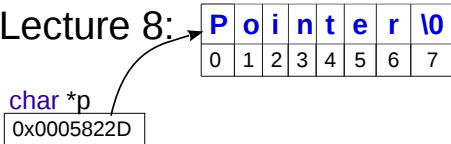


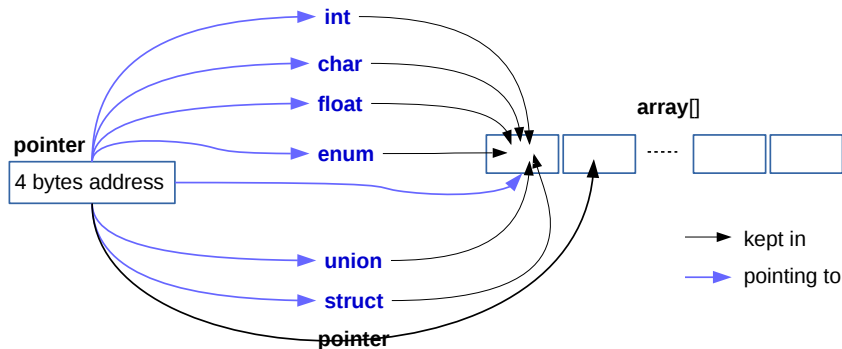
# C Programming

Lecture 8:



Lecturer: *Dr. Wan-Lei Zhao*  
*Autumn Semester 2022*

- 1 Pointer to Primitive Type Variables
- 2 Pointer to Array
- 3 Pointer to struct Variables
- 4 Dynamic Memory Allocation
- 5 List Structure



- Pointer essentially is the address of a variable
- Any types of variable has an address
- Array has address too
- It is allowed to have pointer array (array of addresses)

`dataType` \*`pointVariableName`

- Pointer is a variable too
- A variable keeps address of other variable(s)
- “\*” followed by variable name of the pointer

```
1 int main()  
2 {  
3     int *pt; //pointer points to an integer variable  
4 }
```

# Pointer initialization

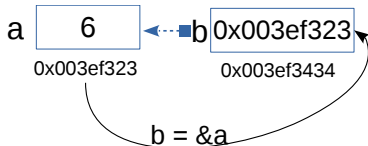
- Pointer is a variable too
- A variable keeps address of other variable(s)
- “\*” followed by variable name of the pointer

```
1 #include <string.h>
2 #include <stdio.h>
3 int main()
4 {
5     short int *pt=NULL; //points to an integer variable
6     float a = 3.1;
7     float *fpt = &a;
8     printf("Size_of_pt: %d\n", sizeof(pt));
9     printf("Size_of_fpt: %d\n", sizeof(fpt));
10    printf("Size_of_short_int: %d\n", sizeof(short int));
11 }
```

- “&” is an operator (something new!)
- “&a” extracts the address of variable **a**
- Address of variable **a** (4 bytes number) is then assigned to “fpt”

# Pointer in its nature

```
1 #include <string.h>
2 #include <stdio.h>
3 int main()
4 {
5     int a = 6;
6     int *b = &a;
7     ....
```



- “&a” extracts the address of variable **a**
- Address of variable **a** (4 bytes number) is then assigned to “fpt”

# Visit variable by its pointer (1)

```
1 #include <string.h>
2 #include <stdio.h>
3 int main()
4 {
5     short int a = 4;
6     short int *pa = &a;
7     float b = 3.1;
8     float *pb = &b;
9     printf("a = %d\n", a);
10    printf("b = %f\n", b);
11    printf("*pa = %d\n", *pa);
12    printf("*pb = %f\n", *pb);
13    printf("pa = %ld\n", pa);
14    printf("pb = %ld\n", pb);
15    return 0;
16 }
```

[Output:]

```
1 ??
2 ??
3 ??
4 ??
5 ??
6 ??
```

- “\*pa” takes the value from the address kept by pa

## Visit variable by its pointer (2)

```
1 #include <string.h>
2 #include <stdio.h>
3 int main()
4 {
5     short int a = 4;
6     short int *pa = &a;
7     float b = 3.1;
8     float *pb = &b;
9     printf("a=%d\n", a);
10    printf("b=%f\n", b);
11    printf("*pa=%d\n", *pa);
12    printf("*pb=%f\n", *pb);
13    printf("pa=%ld\n", pa);
14    printf("pb=%ld\n", pb);
15    return 0;
16 }
```

[Output:]

```
1 4
2 3.1
3 4
4 3.1
5 0439082323
6 0439082336
```

- “\*pa” takes the value from the address kept by pa



## Revisit: swap values of $a$ and $b$ (1)

```
1 #include <stdio.h>
2 void swap(int a, int b)
3 {
4     int tmp = a;
5     a = b;
6     b = tmp;
7     return ;
8 }
9 int main()
10 {
11     int a = 3;
12     int b = 5;
13     printf("a=%d, b=%d\n", a, b);
14     swap(a, b);
15     printf("a=%d, b=%d\n", a, b);
16     return 0;
17 }
```

```
1 #include <stdio.h>
2 int a, b;
3 void swap()
4 {
5     int tmp = a;
6     a = b;
7     b = tmp;
8     return ;
9 }
10 int main()
11 {
12     a = 3;
13     b = 5;
14     printf("a=%d, b=%d\n", a, b);
15     swap(a, b);
16     printf("a=%d, b=%d\n", a, b);
17     return 0;
18 }
```

## Revisit: swap values of $a$ and $b$ (2)

```
1 #include <stdio.h>
2 void swap(int a, int b)
3 {
4     int tmp = a;
5     a = b;
6     b = tmp;
7     return ;
8 }
9 int main()
10 {
11     int a = 3;
12     int b = 5;
13     printf("a=%d, b=%d\n", a, b);
14     swap(a, b);
15     printf("a=%d, b=%d\n", a, b);
16     return 0;
17 }
```

```
1 #include <stdio.h>
2 void swap(int *a, int *b)
3 {
4     int tmp = *a;
5     *a = *b;
6     *b = tmp;
7     return ;
8 }
9 int main()
10 {
11     int a = 3;
12     int b = 5;
13     printf("a=%d, b=%d\n", a, b);
14     swap(&a, &b);
15     printf("a=%d, b=%d\n", a, b);
16     return 0;
17 }
```

## Revisit: swap values of $a$ and $b$ (3)

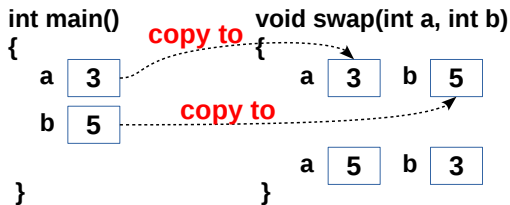


Figure: What happens for `swap(int a, int b)`.

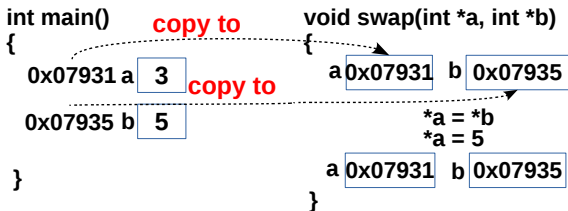


Figure: What happens for `swap(int *a, int *b)`.

## Revisit: swap values of *a* and *b* (4)

```
1 #include <stdio.h>
2 void swap(int *a, int *b)
3 {
4     int tmp = *a;
5     *a = *b;
6     *b = tmp;
7     return ;
8 }
9 int main()
10 {
11     int a = 3;
12     int b = 5;
13     printf(" a=%d, b=%d\n", a, b);
14     swap(&a, &b);
15     printf(" a=%d, b=%d\n", a, b);
16     return 0;
17 }
```

```
1 #include <stdio.h>
2 void swap(adr a, adr b)
3 {
4     int tmp = *a;
5     *a = *b;
6     *b = tmp;
7     return ;
8 }
9 int main()
10 {
11     int a = 3;
12     int b = 5;
13     printf(" a=%d, b=%d\n", a, b);
14     swap(&a, &b);
15     printf(" a=%d, b=%d\n", a, b);
16     return 0;
17 }
```

- Given **adr** is an address type

# Summary over Pointer to Variables (1)

- Pointer is a variable or constant
- It keeps the address of a variable
- One is allowed to do operation on a variable by its address

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

```
1 void incr(int *a)
2 {
3     *a = *a + 1;
4 }
5 int main()
6 {
7     int a = 4, *b = &a;
8     printf("%d\n", *b);
9     incr(&a);
10    printf("%d\n", a);
11    printf("%d\n", *b);
12    return 0;
13 }
```

## Summary over Pointer to Variables (2)

- Pointer is a variable or constant
- It keeps the address of a variable
- One is allowed to do operation on a variable by its address

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 3, *p;
5     int b = 1;
6     *p = a;
7     p = b;
8     p = &c;
9 }
```

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 3, *p;
5     int b = 1;
6     float c = 2.2;
7     p = &a;
8     printf("%d", *p);
9     *p = b;
10    printf("%d", *p);
11    printf("%d", a);
12 }
```

## Summary over Pointer to Variables (3)

```
1 void incr(int *a)
2 {
3     *a = *a + 1;
4 }
5 int main()
6 {
7     int a = 4, *b = &a;
8     printf("%d\n", *b);
9     incr(&a);
10    printf("%d\n", a);
11    printf("%d\n", *b);
12    return 0;
13 }
```

```
1 void incr(int *a)
2 {
3     a = a + 4;
4 }
5 int main()
6 {
7     int a = 4, *b = &a;
8     printf("%d\n", *b);
9     incr(&a);
10    printf("%d\n", a);
11    printf("%d\n", *b);
12    return 0;
13 }
```

- 'incr(int\* a)' on the right, increases the address number of **a**
- It points to **another memory cell**
- **a** inside 'incr(int \*a)' is a local variable
- It has no effect on input variable

# Explained

```
1 void incr(addr a)
2 {
3     *a = *a + 1;
4 }
5 int main()
6 {
7     int a = 4, *b = &a;
8     printf("%d\n", *b);
9     incr(&a);
10    printf("%d\n", a);
11    printf("%d\n", *b);
12    return 0;
13 }
```

```
1 void incr(addr a)
2 {
3     a = a + 4;
4 }
5 int main()
6 {
7     int a = 4, *b = &a;
8     printf("%d\n", *b);
9     incr(&a);
10    printf("%d\n", a);
11    printf("%d\n", *b);
12    return 0;
13 }
```

- Given `addr` is an address type
- Keep the principle that parameter “**transfer by value**” in C
- `a` inside ‘`incr(addr a)`’ is a local variable
- It has no effect on input variable



# A Revisit about “scanf(·,·)”

```
1 int main()
2 {
3     int a = 0;
4     printf("Input value for a: ");
5     scanf("%d", &a); //<— pay attention to here
6     return 0;
7 }
```

- Now we should be clear why we put “&” before **a**
- By this way, we tell **scanf(·)** to put the user input value to which memory address
- Is it possible if we do something as following?

```
1 int main()
2 {
3     int a = 0;
4     printf("Input value for a: ");
5     scanf("%d", a); //<—ask yourself whether this is valid??
6     return 0;
7 }
```

- 1 Pointer to Primitive Type Variables
- 2 Pointer to Array
- 3 Pointer to struct Variables
- 4 Dynamic Memory Allocation
- 5 List Structure

# An Overview: Pointer to Array (1)

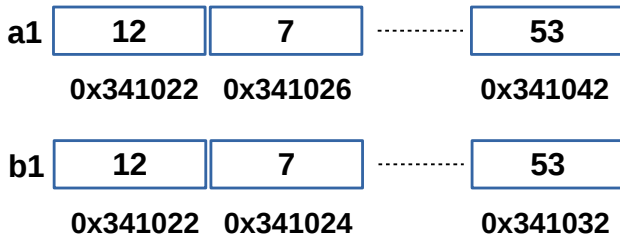


Figure: Two typical arrays of `int` type.

- Array is a continuous memory block
- It has a starting address
- It has a length
- It has a name

## An Overview: Pointer to Array (2)

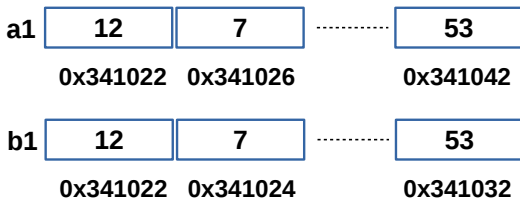


Figure: Two typical arrays of `int` type

- Unlike primitive type variable
- The name of an array is also the starting address of an array

```
1 int main()  
2 {  
3     int a[5]={4, 5, 7, 11, 13, 17};  
4     int *p = a;  
5     p = &a[0];  
6     return 0;  
7 }
```

## Definition and initialization (1)

```
int *p;  
int a1[10];  
p = a1;  
p = &a1[0];  
p = &a1;
```

- Definition of array pointer is the same as variable pointer
- Above two ways are valid
- '**p**' keeps the address of starting address of **a1**
- Now think about what "**p = p+2**" means here??

## Definition and initialization (2)

```
1 #include <stdio.h>
2 int main()
3 {
4     int a1[4] = {31, 1, 11, 4};
5     int i = 0, *p = a1;
6     for(i=0; i<4; i++, p++)
7     {
8         printf("%d_", *p);
9     }
10    return 0;
11 }
```

- '**p**' visits element in array a1 one by one
- '**\*p**' takes the value according to the address in '**p**'

## Definition and initialization (3)

```
1 #include <stdio.h>
2 int main()
3 {
4     int a1[4]={31, 1, 11, 4};
5     int i = 0, *p = a1;
6     for(i=0;i<4; i++,p++)
7     {
8         printf("%d_", *p);
9     }
10    return 0;
11 }
```

```
1 #include <stdio.h>
2 int main()
3 {
4     int a1[4]={31, 1, 11, 4};
5     int i = 0, *p = a1;
6     for(i = 0; i < 4; i++)
7     {
8         printf("%d_", a1[i]);
9     }
10    return 0;
11 }
```

- '**p**' visits element in array a1 one by one
- '**\*p**' takes the value according to the address in '**p**'

## Definition and initialization (3)

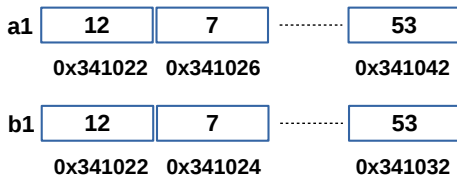


Figure: Two typical arrays of `int` type

```
1 int main()  
2 {  
3     int a1[6] = {12, 7, 7, 11, 13, 53};  
4     short b1[6] = {12, 7, 7, 11, 13, 53};  
5     int *pa = &a1;  
6     short *pb = &b1;  
7     return 0;  
8 }
```

- Like pointer to variable, different types of array need different types of



# Operations on Pointer of Array (1)

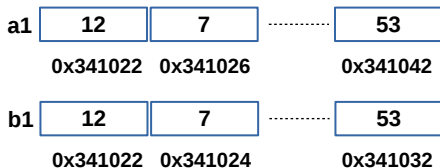


Figure: Two typical arrays of `int` type

```
1 int main()  
2 {  
3     int a1[6]={12, 7, 17, 11, 13, 53};  
4     short b1[6]={12, 7, 17, 11, 13, 53};  
5     int *pa = &a1;  
6     short *pb = &b1;  
7     pa++; pb++;  
8     printf("%d\n", *pa);  
9     printf("%d\n", *pb);  
10    return 0;  
11 }
```

[Output]

?

?

## Operations on Pointer of Array (2)

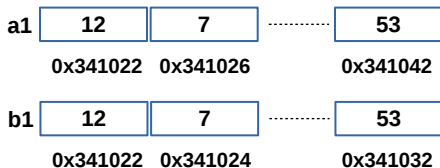


Figure: Two typical arrays of `int` type

```
1 int main()  
2 {  
3     int a1[5] = {12, 7, 17, 11, 13, 53};  
4     short b1[5] = {12, 7, 17, 11, 13, 53};  
5     int *pa = &a1;  
6     short *pb = &b1;  
7     pa++; pb++;  
8     printf("%d\n", *pa);  
9     printf("%d\n", *pb);  
10    return 0;  
11 }
```

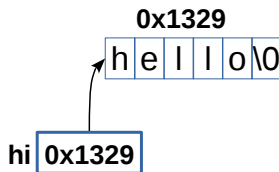
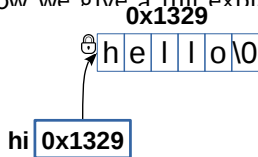
[Output]

7

7

# Pointer to String

- Think about following example
- We saw it many times
- Now we give a full explanation over it



```
1 #include <stdio.h>
2 int main()
3 {
4     char *hi = "hello";
5     hi[1] = 'a'; //<—
6     illegal
7     printf("%s\n", hi);
8     return 0;
9 }
```

```
1 #include <stdio.h>
2 int main()
3 {
4     char hi[] = "hello";
5     hi[1] = 'a'; //<—legal
6     printf("%s\n", hi);
7     return 0;
8 }
```

# Array of chars, String and Pointer of String

- Since pointer points to the first address of an array
- “str1” is defined as **constant** array of chars, and pointed by pointer str
- Definitions about “str2” and “str3” are **equivalent**
- Definition about “str4” is **different** from above three

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char *str1 = "hello"; //<—str1[0] = 'a' will be illegal
6     char str2[10] = "hello";
7     char str3[10] = {'h', 'e', 'l', 'l', 'o', '\0'};
8     char str4[10] = {'h', 'e', 'l', 'l', 'o'}; //<—it is
        different
9     printf("%s\n", str1);
10    printf("%s\n", str2);
11    printf("%s\n", str3);
12    printf("%s\n", str4);
13    return 0;
14 }
```

# Example of Pointer to Array (1)

- Given **str1**="abserds" and **str2**="xxxxx"
- You are required to copy the contents of one string to another

## Example of Pointer to Array (2)

- Given **str1**="abserds" and **str2**="xxxxx"
- You are required to copy the contents of one string to another
  - 1 Define pointers (p1 and p2) for **str1** and **str2**
  - 2 Pointing to the start of each
  - 3 Assign value of p1 to p2
  - 4 Repeat **Step 3** until the end of **str1**
  - 5 Assign '\0' to the end of **str2**

## Example of Pointer to Array (3)

```
1 #include <stdio.h>
2 int main()
3 {
4     char *str1="hello_world!";
5     char str2[16];
6     char *p1 = str1;
7     char *p2 = str2;
8     while(p1 != '\0')
9     {
10         *p2 = *p1;
11         p1++; p2++;
12     }
13     printf("%s\n", str1);
14     printf("%s\n", str2);
15     return 0;
16 }
```

- There is a **bug**, please tell me:)

## Example of Pointer to Array (4)

```
1 #include <stdio.h>
2 int main()
3 {
4     char *str1="hello_world!";
5     char str2[16];
6     char *p1 = str1;
7     char *p2 = str2;
8     while(*p1 != '\0')
9     {
10         *p2 = *p1;
11         p1++; p2++;
12     }
13     *p2='\0'; //<—indicate the end of the string
14     printf("%s\n", str1);
15     printf("%s\n", str2);
16     return 0;
17 }
```

- Be careful all the time



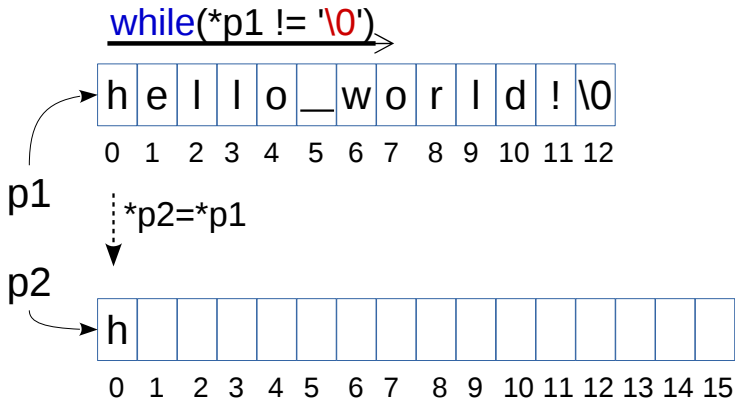
## Example of Pointer to Array (5)

```
1 #include <stdio.h>
2 void strCopy(char *p1, char *p2)
3 {
4     while(*p1 != '\0')
5     {
6         *p2 = *p1;
7         p1++; p2++;
8     }
9     *p2 = '\0';
10 }
11
12 int main()
13 {
14     char *str1 = "hello_world!";
15     char str2[16];
16     strCopy(str1, str2);
17     printf("%s\n", str1);
18     printf("%s\n", str2);
19     return 0;
20 }
```

## Example of Pointer to Array (6)

```
1 #include <stdio.h>
2 void strCopy(char *p1, char *p2)
3 {
4     while(*p1 != '\0')
5     {
6         *p2 = *p1;
7         p1++; p2++;
8     }
9     *p2 = '\0';
10 }
11
12 int main()
13 {
14     char *str1="hello_world!";
15     char str2[16];
16     strCopy(str1, str2);
17     printf("%s\n", str1);
18     printf("%s\n", str2);
19     return 0;
20 }
```

## Example of Pointer to Array (7)



- The while loop stop at '\0'
- '\0' will not be copied in the loop

## Popular functions for string operation (1)

1. **strlen**(str1); length of str1, '\0' is not counted
2. **strcpy**(str1, str2); copy str2 to str1
3. **strcmp**(str1, str2); compare two strings
4. **strcat**(str1, str2); concatenate two strings
5. **strncpy**(str1, str2, n); copy first n chars of str2 to str1

## Popular functions for string operation (2)

2. **strcpy**(str1, str2); copy str2 to str1
4. **strcat**(str1, str2); concatenate two strings

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char *str1="hello", *str2 = "world";
6     char hi[32];
7     strcpy(hi, str1);
8     strcat(hi, "_");
9     strcat(hi, str2);
10    printf("%s\n", hi);
11    return 0;
12 }
```

## Popular functions for string operation (3)

### 3. **strcmp**(str1, str2); compare two strings

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char *str1="hello", *str2 = "hi", *str3="hello";
6     if(strcmp(str1, str2) == -1)
7     {
8         printf("str1 < str2!\n");
9     } else if(strcmp(str1, str2) == 1) {
10         printf("str1 > str2!\n");
11     }
12     if(strcmp(str1, str3) == 0)
13     {
14         printf("They are equal!\n");
15     } else {
16         printf("They are unequal!\n");
17     }
18     return 0;
19 }
```

- 1 Pointer to Primitive Type Variables
- 2 Pointer to Array
- 3 Pointer to struct Variables**
- 4 Dynamic Memory Allocation
- 5 List Structure

# Pointer to struct Type Variable (1)

- The declaration of pointer to **struct** type is similar as pointer to primitive type and array

```
1 struct STD {  
2     char name[16];  
3     float gpa;  
4 };  
5 int main()  
6 {  
7     struct STD std1 = {"Peter", 3.8};  
8     struct STD *p = &std1;  
9     printf("Name: %s\n", (*p).name);  
10    printf("GPA: %f\n", (*p).gpa);  
11    return 0;  
12 }
```

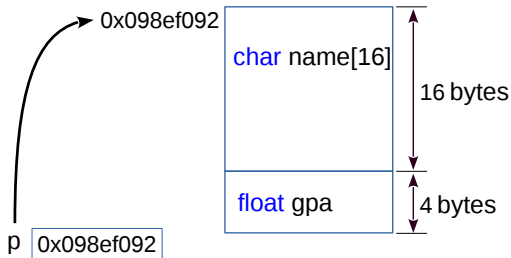


# Pointer to struct Type Variable (1)

- The declaration of pointer to `struct` type is similar as pointer to primitive type and array

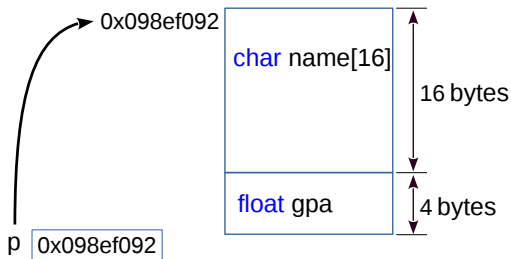
```
1 struct STD {  
2     char name[16];  
3     float gpa;  
4 };  
5 int main()  
6 {  
7     struct STD std1 = {"Peter", 3.8};  
8     struct STD *p = &std1;  
9     printf("Name: %s\n", (*p).name);  
10    printf("GPA: %f\n", (*p).gpa);  
11    return 0;  
12 }
```

## Pointer to struct Type Variable (2): explained



- Pointer keeps the starting address of the struct type variable
- `sizeof(p) = ?`

## Pointer to struct Type Variable (3): explained



- Pointer keeps the starting address of the struct type variable
- `sizeof(p) = ?`
- Notice that the address is only 4 bytes (32 bits system)

# Pointer to struct Type Variable

```
1 struct STD {
2     char name[16];
3     float gpa;
4 };
5 typedef struct STD STDT;
6 int main()
7 {
8     STDT std1 = {"Peter", 3.8};
9     struct STD *p = &std1;
10    printf("%s\n", (*p).name);
11    printf("%f\n", (*p).gpa);
12    return 0;
13 }
```

```
1 struct STD {
2     char name[16];
3     float gpa;
4 };
5 typedef struct STD STDT;
6 int main()
7 {
8     STDT std1 = {"Peter", 3.8};
9     struct STD *p = &std1;
10    printf("%s\n", p->name);
11    printf("%f\n", p->gpa);
12    return 0;
13 }
```

- `typedef` denotes “`struct STD`” as “`STDT`”
- “`p->`” is equivalent to “`(*p).`”

# Comparison Study over Pointers

```
1 #include <stdio.h>
2 struct STD {
3     char name[16];
4     float gpa;
5 };
6 int main()
7 {
8     struct STD std1 = {"Peter", 3.8};
9     struct STD *p = &std1;
10    int *q;
11    char *r;
12    printf("size_of_STD: %d\n", sizeof(struct STD));
13    printf("size_of_p: %d\n", sizeof(p));
14    printf("size_of_q: %d\n", sizeof(q));
15    printf("size_of_r: %d\n", sizeof(r));
16    return 0;
17 }
```

- The size is the same for different kinds of pointers
- Why??

- 1 Pointer to Primitive Type Variables
- 2 Pointer to Array
- 3 Pointer to struct Variables
- 4 Dynamic Memory Allocation**
- 5 List Structure

# Static and Dynamic Memory Allocation (1)

- Recall what the variables we learned so far

- 1 Primitive type variables
- 2 Primitive type arrays
- 3 Composite type variables
- 4 Composite type arrays

```
1 struct STD {  
2     char name[16];  
3     float gpa;  
4 };  
5 typedef STD STDT;  
6 int main()  
7 {  
8     int a, a1[10];  
9     STDT b, b1[10];  
10 }
```

- The memory cells for a, a1, b and b1 are allocated when your code is loaded into memory
- It is done **before the code is executed**

## Static and Dynamic Memory Allocation (2)

- In some cases, we are not sure how long is the array we need before run it
- We have two options for this case
  - ① Apply for a very long array, i.e., 65,536
  - ② Apply the memory cells in the **runtime**
- The second way is called dynamic memory allocation



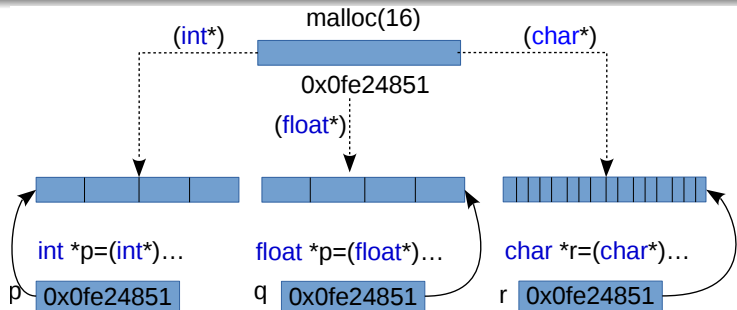
```
int *p = (int*)malloc(sizeof(int)*10);
```

- 1 Apply a block of memory sized of  $10 * \text{sizeof}(\text{int}) = ??$
- 2 Function “**malloc**(.)” returns the starting address of this memory
- 3 Convert this starting address to an `int` type pointer
- 4 Assign this starting address to **p**

```
int *p = (int*)malloc(sizeof(int)*10);
```

- 1 Function “**malloc**(.)” sends the applicaiton to OS
- 2 When the application is approved, a block of memory is returned
- 3 OS extracts memory from **Heap**
- 4 Once it is allocated, you can operate it as an array

# Dynamic Memory Allocation: explained



```
1 #include <stdlib.h>
2 int main()
3 {
4     void *x = malloc(16);
5     int *p = (int*)x;
6     float *q = (float*)x;
7     char *r = (char*)x;
8 }
```

- We just show it is possible
- It is **NOT** suggested in practice

# Dynamic Memory Allocation: example

```
1 #include <stdlib.h>
2 int main()
3 {
4     int i = 0, *a1 = (int*)malloc(5*sizeof(int));
5     for(i = 0; i < 5; i++)
6     {
7         a1[i] = i+1;
8     }
9     free(a1); //<—release the memory pointing by a1
10    return 0;
11 }
```

- 1 Function “**malloc**(.)” returns the starting address of this block of memory
- 2 Once it is allocated, you can operate it as an array
- 3 Always remember to **release** it by calling **free**(.)

# Dynamic Memory Allocation: memory leakage (1)

- Different from static memory allocation
- You are required to release the dynamically allocated memory on your own
- If you fail to do that, memory leakage occurs (90%) C bugs arise from this

```
1 #include <stdlib.h>
2 int main()
3 {
4     int i = 0, *a1 = (int*)malloc(5*sizeof(int));
5     for(i = 0; i < 5; i++)
6     {
7         a1[i] = i+1;
8     }
9     free(a1); //<— very important here
10    return 0;
11 }
```

## Dynamic Memory Allocation: memory leakage (2)

```
1 #include <stdlib.h>
2 int main()
3 {
4     int i = 0, *a1 = (int*) malloc(5* sizeof(int));
5     for(i = 0; i < 5; i++)
6     {
7         a1[i] = i+1;
8     }
9     free(a1);
10    a1[2] = 3; //<— illegal memory access
11    return 0;
12 }
```

- You are not allowed to use memory that has been released
- Above code (line 10) causes **illegal memory access exception**

## Dynamic Memory Allocation: memory leakage (3)

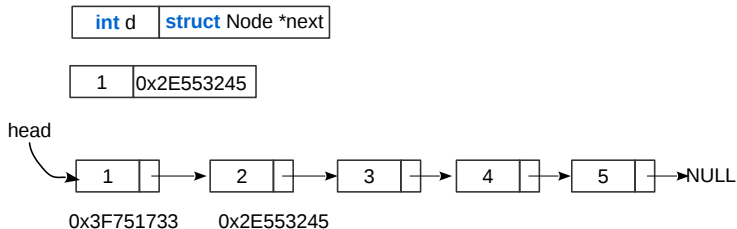
```
1 #include <stdlib.h>
2 int main()
3 {
4     int i = 0, *a1 = (int*) malloc(5* sizeof(int));
5     for(i = 0; i < 5; i++)
6     {
7         a1[i] = i+1;
8     }
9     a1 = (int*) malloc(15* sizeof(int)); //<-something wrong here
10    free(a1);
11    return 0;
12 }
```

- You are not allowed to use memory that has been released
- We lose the pointer to one block of memory (at line 9)
- Memory leaks (**ghost** memory cells)

- 1 Pointer to Primitive Type Variables
- 2 Pointer to Array
- 3 Pointer to struct Variables
- 4 Dynamic Memory Allocation
- 5 List Structure**

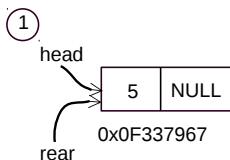


# Overview of List Structure



```
1 struct Node {  
2     int a;  
3     struct Node *next;  
4 };  
5 typedef Node TNode;
```

# Build List—Step 1



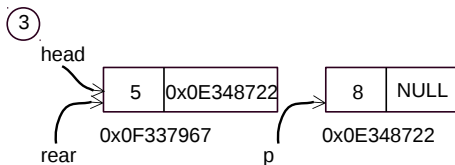
```
1 struct Node {  
2     int a;  
3     struct Node *next;  
4 };  
5 typedef Node TNode;  
6 int main()  
7 {  
8     TNode *head = NULL, *rear = NULL;  
9     TNode *p = (TNode*) malloc(sizeof(TNode));  
10    p->a = 5; p->next = NULL;  
11    head = p; rear = p;  
12 }
```

## Build List—Step 2



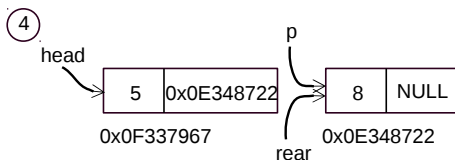
```
1 struct Node {  
2     int a;  
3     struct Node *next;  
4 };  
5 typedef Node TNode;  
6 TNode *buidList()  
7 {  
8     TNode *head = NULL, *rear = NULL;  
9     TNode *p = (TNode*) malloc(sizeof(TNode));  
10    p->a = 5; p->next = NULL;  
11    head = p; rear = p;  
12    p = (TNode*) malloc(sizeof(TNode));  
13    return head;  
14 }
```

## Build List—Step 3



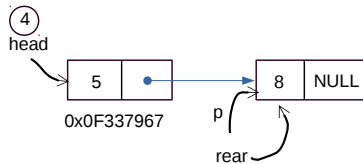
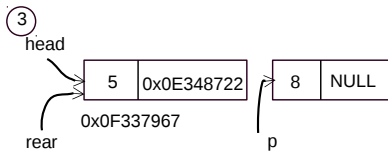
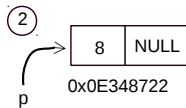
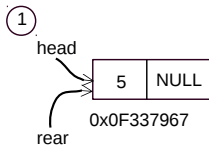
```
1 TNode *buidList()
2 {
3     TNode *head = NULL, *rear = NULL;
4     TNode *p = (TNode*) malloc(sizeof(TNode));
5     p->a = 5; p->next = NULL;
6     head = p; rear = p;
7     p = (TNode*) malloc(sizeof(TNode));
8     rear->next = p;
9     return head;
10 }
```

## Build List—Step 4

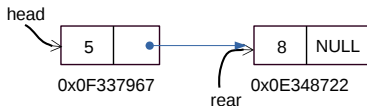


```
1 TNode *buidList()  
2 {  
3     TNode *head = NULL, *rear = NULL;  
4     TNode *p = (TNode*) malloc( sizeof(TNode) );  
5     p->a = 5; p->next = NULL;  
6     head = p; rear = p;  
7     p = (TNode*) malloc( sizeof(TNode) );  
8     rear->next = p;  
9     rear = p;  
10    return head;  
11 }
```

# Build List—Summary



# Print List



```
1 int printList(TNode *head)
2 {
3     TNode *p = head;
4     int i = 0;
5     while(p != NULL)
6     {
7         printf("%3d\n", p->a);
8         p = p->next;
9         i++;
10    }
11    return i;
12 }
```

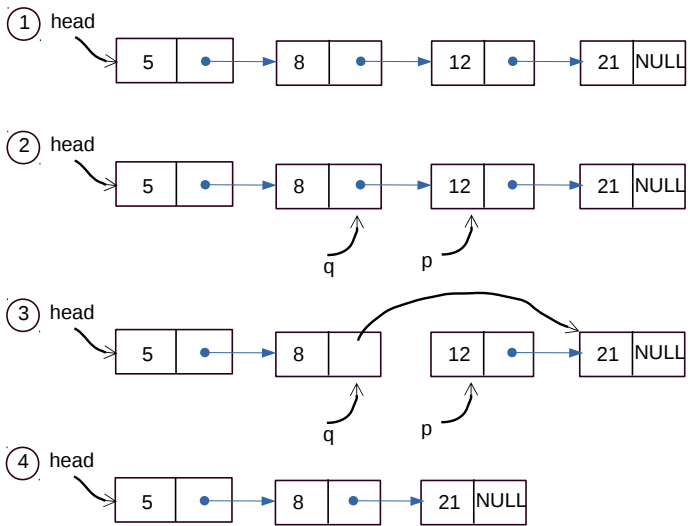
# Delete Node from List



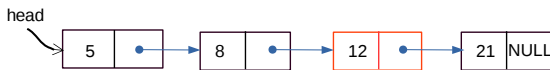
- We want to delete the node in which **a** equals to **12**



# Delete Node from List-Steps



# Delete Node from List—Procedure



- We want to delete the node in which **a** equals to 12
- 1 Find the node, whose **a** equals to 12
  - 2 Given it is **p**, the node before it is **q**
    - 1  $q \rightarrow \text{next} = p \rightarrow \text{next};$
    - 2  $p \rightarrow \text{next} = \text{NULL};$
    - 3  $\text{free}(p);$

# Delete Node from List—Codes



```
1 void deleteNode(int val , TNode *head)
2 {
3     TNode *p = head , *q = head;
4     //filling the codes here
5 }
```

# Delete Node from List—The answer



```
1 void deleteNode(int val , TNode *head)
2 {
3     TNode *p = head , *q = head;
4     while(p != NUL && p->a != val)
5     {
6         q = p;
7         p = p->next;
8     }
9     if(p != NULL && p->a == val)
10    {
11        q->next = p->next;
12        p->next = NULL;
13        free(p);
14    }
15 }
```

Why condition “p != NUL” first???

# What are the differences between Array and List

	Array	List
Structure	linear	linear
Memory	continous block	chain of blocks
Visit	subscript	linear scan
Insert/delete	element shifting	direct operation