C++ Programming

Ch. 4 Compound Types

Spring 2014

Myung-Il Roh

Department of Naval Architecture and Ocean Engineering
Seoul National University

Ch. 4 Compound Types

Contents

- ☑ Compound Types
- ☑ Arrays
- ☑ Strings
- ☑ Structures
- ☑ Pointers
- ☑ Dynamic Memory Allocation
- ☑ Practice

Compound Types

- Data form that can hold several values with one type
  : Array of char type variables
- Ex. Exam scores of students

- Data form that can hold several values of differing types
- Collection of variables under a single name with various types
- Ex. Student information: name, phone number, score, rank

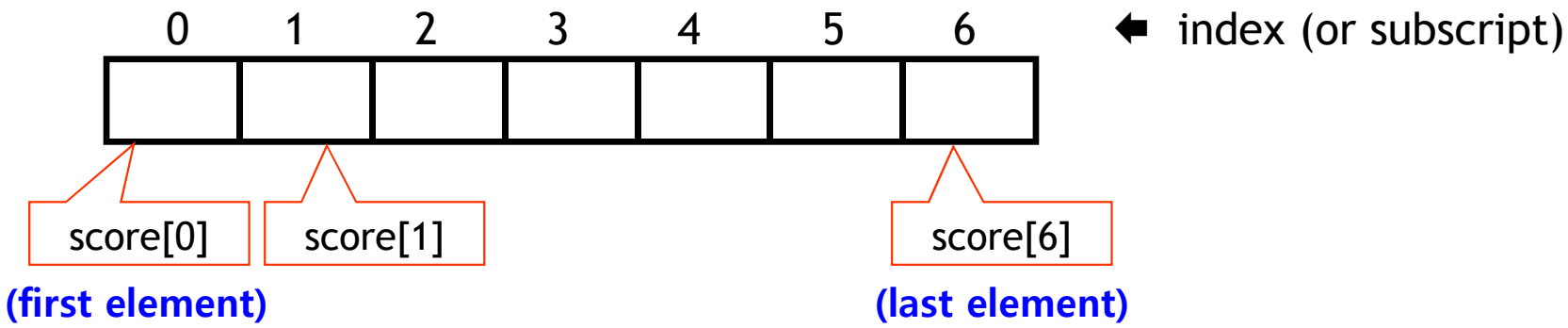
- Variable that can store a memory address of a value

Arrays (1/2)

☑ General form for declaring an array

- *type_name array_name[array_size];*
- Ex.

```
int score[7];        // create array of 7 integer values
                     // Each element has an int type value
```



Arrays (2/2)

☑ Initialization of Arrays

- Ex.

```
int cards[4] = {3, 6, 8, 10};          //
int cards[4];                          //
...
hand[4] = {5, 6, 7, 9};        //                . You can initialize array only
                                  when defining the array.
hand = cards;                  //                . You can use indices and
                                  assign values to the elements of an array
                                  individually.
```

- Ex.

```
float hotel[5] = {5.0, 2.5}; // What is the value of the other elements
                                ’hotel [2]’~’hotel[4]’?
                                                                    .
float hotel[500] = {5.0};    // Are all elements initialized to 5.0?
                                ‘hotel[0]’ is to be 5.0, ‘hotel[1]’~
                                ’hotel[499]’ are initialized to 0.0.
```

Strings

: A series of characters stored in consecutive bytes of memory

☑ Dealing with Strings

- Functions: getline(), get()
- Ex.

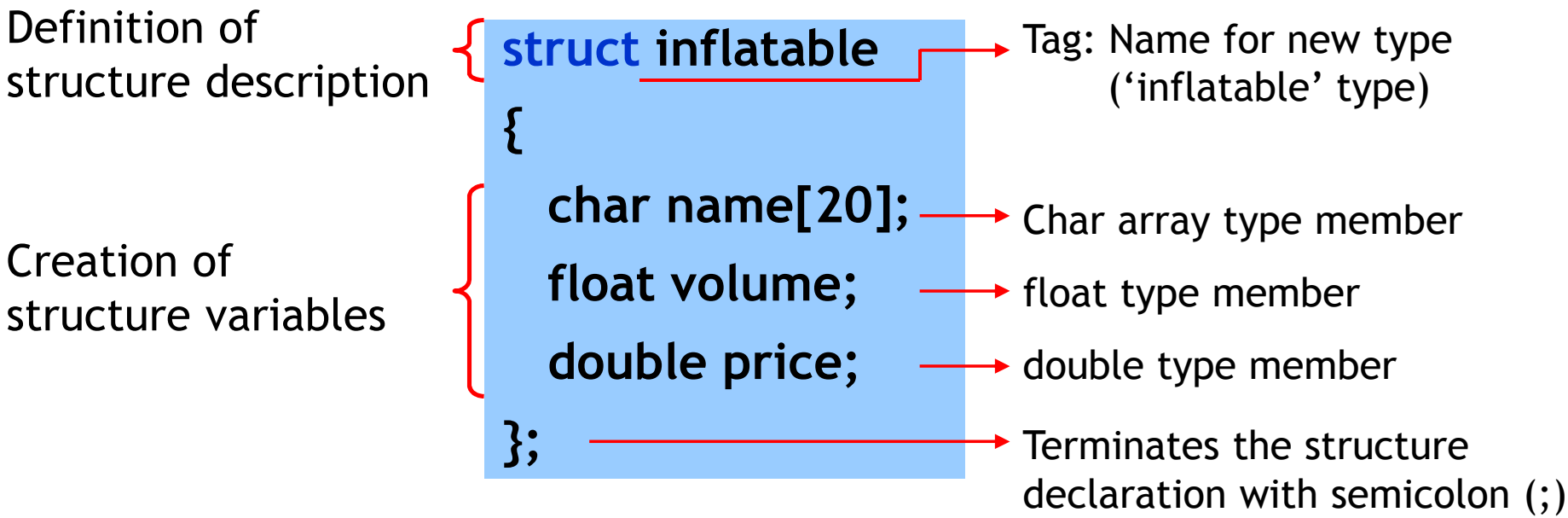
```
char name[30];
...
cin >> name;        // Read by a word. Ignore the initial white-space
                       character. Consider white-space character
                       as the end of the string.
cin.getline(name, 50);      // Read up to 50 characters or until
                               a newline character.

cin.get(name, 50);          // Read up to 50 characters or until
                               a newline character.

cin.get();          // Read one character regardless of its type.
```

Structures (1/6)

- ☑ More versatile data form than an array because a single structure can hold items of more than one data types
- ☑ Use of Structures
  - Step 1: Define a structure description. That is, describe and label the different types of data that can be stored in the structure.
  - Step 2: Create structure variables (data object)



Structures (2/6)

☑ Creation and Access of Structure Members

■ Create a new structure variable

- Ex.

```
struct inflatable goose;    // old C syntax
inflatable vincent;         // new C++ syntax. We can omit the 'struct'
                               keyword.
                            // create a structure variable of type 'inflatable'.
```

■ Access to the new structure variable by member access operator

- Ex.

```
vincent.price;    // refer to the 'price' member of the structure 'vincent'.
                  // 'vincent.price' is a double type variable.
```

Structures (3/6)

☑ Initialization of Structure Variables

- Method 1: .
  - Ex.

```
inflatable guest = {“Glorious Gloria”,      // name
                    1.88,                   // volume
                    29.99                   // price
                   };
```

- Method 2: .
  - Ex.

```
inflatable guest = {“Glorious Gloria”, 1.88, 29.99};
```

guest.name[0] is ‘G’

Structures (4/6)

of Structure Variables with the

- Ex.

```
struct perks {                  // template for the structure 'perks'
  int key_number;
  char car[12];
} mr_smith, ms_jones;           // create 2 structure variables of type 'perks'.
```

- Memberwise assignment
  - Ex.
    mr_smith.key_number = ms_jones.key_number;

- Assign a structure variable to the other structure variable
  - Ex.
    guest = vincent;

Structures (5/6)

of a Structure Variable with the

- Ex.

```
struct perks {                  // template for the structure 'perks'
  int key_number;
  char car[12];
} mr_glitz = {                  // create a structure variable of type 'perks'.
            7,                  // value for 'mr_glitz.key_number' member
            "parckard"          // value for 'mr_glitz.car' member
};
```

Structures (6/6)

☑ Arrays of Structures

- Ex.

```
inflatable gifts[100];          // create an array of 100 inflatable
                                   structures.
cin >> gifts[0].volume;         // use volume member of first structure.
cout << gifts[99].price;        // display price member of last structure.
```

☑ Initialization of a Structure Array

- Ex.

```
inflatable guests[2] = {        // initialize an array of structures.
  {"Bambi", 0.5, 21.99},        // first structure in array
  {"Godzilla", 200, 56.5}       // last structure in array
};
```

Pointers (1/3)

: represent

.

: represent .

☑ Declaration of Pointers

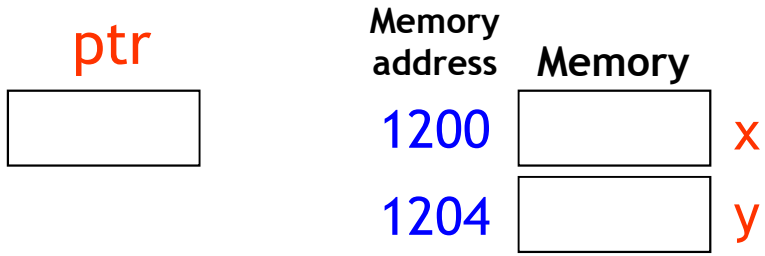
- Declare a pointer to a data type of an indicated variable.
- A pointer declaration must specify what type of data to which the pointer points.
- Ex.

```
int *ptr;          // declare a pointer 'ptr' to an int. old C style
int* ptr;          // same as above. new C++ style
                   // emphasize the idea that int* is a type,
                       'pointer-to-int'.
                   // spaces around the * operator are optional.
int* p1, p2;       //
```

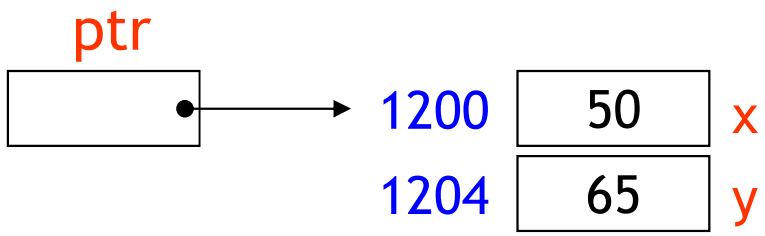
Pointers (2/3)

☑ Pointer and Numbers

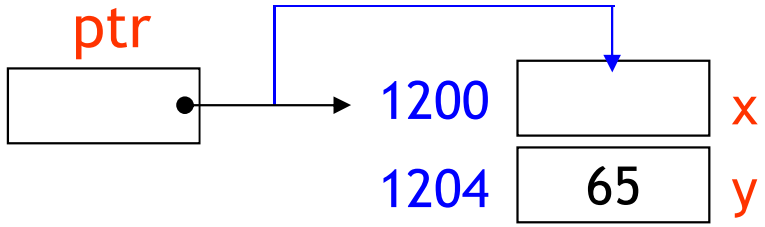
```
int x, y, *ptr;
```



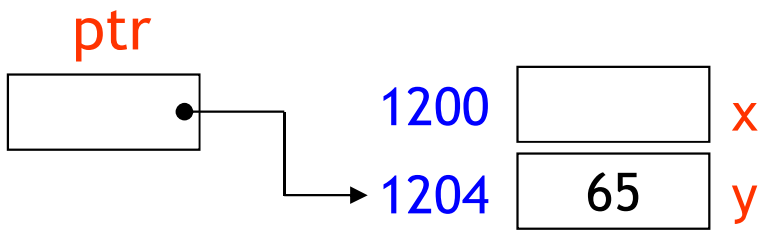

```
x = 50;
y = 65;
ptr = &x;
```




```
*ptr = 100;
```




```
ptr = &y;
```



Pointers (3/3)

☑ Declaration and Initialization of Pointers

- Ex.

```
int higgens = 5;          // declare a variable.
int* pi = &higgens;       // declare a pointer to an int and
                             assign the address of int to the pointer.
```

☑ A pointer represents a memory address (managed by the operating system) of a variable. Thus,

.

- Ex.

```
int* pi = 120;      //        .
```

.

Dynamic Memory Allocation (1/7)

:

.

:

while a program is running.

☑ Dynamic Memory Allocation with ‘ ’

- Allocate a dynamic memory when it is needed during runtime.
- Representation
  - *type_name *pointer_name = new type_name;*
- Ex.

```
int *pi = new int;
*pi = 1001;
cout << “int”;
cout << “ value = “ << *pi <<“ : address = “<< pi << “ \ n”;
```

Result:
int value = 1001 : address = 007D0CA0
(The address can be different.)

Dynamic Memory Allocation (2/7)

☑ Free Memory with ‘ ’

- Return memory to the memory pool when the memory that is allocated by ‘new’ is no longer needed.
- Static allocated memory by the declaration of variable is not freed by ‘new’.

.

- Representation
  - *delete pointer_name;*
- Ex.

```
int *ps = new int;          // allocate memory by new.
delete ps;                  // free memory by delete.
```

Dynamic Memory Allocation (3/7)

☑ Creation and Deletion of a Dynamic Array

: Create an array by declaring it. The memory is allocated when the program is compiled whether or not the program uses the array. ➡ .

: Create an array during runtime by using dynamic memory if you need it, and select an array size after the program is running.

☑ Representation

- Create
  - *type_name *pointer_name = new type_name[array_size];*
- Delete
  - *delete[] pointer_name;*
- Ex.

```
int *psome = new int[10];      // get a block of memory for 10 int type values.
                               // return the address of the first element of
                                  the block, and assign to psome.
delete[] psome;                // free the dynamic array.
```

Dynamic Memory Allocation (4/7)

☑ Pointers and Arrays

- Pointers and arrays are closely connected.
- If arr is an array name, then the expression arr[i] is interpreted as *(arr + i), with the array name interpreted as the address of the first element of the array.
- That is,
- Ex.

```
int arr[5] = {1, 2, 3, 4, 5};
int *parr = new int[5];
int abb[5] = {5, 4, 3, 2, 1};
arr[3] = 10;                    //
parr[3] = 5;                    //
arr = arr + 1;                  //
parr = parr + 1;                //
cout << *(arr + 3);             //
cout << *(parr + 2);            //
parr = abb;                     //
arr = abb;                      //
```

Dynamic Memory Allocation (5/7)

☑ Using 'new' to Create Dynamic Structures

- Representation
  - *structure_name *pointer_name = new structure_name;*
- Ex.

```
inflatable *ps = new inflatable;
```

☑ Method for Accessing Members

- Using the arrow membership operator (simply, member operator) " "
- It is used usually for the structure pointing pointer.
- Ex.

```
ps->price;
or
(*ps).price;
```

Dynamic Memory Allocation (6/7)

☑ Arrow Membership Operator for Structure and Structure Pointer

- **Ex.**

```
struct things
{
    int good;
    int bad;
};
```

Structure of type 'things'

```
things grubnose = {3, 453};
things *pt = &grubnose;
```

Pointer to type 'things'

```
grubnose.good = 5;   // access to the structure member with '.' operator.
pt->bad = 5;         // access to the structure member pointed by structure
                        pointer with '->' operator.
```

Dynamic Memory Allocation (7/7)

☑ Three Ways of Managing Memory in C++

- Ordinary variables defined inside a function or a block (section of code enclosed between braces) use automatic storage.

- Two ways to make a variable static
  - Define variables externally, outside a function (global variables).
  - Use the keyword ‘ ’ when declaring a variable.
    - » Ex. static double fee = 56.60;

- The method using ‘new’ and ‘delete’ ➡
- It is possible to allocate memory in one function and free it in another.
- The lifetime of the data is not tied arbitrarily to the life of the program or the life of a function.

Summary (1/2)

, , and are three C++ .

. By using an index, or subscript, the individual elements in an array can be accessed.

, and the membership operator (.) can be used to access individual members. The first step in using structures is to create a structure template that defines what members the structure holds. The name, or tag, for this template then becomes a new type identifier. Then structure variables of that type can be declared.

Summary (2/2)

. A pointer points to the address it holds. The pointer declaration always states to what type of object a pointer points. Applying the dereferencing operator (*) to a pointer yields the value at the location to which the pointer points.

. The operator returns the address of the memory it obtains, and that address to a pointer can be assigned.

.

☑ The new and delete operators let you explicitly control when data objects are allocated and when they are returned to the memory pool.

Practice 1 (1/2)

☑ **Make a program that reads exam scores of 10 students and store them at an array, and calculates the average score of the class.**

- **Use an array to storing the score of students.**

```
Preprocessor directives
int main(void)
{
    declare a float type array 'score' for storing scores of 10 students.
    declare other variable if it is needed.

    for (int i = 0; i < 10; i++) {
        input the score of the ith student and
        store it in the corresponding element of the array.
    }

    return 0;
}
```

Practice 1 (2/2)

☑ **An example of calculating the average score**

```
// score[i] represents the score of the ith student
sum = 0.0;      // initialize a variable for storing total score

for (i = 0; i < 10; i++) {
    sum = sum + the score of the ith student
}

calculate the average score.
```

Practice 2

☑ **Make a program that reads the information of 10 students and prints out it to the screen, and calculates the average weight of the class.**

- **The information of students is the name and weight, and it should be stored at a structure array.**

```
Preprocessor directives
int main(void)
{
    define a structure template 'student' for the student information.
    declare a structure array for storing the information of 10 students.
    declare other variable if it is needed.

    for (int i = 0; i < 10; i++) {
        input the information of the ith student.
    }
    calculate the average weight.
    return 0;
}
```

Practice 3

☑ **Make a program that reads the information of students and prints out it to the screen.**

- **Get the number of students when the program is running.**
- **The information of students should stored at a dynamic structure array.**

```
Preprocessor directives
int main(void)
{
    define a structure template 'student' for the student information.
    get the number of students 'n'.
    // allocate memory the size of student[n] when the program is running.
    student *p = new student[n];
    for (i = 0; i < n; i++) {
        input the information of the ith student.
        print out the student information stored at ptr[i];
    return 0;
}
```

Practice 4 (1/2)

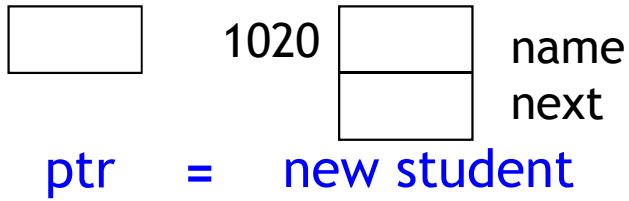
☑ **Make a program that reads the information of students and print out it to the screen.**

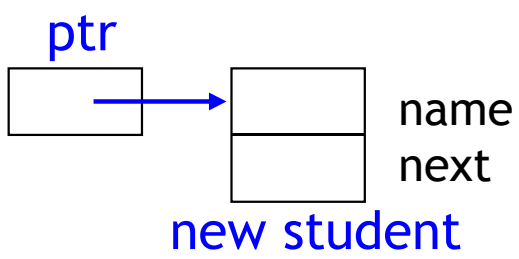
- **Get the number of students when the program is running.**
- **Represent the information students with a linked list (See Appendix).**

```
Preprocessor directives
int main(void)
{
    define a structure template 'student' for the student information.
    student *head, *ptr;
    get the number of students 'n'.
    ptr = new student;
    head = ptr;
    for(int i = 0; i < n; i++) {
        input the information of the ith student and point to the storage by ptr.
        ptr->next = new student;
        ptr = ptr->next;
    }
    ptr->next = NULL;
    ptr = head;
    for (i = 0; i < n; i++) {
        print out the student information stored at the storage pointed by ptr.
        ptr = ptr->next;
    }
    delete the allocated memory.
    return 0;
}
```

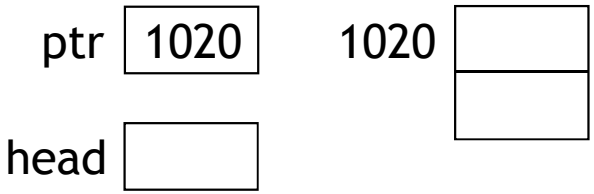
Practice 4 (2/2)

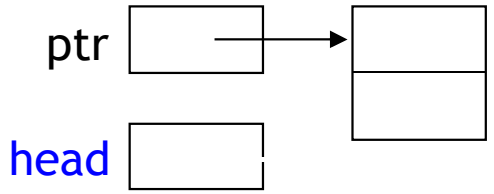
ptr = new student;



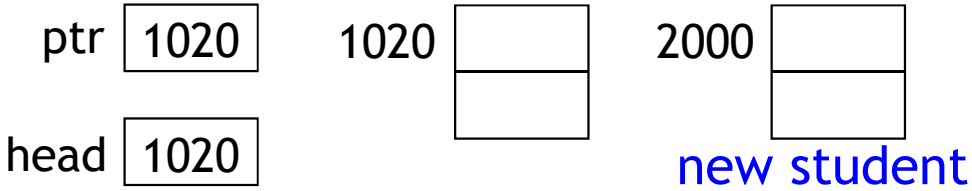


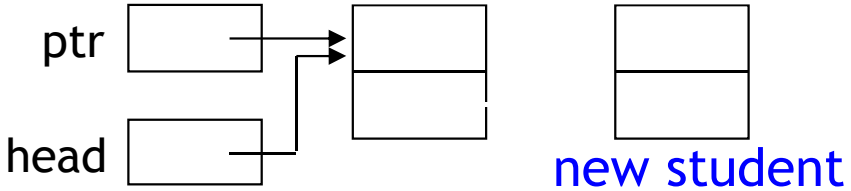
head = ptr;



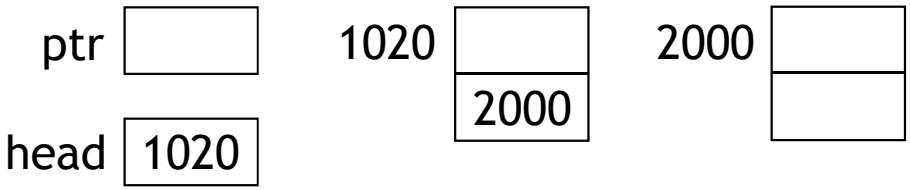


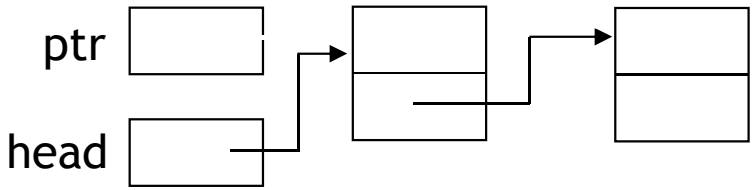
ptr->next = new student;





ptr = ptr->next;





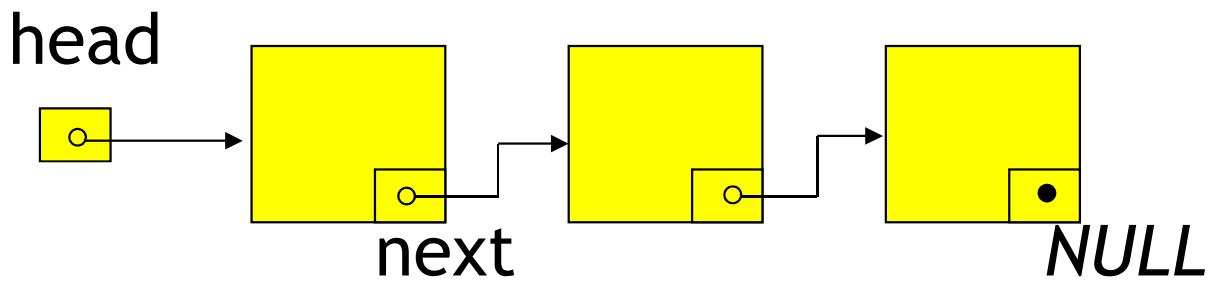
Reference Slides

Data Structures Using Structures & Pointers (1/4)

☑ **Linked List: LIFO(Last In First Out) Structure**

- **Ex.**

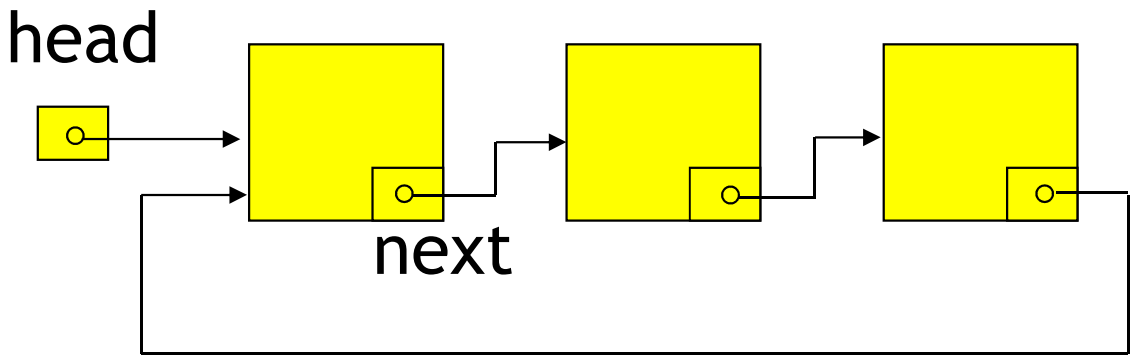
```
struct student {
    char name[20];
    student *next;          // pointer to the next element
} *pptr, *head;
```



Data Structures Using Structures & Pointers (2/4)

☑ **Ring Structure**

- **Same as the linked list**
- **The last element is connected to the first element.**

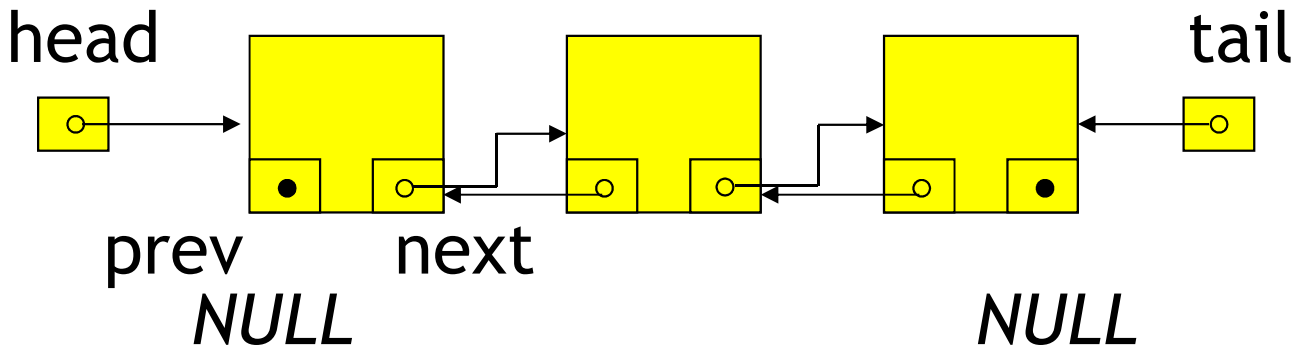


Data Structures Using Structures & Pointers (3/4)

☑ Double-Linked List

- **Ex.**

```
struct student {
    char name[20];
    student *prev;          // pointer to the previous element
    student *next;          // pointer to the next element
} *pptr, *head;
```



Data Structures Using Structures & Pointers (4/4)

☑ **B-Tree**

- **Ex.**

```
struct student {
    char name[20];
    student *left;
    // pointer to the left element
    student *right;
    // pointer to the right element
} *pptr, *head;
```