



Course organization

- **Course introduction (Week 1)**
 - Code editor: Emacs (Week 2)
- **Part I: Introduction to C programming language (Week 3 - 12)**
 - Chapter 1: Overall Introduction (Week 3-4)
 - Chapter 2: Types, operators and expressions (Week 5)
 - Chapter 3: Control flow (Week 6)
 - Chapter 4: Functions and program structure (Week 7)
 - **Chapter 5: Pointers and arrays (Week 8)**
 - Chapter 6: Structures (Week 10)
 - Chapter 7: Input and Output (Week 11)
- **Part II: Skills others than programming languages (Week 12- 13)**
 - Debugging tools (Week 12)
 - Keeping projects documented and manageable (Week 13)
 - Source code managing (Week 13)
- **Part III: Reports from the battle field (student forum) (Week 14– 16)**
 - Presentation (week 14-15)
 - Demo (week 16)



上海交通大学

SHANGHAI JIAO TONG UNIVERSITY



Chapter 5. Pointers and Arrays

Chaochun Wei

Shanghai Jiao Tong University

Spring 2019



Contents

- ⑤ **5.1 Pointers and addresses**
- ⑤ **5.2 Pointers and function arguments**
- ⑤ **5.3 Pointers and arrays**
- ⑤ **5.4 Address arithmetic**
- ⑤ **5.5 Character pointers and functions**
- ⑤ **5.6 Pointer arrays, pointers to pointers**
- ⑤ **5.7 Multi-dimensional arrays**
- ⑤ **5.8 Initialization of pointer arrays**
- ⑤ **5.9 Pointers vs. multi-dimensional arrays**
- ⑤ **5.10 Command-line arguments**
- ⑤ 5.11 Pointers to functions
- ⑤ 5.12 Complicated declarations



4.18



5.1 Pointers and address

- For any type T, you may form a pointer type to T.
 - Pointers may reference a function or an object.
 - The value of a pointer is the address of the corresponding object or function
 - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators: ***** dereferences a pointer, **&** creates a pointer (reference to)
 - `int i = 3; int *j = &i;`
`*j = 4; printf("i = %d\n", i); // prints i = 4`
 - `int myfunc (int arg);`
`int (*fptr)(int) = myfunc;`
`i = fptr(4); // same as calling myfunc(4);`
- Generic pointers:
 - Traditional C used (`char *`)
 - Standard C uses (`void *`) – these can not be dereferenced or used in pointer arithmetic. So they help to reduce programming errors
- Null pointers: use **NULL** or **0**. *It is a good idea to always initialize pointers to NULL.*



5.1 Pointers and address

Step 1:

```
int main (int argc, argv) {  
    int x = 4;  
    int * y = &x;  
    ...  
}
```

Eg: chpt5.1_pointers.c

	Program Memory	Address
<i>x</i>	4	0x3dc
<i>y</i>	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
	NA	0x3cc
	NA	0x3c8
	NA	0x3c4
	NA	0x3c0
	NA	0x3bc
	NA	0x3b8
	NA	0x3b4
	NA	0x3b0



5.1 Pointers and address

Eg: chpt5.1_pointers_1.c

More example operations on pointers

```
int x = 1, y = 2;
```

```
int *ip;
```

```
ip = &x; /* ip points to x */
```

```
y = *ip; /* y = 1; */
```

```
*ip = *ip + 10; /* equivalent to x = x + 10; */
```

```
y = *ip + 1; /* note the difference with *ip += 1 */
```

```
++ *ip; /* similar to *ip += 1 and (*ip) ++ */
```

(See more details in hands-on experiment 5.1)



5.2 Pointers and function arguments

- Arguments are passed to functions by **value**.

```
/* function to swap the values of two variable */
```

```
int a = 1, b = 2;  
swap(a, b);
```

```
void swap (int x, int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

```
int a = 1, b = 2;  
swap(&a, &b);
```

```
void swap (int *x, int *y) {  
    int temp;  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

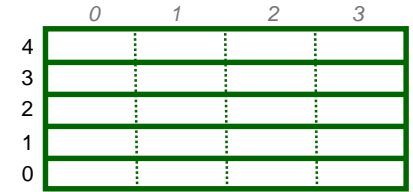
G!!!

5.3 Arrays and Pointers

- ⊙ A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

```
int x[5]; // an array of 5 4-byte ints.
```

- ⊙ All arrays begin with an index of 0



memory layout for array x

- ⊙ An array identifier is equivalent to a pointer that references the first element of the array

- ```
int x[5], *ptr;
```

```
ptr = &x[0]
```

 is equivalent to 

```
ptr = x;
```

- ⊙ Pointer arithmetic and arrays:

- ```
int x[5];
```

```
x[2]
```

 is the same as

```
*(x + 2)
```

, the compiler will assume you mean 2 objects beyond element x.



5.3 Arrays and pointers (continued I)

Step 1:

```
int main (int argc, argv) {  
    int x = 4;  
    int *y = &x;  
    int *z[4] = {NULL, NULL, NULL, NULL};  
    int a[4] = {1, 2, 3, 4};  
    ...  
}
```

Note: The compiler converts `z[1]` or `*(z+1)` to
Value at address (Address of `z` + `sizeof(int)`);

In C you would write the byte address as:
`(char *)z + sizeof(int);`

or letting the compiler do the work for you
`(int *)z + 1;`

	Program Memory	Address
<i>x</i>	4	0x3dc
<i>y</i>	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
<i>z[3]</i>	0	0x3cc
<i>z[2]</i>	0	0x3c8
<i>z[1]</i>	0	0x3c4
<i>z[0]</i>	0	0x3c0
<i>a[3]</i>	4	0x3bc
<i>a[2]</i>	3	0x3b8
<i>a[1]</i>	2	0x3b4
<i>a[0]</i>	1	0x3b0



5.3 Arrays and pointers (Continued II)

Step 1:

```
int main (int argc, argv) {  
    int x = 4;  
    int *y = &x;  
    int *z[4] = {NULL, NULL, NULL, NULL};  
    int a[4] = {1, 2, 3, 4};  
}
```

Step 2: Assign addresses to array z

```
z[0] = a; // same as &a[0];  
z[1] = a + 1; // same as &a[1];  
z[2] = a + 2; // same as &a[2];  
z[3] = a + 3; // same as &a[3];
```

	Program Memory	Address
<i>x</i>	4	0x3dc
<i>y</i>	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
<i>z[3]</i>	0x3bc	0x3cc
<i>z[2]</i>	0x3b8	0x3c8
<i>z[1]</i>	0x3b4	0x3c4
<i>z[0]</i>	0x3b0	0x3c0
<i>a[3]</i>	4	0x3bc
<i>a[2]</i>	3	0x3b8
<i>a[1]</i>	2	0x3b4
<i>a[0]</i>	1	0x3b0

More details see hands-on experiment 5.3



5.4 Address arithmetic

- **Pointers can do arithmetic operation**
 - **+ , - , ++**
 - **== , != , < , > , >= , etc**
- **Example: let p, and q be two pointers to an array**
 - **p++**
 - **p+= 1**
 - **p < q**
 - **p + n** /* next n object p points to */

See hands-on experiment 5.4 for more details



5.5 Character pointers and functions

- String constant: an array of characters, ending with '\0'

```
char *pmessage = "now is the time";
```

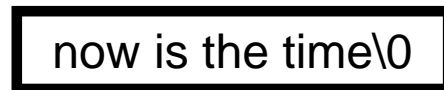
```
/* The pointer to the character array is  
assigned to pmessage. */
```

```
char amessage[ ] = "now is the time" ; /* an array */
```

pmessage:



amessage:





5.5 Character pointers and functions

- Assignment: is not a string copy operation

```
char *s = "this is a string", *t;
```

```
t = s ; /* this is not a string copy */
```

```
/* this copies to t the address that s points to */
```

- To copy a string, we need a loop

```
/* strcpy: copy t to s */
```

```
void strcpy(char *s, char *t) {
```

```
    while (( *s++ = *t++) != '\0') ;
```

```
}
```

See hands-on experiment 5.5 for more details.

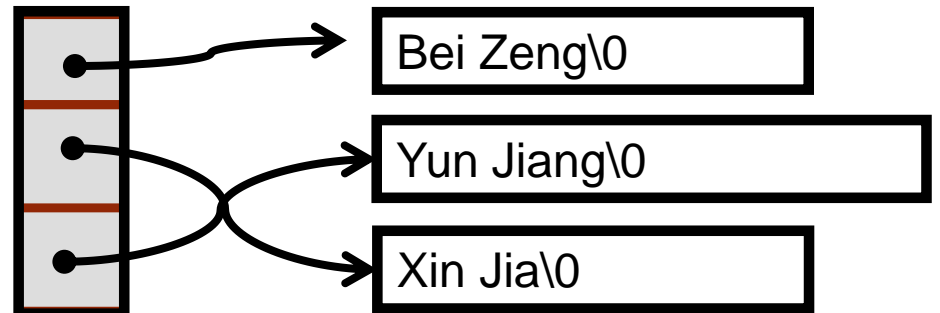
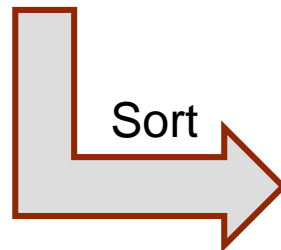
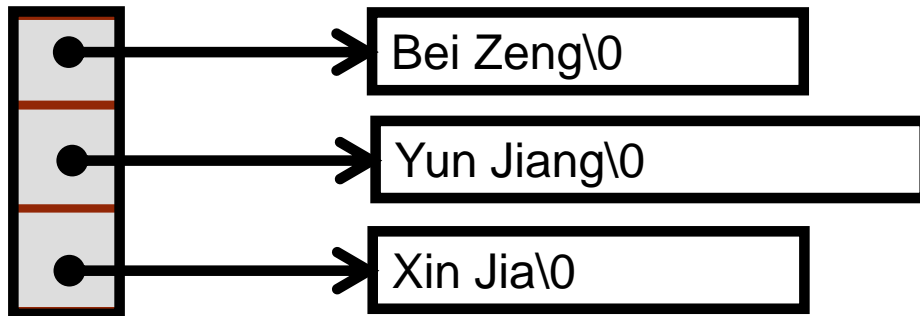


5.6 Pointer arrays; pointers to pointers

- **Pointers are variables**
 - can be stored in arrays
- **Example: student name list: a 2 dimension array, which can be a pointer array;**

Pointer array

char name*





5.7 Multi-dimensional arrays



Array of pointers

- flexible



Multi-dimensional arrays

- Rectangular, therefore inflexible



5.9 Pointers vs. multi-dimensional arrays



Definition:

```
int a[10][20];
```

```
int *b[10];
```

The size of a is $10 * 20 = 200$

The size of b is flexible.

The following two expressions are both legal.

```
a[3][4];
```

```
b[3][4];
```



5.10 Command-line arguments

main function has two arguments

- **Argc:** argument count
- **Argv:** argument vector

Example

```
/* echo command-line arguments */  
  
main(int argc, char *argv[ ]) {  
    int i;  
    for (i = 1; i < argc; i ++ )  
        printf("%s%s", argv[i], (i < argc - 1) ? " ": "");  
    printf("\n");  
    return 0;  
}
```

See hands-on experiment 5.10 for more details.