Course organization

- Course introduction (Week 1)
  - Code editor: Emacs
- Part I: Introduction to C programming language (Week 1 - 12)
  - Chapter 1: Overall Introduction (Week 1-4)
    - C
    - Unix/Linux
  - Chapter 2: Types, operators and expressions (Week 4)
  - Chapter 3: Control flow (Week 5, 6)
  - Chapter 4: Functions and program structure (Week 6- 7)
  - **Chapter 5: Pointers and arrays (Week 8-9)**
  - Chapter 6: Structures (Week 10)
  - Chapter 7: Input and Output (Week 11)
- Part II: Skills others than programming languages (Week 12- 14)
  - Debugging tools（Week 12-13）
  - Keeping projects documented and manageable （Week 14）
  - Source code managing （Week 14）
- Part III: Reports from the battle field (student forum) (Week 15 – 16)

Chapter 5. Points and Arrays

Chaochun Wei

Shanghai Jiao Tong University

Spring 2014

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Contents

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

5.1 Pointers and address

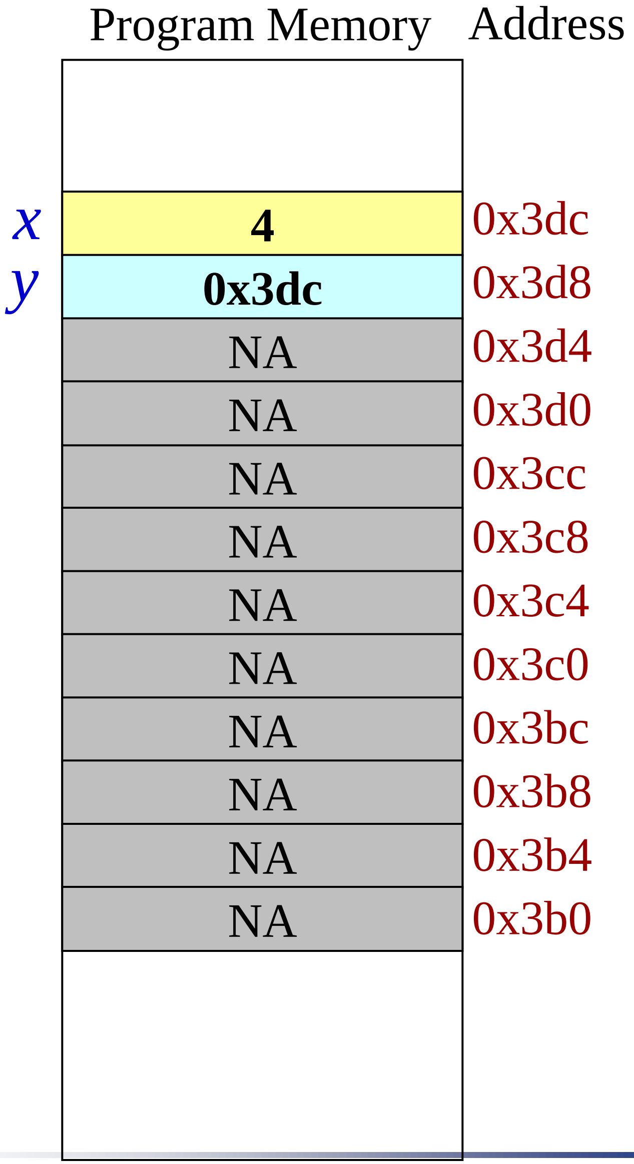
- For any type T, you may form a pointer type to T.
  - Pointers may reference a function or an object.
  - The value of a pointer is the address of the corresponding object or function
  - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators: *** dereferences a pointer, & creates a pointer** (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;
 ...
```



	Program Memory	Address
x	**4**	0x3dc
y	**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

- 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!!!

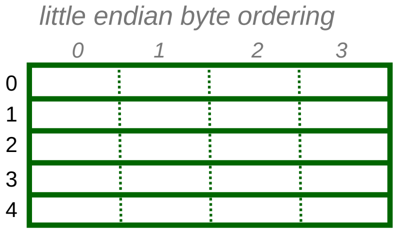
More details see hands-on experiments 5.2

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};
...
```

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

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;
```

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

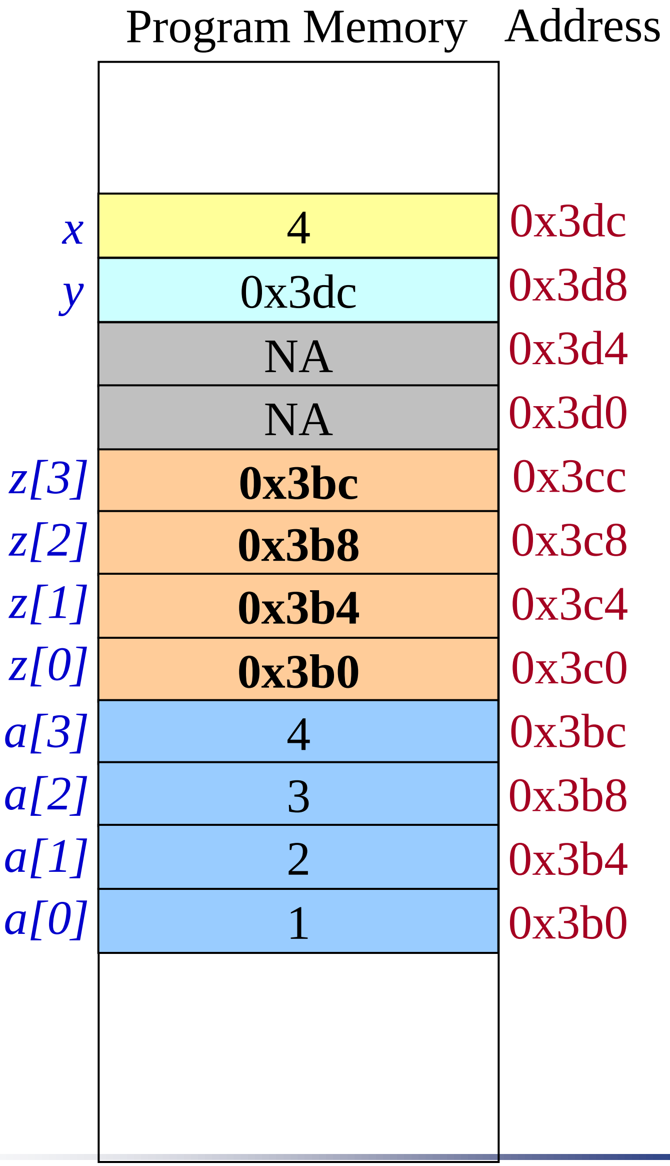
5.3 Arrays and pointers (Continued II)

```
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
x	4	0x3dc
y	0x3dc	0x3d8
	NA	0x3d4
	NA	0x3d0
z[3]	**0x3bc**	0x3cc
z[2]	**0x3b8**	0x3c8
z[1]	**0x3b4**	0x3c4
z[0]	**0x3b0**	0x3c0
a[3]	4	0x3bc
a[2]	3	0x3b8
a[1]	2	0x3b4
a[0]	1	0x3b0

5.3 Arrays and pointers (Continued III)

```
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:
  z[0] = a;
  z[1] = a + 1;
  z[2] = a + 2;
  z[3] = a + 3;
Step 3: No change in z’s values
  z[0] = (int *)((char *)a);
  z[1] = (int *)((char *)a
                 + sizeof(int));
  z[2] = (int *)((char *)a
                 + 2 * sizeof(int));
  z[3] = (int *)((char *)a
                 + 3 * sizeof(int));
```



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

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 experiments for more details

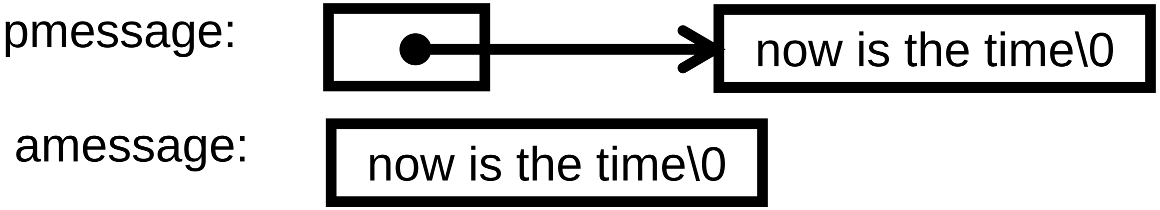
5.5 Character pointers and functions

- 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:

now is the time\0

amessage:

now is the time\0

5.5 Character pointers and functions

- Assignment: is not a string copy operation

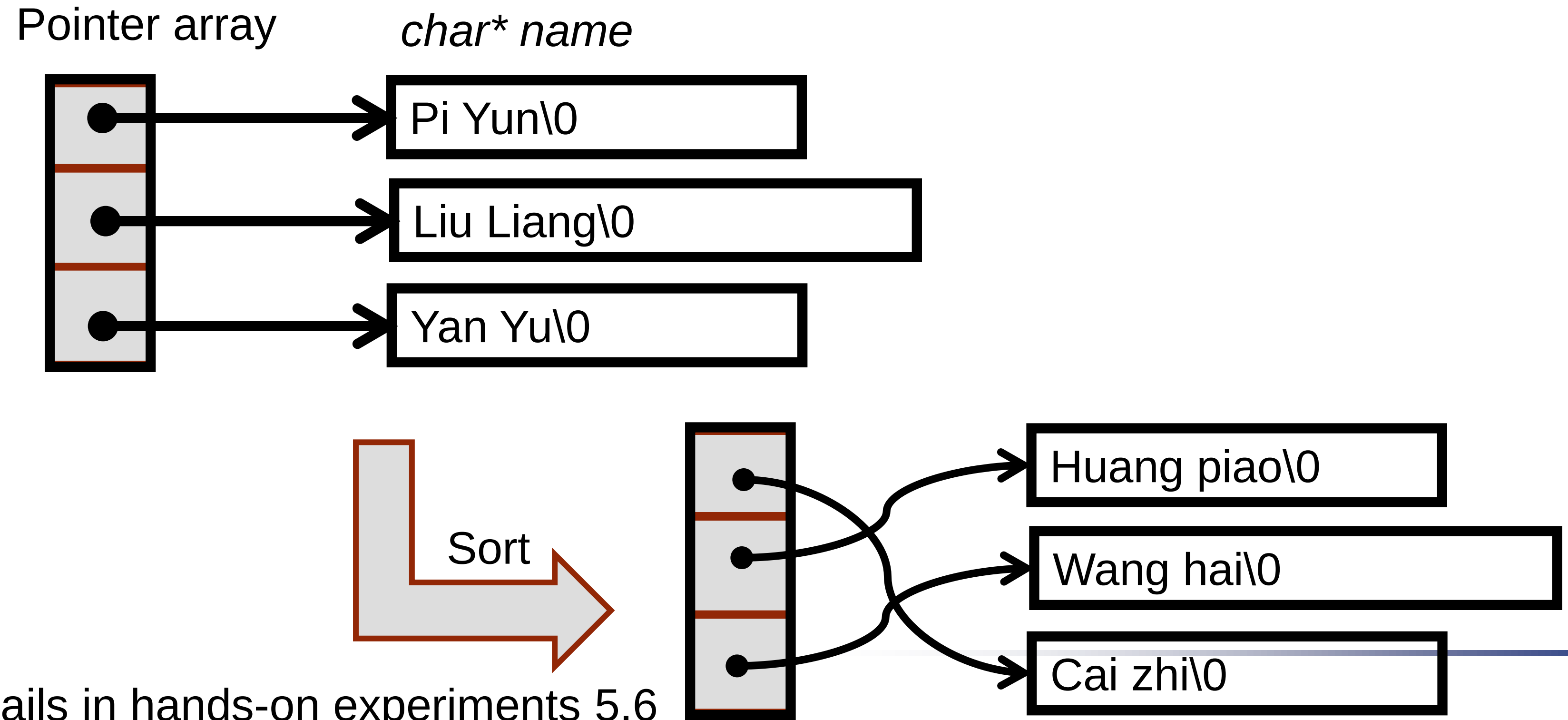
```
char *s = “this is a string”, *t;
t = s ;  /* this is not a string copy */
           /* this copies to t the address that s points to */
```

- To copy a string, we need a loop

```
/* strcpy: copy t to s */
void strcpy(char *s, char *t) {
      while (( *s++ = *t++) != ‘\0’) ;
}
```

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;



More details in hands-on experiments 5.6

5.7 Multi-dimensional arrays

- Array of pointers
  - flexible
- Multi-dimensional arrays
  - Rectanglar, therefore inflexible

5.9 Pointers vs. multi-dimensional arrays

- Definition:

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

The following two expression are both legal.

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

The size of a is 10*20 = 200

The size of b is flexible.

5.10 Command-line arguments

- main function has two arguments
  - Argc: argument count
  - Argv: argument vector
- Example

```
/* echo comman-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;
}
```