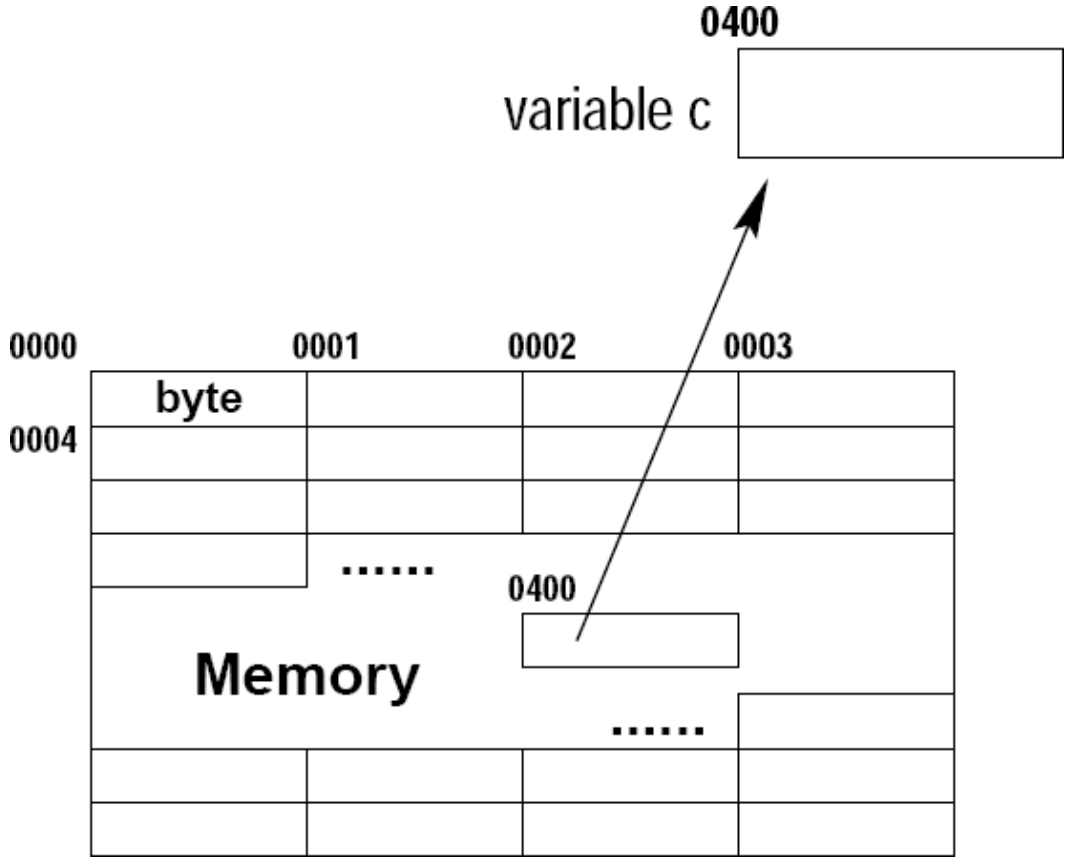
- What is Pointer
  - Memory Address & Pointers
- How to use Pointers
  - Pointers Assignments
  - Call-by-Value & Call-by-Address Functions
- When Pointers meet Arrays
  - Arrays & Memory Address
  - Passing Arrays to Functions

Outline

- What is Pointer
  - Memory Address & Pointers
- How to use Pointers
  - Pointers Assignments
  - Call-by-Value & Call-by-Address Functions
- When Pointers meet Arrays
  - Arrays & Memory Address
  - Passing Arrays to Functions

What is Pointer

- Where do variables store
  - Memory (Stack, Heap, Static, …)
- Address in memory
  - Byte-addressable
  - Each address identifying a single 8 bit of storage (larger data reside in multiple bytes occupying a sequence of consecutive address)
  - E.g., Memory address of the variable C is 0400.



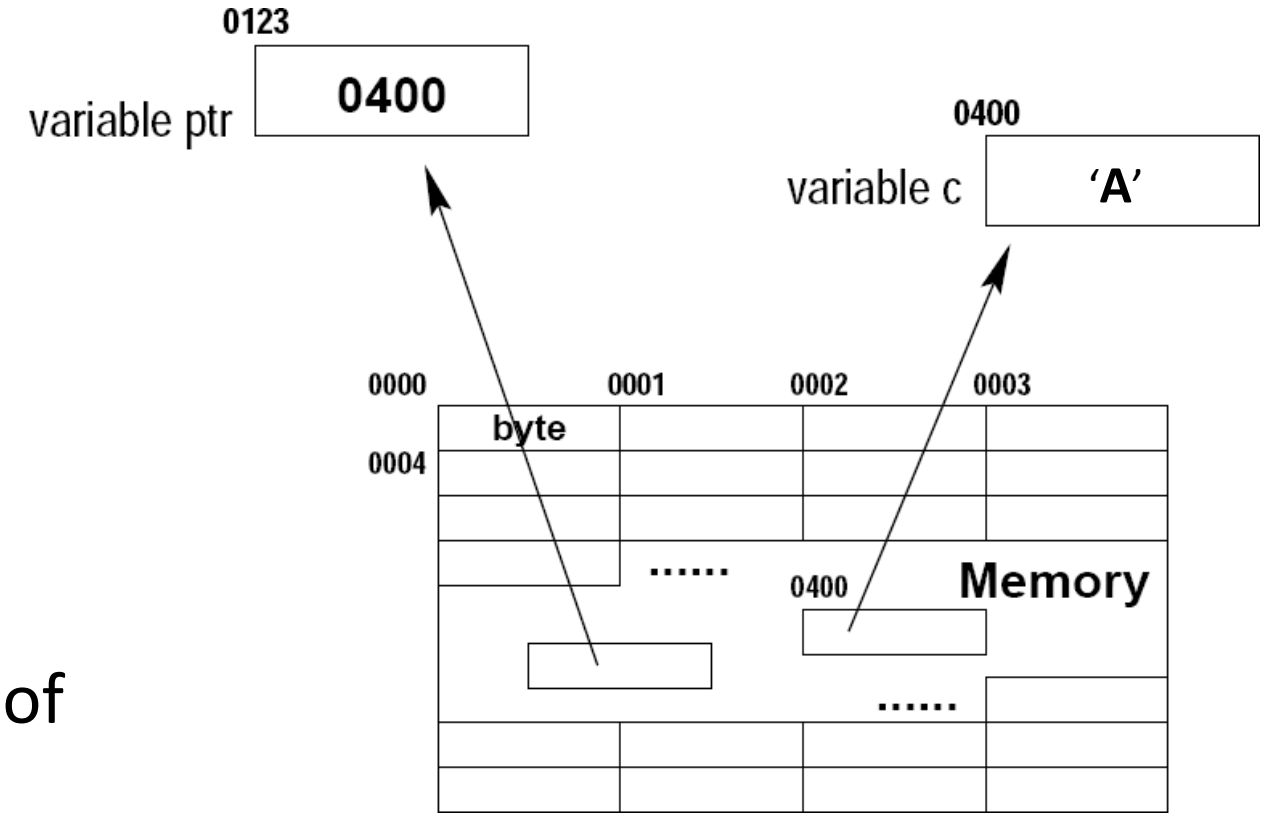
What is Pointer

```
char c;
scanf("%c", &c);
```

- &c
  - denotes the address of the variable c in memory, say, 0400.
- scanf(“%c”, &c);
  - The input character, say ‘A’, will be stored in the memory address of c
  - So variable c is ‘A’ after the statement;

Definition of Pointers

```
char c;
char* ptr;
ptr = &c;
```



- ptr is another variable, storing the memory address of the variable c

Definition of Pointers

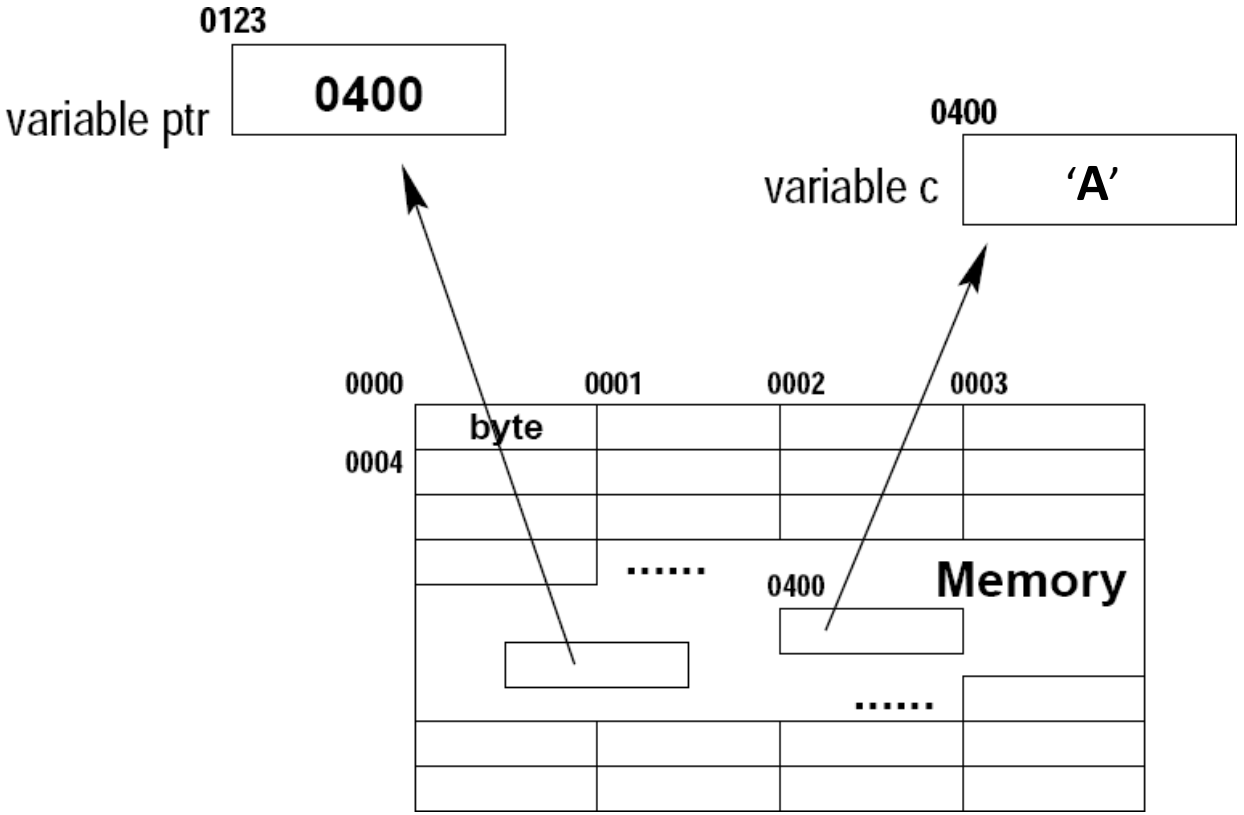
- Declare the variable ptr
  - char *ptr; OR char* ptr;

- Variable name is ptr, Not *ptr
  - The * denotes that ptr is a pointer

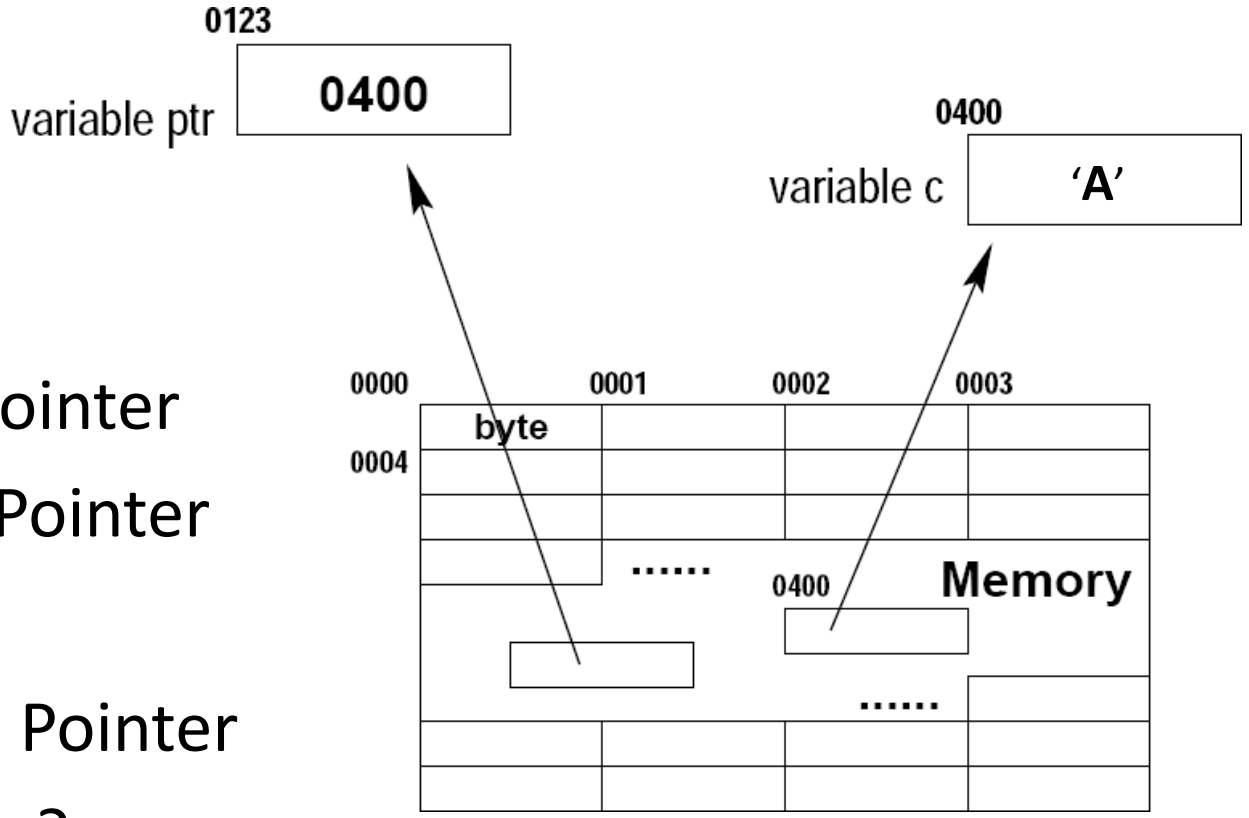
- The type of ptr is ***pointer to char***, or ***char pointer***
  - Variable ptr is to be used to store the memory address of another char variable

Definition of Pointers

- c: ‘A’
- &c: 0400
- ptr: 0400
- *ptr: ‘A’
- &ptr: 0123



Definition of Pointers



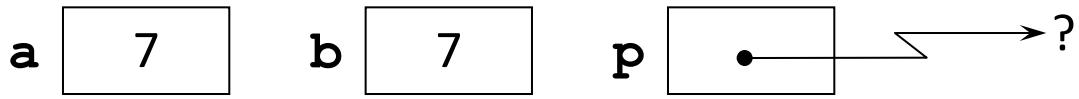
- c: 'A'
- &c: 0400 ----- Pointer
- ptr: 0400 ------ Pointer
- *ptr: 'A'
- &ptr: 0123 ----- Pointer
- Q: *&ptr? &*ptr?

```c
#include <stdio.h>

int main(void)
{
    int a = 7, b = 7, *p;
    p = &a;
    printf("a = %d\n", *p);
    *p = 3;
    printf("a = %d\n", a);
    p = &b;
    *p = 2 * *p - a;
    printf("b = %d\n", b);
    return (0);
}
```

```
a = 7
a = 3
b = 11
```

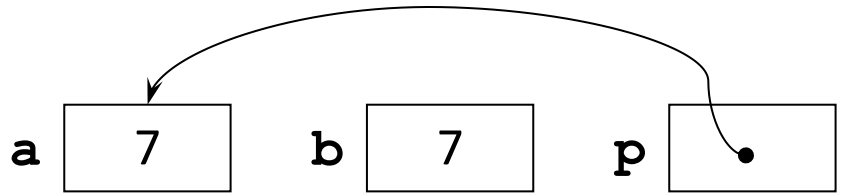
```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int a = 7, b = 7, *p;
6       p = &a;
7       printf("a = %d\n", *p);
8       *p = 3;
9       printf("a = %d\n", a);
10      p = &b;
11      *p = 2 * *p - a;
12      printf("b = %d\n", b);
13      return (0);
14  }
```



Declare the two integer variables **a** and **b**, and the integer pointer variable **p**.

```
a = 7
a = 3
b = 11
```

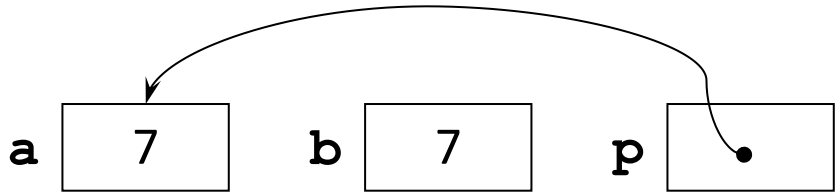
```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int a = 7, b = 7, *p;
6       p = &a;
7       printf("a = %d\n", *p);
8       *p = 3;
9       printf("a = %d\n", a);
10      p = &b;
11      *p = 2 * *p - a;
12      printf("b = %d\n", b);
13      return (0);
14  }
```



Store the address of variable **a** into pointer **p**.

```
a = 7
a = 3
b = 11
```

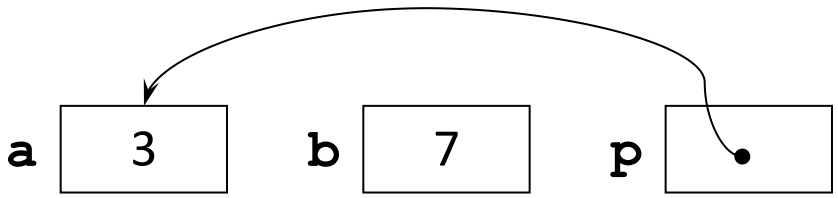
```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int a = 7, b = 7, *p;
6       p = &a;
7       printf("a = %d\n", *p);
8       *p = 3;
9       printf("a = %d\n", a);
10      p = &b;
11      *p = 2 * *p - a;
12      printf("b = %d\n", b);
13      return (0);
14  }
```



- Now, we can use pointer **p** to access the value of variable **a** by using the indirection (or called the dereferencing) operator ***.**
- ***p** refers to the value of the variable to which **p** points, i.e., **a**

```
a = 7
a = 3
b = 11
```

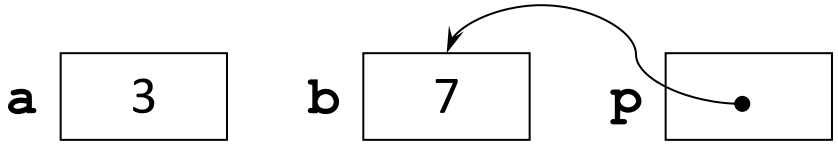
```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int a = 7, b = 7, *p;
6       p = &a;
7       printf("a = %d\n", *p);
8       *p = 3;
9       printf("a = %d\n", a);
10      p = &b;
11      *p = 2 * *p - a;
12      printf("b = %d\n", b);
13      return (0);
14  }
```



When ***p** appears on the LHS of an assignment, it means the value on the RHS is to be written onto the memory location to which **p** points.

```
a = 7
a = 3
b = 11
```

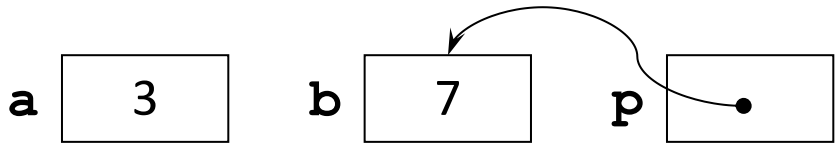
```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int a = 7, b = 7, *p;
6       p = &a;
7       printf("a = %d\n", *p);
8       *p = 3;
9       printf("a = %d\n", a);
10      p = &b;
11      *p = 2 * *p - a;
12      printf("b = %d\n", b);
13      return (0);
14  }
```



Update pointer **p** to store the address of variable **b**.

```
a = 7
a = 3
b = 11
```

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int a = 7, b = 7, *p;
6       p = &a;
7       printf("a = %d\n", *p);
8       *p = 3;
9       printf("a = %d\n", a);
10      p = &b;
11      *p = 2 * *p - a;
12      printf("b = %d\n", b);
13      return (0);
14  }
```



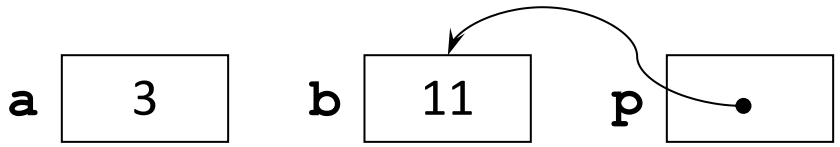
The actual calculation performed is $2 \times 7 - 3$.

Re-read line 11 as:

```
b = 2 * b - a;
```

```
a = 7
a = 3
b = 11
```

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int a = 7, b = 7, *p;
6       p = &a;
7       printf("a = %d\n", *p);
8       *p = 3;
9       printf("a = %d\n", a);
10      p = &b;
11      *p = 2 * *p - a;
12      printf("b = %d\n", b);
13      return (0);
14  }
```



The actual calculation yields eleven.

Re-read line 11 as:

```
b = 2 * b - a;
```

```
a = 7
a = 3
b = 11
```

Outline

- What is Pointer
  - Memory Address & Pointers
- How to use Pointers
  - Pointers Assignments
  - Call-by-Value & Call-by-Address Functions
- When Pointers meet Arrays
  - Arrays & Memory Address
  - Passing Arrays to Functions

How to use Pointers

- Pointers Assignments
  - One pointer can be assigned to another only when both pointers have the same type.

```
int  x, *ptr1, *ptr2;
char c, *ptr3;
ptr1 = &x;      /* valid */
ptr2 = ptr1;    /* valid */
ptr3 = &c;      /* valid */
ptr2 = ptr3;    /* invalid */
```

How to use Pointers

- The void Pointer
  - Pointer assignment is allowed when one of the operands is of type "***pointer to void***"
  - We treat void* as a generic/ universal pointer type

```
int  x, *xptr;
char *cptr;
void *vptr;
xptr = &x;
vptr = xptr;    /* valid */
cptr = vptr;    /* valid */
```

Call-by-Value & Call-by-Address Functions

- Call-by-Value Functions
  - Input: 4 5
  - Output: Result = 9

```c
#include <stdio.h>
int plus(int, int);
int main(){
    int a, b, res;
    scanf("%d %d", &a, &b);
    res = plus(a, b);
    printf("Result = %d", res);
    return 0;
}
int plus(int x, int y){
    return x + y;
}
```

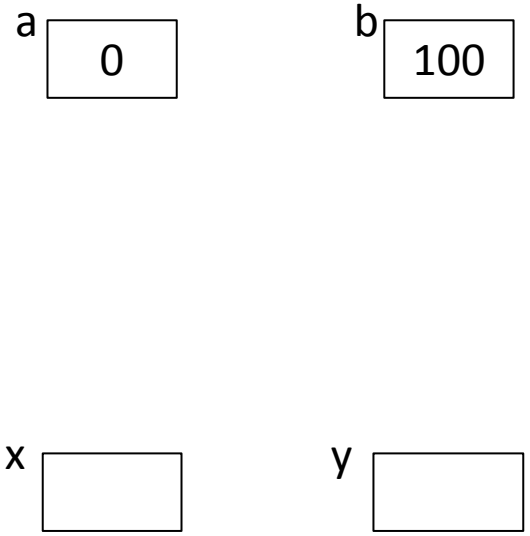
Call-by-Value Functions

- Another example
  - What's the output?
  - a = 0, b = 100
  - Why?

```c
#include <stdio.h>
void swap(int, int);
int main(){
    int a = 0, b = 100;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int x, int y){
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Call-by-Value Functions

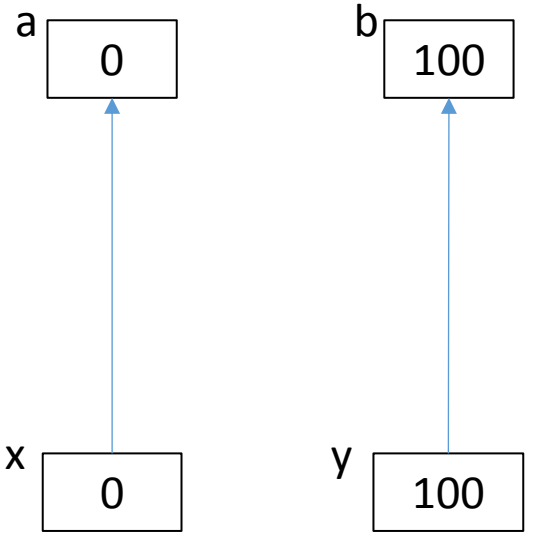
- Another example




```c
#include <stdio.h>
void swap(int, int);
int main(){
    int a = 0, b = 100;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int x, int y){
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Call-by-Value Functions

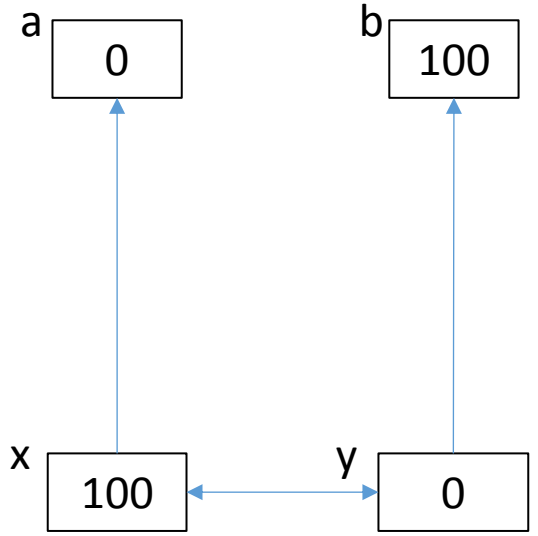
- Another example




```c
#include <stdio.h>
void swap(int, int);
int main(){
    int a = 0, b = 100;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int x, int y){
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Call-by-Value Functions

- Another example




```c
#include <stdio.h>
void swap(int, int);
int main(){
    int a = 0, b = 100;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int x, int y){
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Call-by-Value Functions

- The formal parameters x and y (the local variables of swap()) are created when the function is entered, and are destroyed (free from the memory) when the function returns/ terminates

- No matter how the value of the formal parameters changes, the variable in the calling environment (main() in this example) are never changed

Call-by-Address Functions

```c
#include <stdio.h>
void swap(int*, int*);
int main(){
    int a = 0, b = 100;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int* x, int* y){
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

```c
#include <stdio.h>
void swap(int, int);
int main(){
    int a = 0, b = 100;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int x, int y){
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

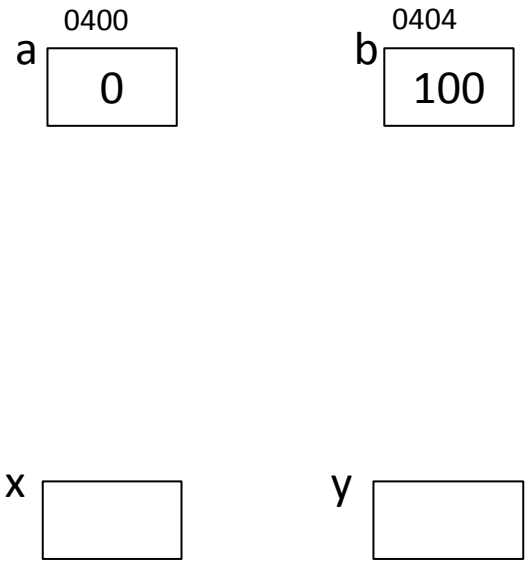
Call-by-Address Functions

- Another example
  - What's the output?
  - a = 100, b = 0
  - Success !!!

```c
#include <stdio.h>
void swap(int*, int*);
int main(){
    int a = 0, b = 100;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int* x, int* y){
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Call-by-Address Functions

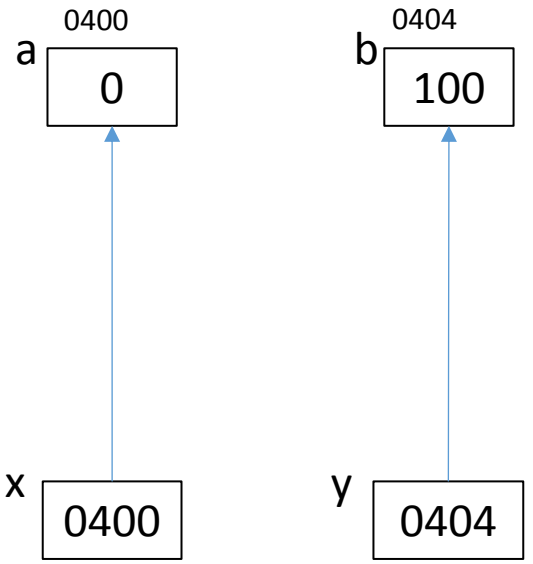
- Another example




```c
#include <stdio.h>
void swap(int*, int*);
int main(){
    int a = 0, b = 100;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int* x, int* y){
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Call-by-Address Functions

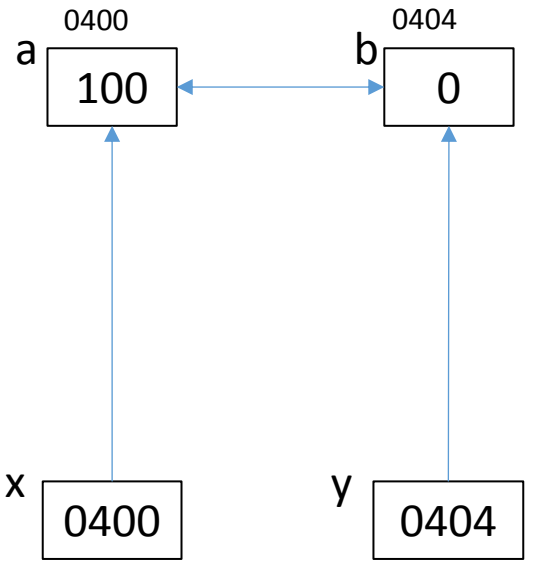
- Another example




```c
#include <stdio.h>
void swap(int*, int*);
int main(){
    int a = 0, b = 100;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int* x, int* y){
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Call-by-Address Functions

- Another example




```c
#include <stdio.h>
void swap(int*, int*);
int main(){
    int a = 0, b = 100;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
void swap(int* x, int* y){
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Call-by-Address Functions

- The address of the actual parameters (the variables a and b in main()) are passed to the function

- The functions applies dereference operator * on the received addresses (stored in pointer variable x and y) to access/ modify the "remote" variables in the calling environment (main() in this example) indirectly.

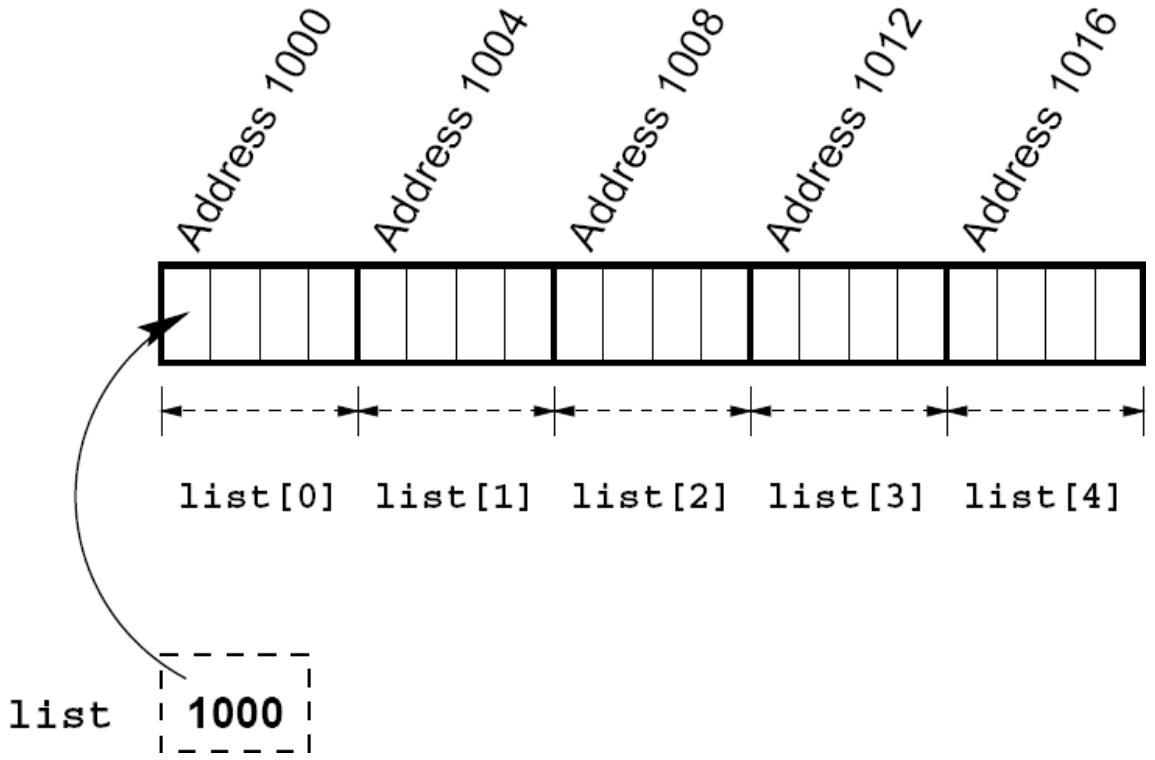
- Therefore, dereference is also called indirection

Outline

- What is Pointer
  - Memory Address & Pointers
- How to use Pointers
  - Pointers Assignments
  - Call-by-Value & Call-by-Address Functions
- When Pointers meet Arrays
  - Arrays & Memory Address
  - Passing Arrays to Functions

When Pointers Meet Arrays

- One-Dimensional Arrays
  - E.g., int list[5];
  - Assumption
    - 4-byte integers
    - the array is stored at memory address 1000 onwards



One-Dimensional Arrays

- An array name without index denotes the address of the first element of the array
  - list == &list[0] (base address)
  - i.e., list == 1000
  - *list == *&list[0] == list[0]
  - list + 1 == &list[1]
  - *(list +1) == *&list[1] == list[1]

- Starting address of list[i] is given by
  - &list[i] = base address + sizeof(int) $\times$ i = 1000 + 4i

One-Dimensional Arrays

- String: One-Dimensional Character Arrays
  - char s[] = “abc”;
  - char s[3] = {‘a’, ‘b’, ‘c’};
  - char *s = “abc”;

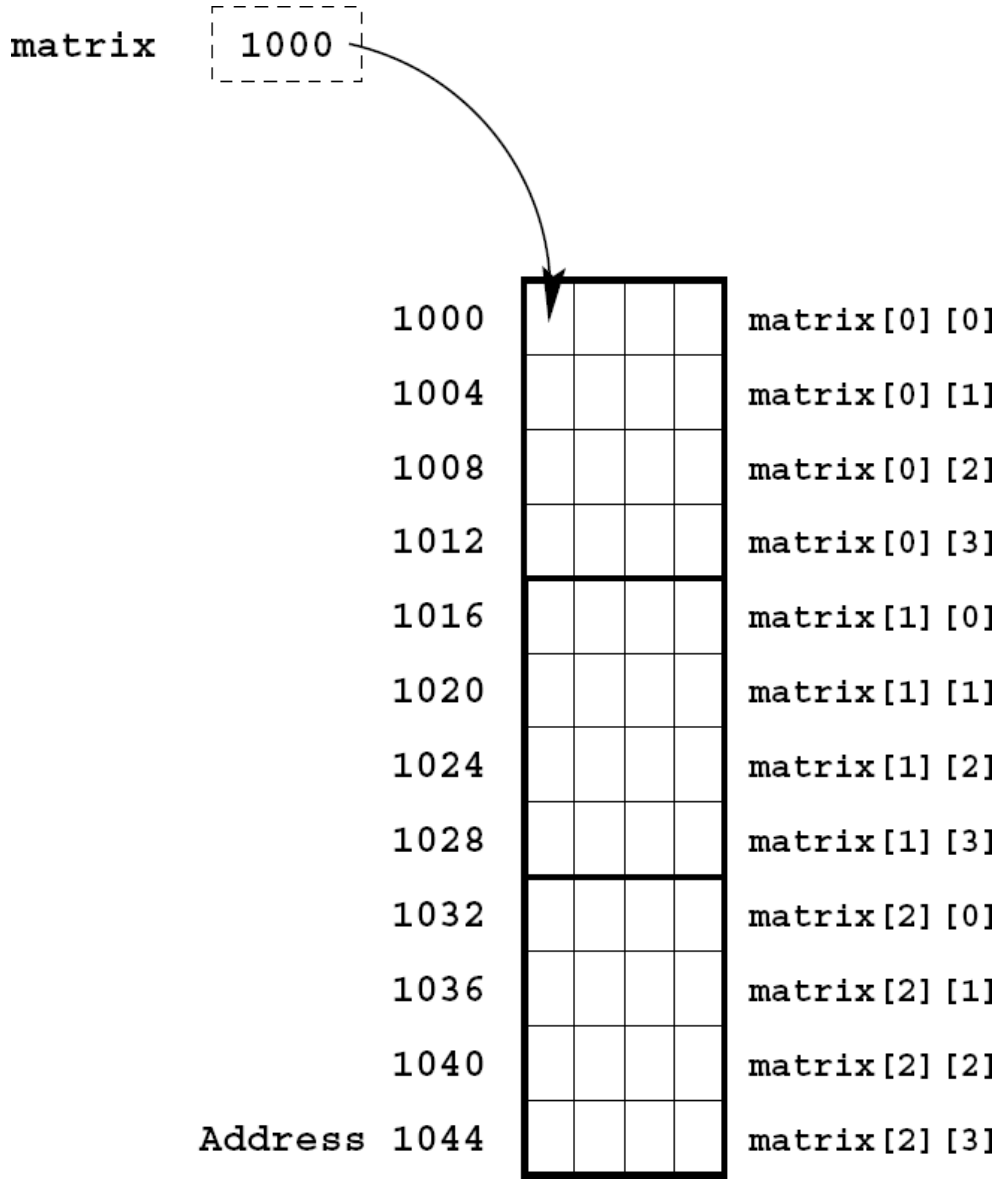
One-Dimensional Arrays

- String: One-Dimensional Character Arrays
  - char s[] = “abc”;
  - char s[3] = {‘a’, ‘b’, ‘c’};
  - char *s = “abc”; ---- Read Only !
  - Place “abc” in the read-only part of the memory, and make s a pointer to that.
  - *s == ‘a’;

  - char s[] = “abc”;
  - char *t = &s; ----- Writable !

Two-Dimensional Arrays

- int matrix[3][4];
  - An array name without index denotes the address of the first element of the array
    - matrix == &matrix[0][0]
  - For int matrix[ROW][COL], starting address of matrix[i][j] is given by
    - Base address + sizeof(int) × (COL × i + j)
    - E.g. &matrix[2][3] = 1000 + 4 × (4 × 2 + 3) =1044



Passing Arrays to Functions

```c
#include <stdio.h>
void times2(int []);
int main(){
    int i, list[]={1,2,3,4,5};
    times2(list);
    for(i = 0; i < 5; ++i)
        printf("list[%d] = %d\n", i, list[i]);
    return 0;
}
void times2(int a[]){
    int i;
    for(i = 0; i < 5; ++i)
        a[i] = a[i] * 2;
}
```

- Output
  - list[0] = 2
  - list[1] = 4
  - list[2] = 6
  - list[3] = 8
  - list[4] = 10

Passing 1-D Arrays to Functions

- In the calling environment main()
  - The name of the array is used as the parameter
  - The bracket pair [ ] must be omitted

- In the called function times2()
  - The bracket pair is required, which is to tell the complier that the parameter of this function is an array
  - The array size can be omitted

Passing Multi-D Arrays to Functions

- In the calling environment
  - The name of the array is used as the parameter
  - The bracket pairs [ ] [ ] … [ ] must be omitted
  - E.g., times2 (array3d);

- In the called function
  - The bracket pairs [ ] [ ] … [ ] are required to tell the compiler that the parameter is a multi-dimensional array
  - The first dimension's size can be omitted, while the all the other dimension's size must be specified.
  - E.g., void times2 (array3d[ ][10][20]) { … }

Passing Arrays to Functions

- Call-by-Address Mechanism
  - In the calling environment main(), after calling times2(), elements in list[ ] are doubled.
  - Why?
  - The array list[ ] is not copied to the function times2(). No new array is created.
  - Instead, a[ ] in times2() refers to the same array list[ ] in main()
  - Thus modifying any element in a[ ] causes corresponding modifications on list[ ] actually.
  - Call-by-Address!

Passing Arrays to Functions

- An Alternative View

```c
#include <stdio.h>
void times2(int *);
int main(){
    int i, list[]={1,2,3,4,5};
    times2(list);
    for(i = 0; i < 5; ++i)
        printf("list[%d] = %d\n", i, list[i]);
    return 0;
}
void times2(int * ptr){
    int i;
    for(i = 0; i < 5; ++i){
        *ptr = *ptr * 2;
        ptr ++;
    }
}
```

Practice

- 1.Array Swap

  Let A[10] and B[10] be two arrays

  - int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  - int B[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

  Write a function to swap the elements of A and B and write another function to print the result

Practice

- 2. Reverse String

  Let S be a string:

  - char S[] = “nametag”;

  Write a function to reverse the elements in S and write another function to print S

  - E.g., “ABCDE” -> “EDCBA”

Thanks