

CS2311 Computer Programming

LT9: Object Oriented Programming I

Computer Science, City University of Hong Kong

Semester A 2023-24

Exercise 1

```
char str[] = "Local";
```

```
char* p = str;
```

```
char** q = &p;
```

```
cout << p << endl;
```

```
cout << *p << endl;
```

```
cout << &p << endl;
```

```
cout << q << endl;
```

```
cout << *q << endl;
```

```
cout << &q << endl;
```

Exercise 2

```
void myFunc(char *myStr) {  
    int size = sizeof(myStr);  
    int len  = strlen(myStr);  
}
```

```
void main() {  
    char str[3];  
    strcpy(str, "hi");  
    int size = sizeof(str);  
    myFunc(str);  
}
```

Exercise 3

```
int main() {  
    char primaryStr[] = "abcdefg";  
    char secondaryStr[] = "ABC";  
    primaryStr[5] = '\\0';  
    strcpy(secondaryStr, primaryStr + 4);  
    strcat(primaryStr, secondaryStr);  
    cout << primaryStr << " " << strlen(primaryStr) << " " << sizeof(primaryStr);  
    return 0;  
}
```

Exercise 4

Write a function that flips the case of a letter

```
void flipCase( ) {  
    if ( )  
          
    else if ( )  
          
}  
  
int main(int argc, char *argv[]) {  
    char ch = 'g';  
    flipCase(  
    cout << ch;    // want this to print 'G'  
}
```

Exercise 5

Write a program that reverses a user entered string (assume max length is 1023)

```
int main() {  
    char str[1024];  
    cin >> str;  
  
    return 0;  
}
```

Today

- C-like struct

Struct: Definition

- A *composite data type* that groups a list of variables (possibly different types) under one name
- Variables are stored in a continuous memory areas
- Syntax and example:

```
struct typename {  
    type1  member_var1;  
    type2  member_var2;  
    ...  
};
```

```
struct StudentRecord {  
    char    name[51];  
    char    sid[9];  
    float   GPA;  
};
```

Initialization

- No memory is allocated when you *define* a struct
- When you declare a variable of a given struct type, enough memory is allocated for storing all struct members *contiguously*
- Example

```
StudentRecord danny = {"Danny", "50123456", 80};
```

Accessing Individual Members

- A member variable can be accessed with the use of the dot operator `.` :

```
danny.quiz += 10;
```

- Two structure types can have the same member name:

```
CS2363Student peter;
```

```
cin >> peter.quiz;
```

Accessing Individual Members (cont'd)

- A member variable can be accessed with the use of the dot operator “.”
 - danny.gpa += 10;
- Structure types can have the same member name without confliction

```
struct CS2311Student {  
    char    sid[9];  
    float   asg[3];  
    float   lab[10];  
    float   midterm;  
    float   final;  
};
```

```
struct CS6789Student {  
    char    sid[9];  
    float   asg[5];  
    float   final;  
};
```

```
CS2311Student peter;  
cin >> peter.final;  
CS6789Student danny;  
cin >> danny.final;
```

Example

```
struct CS2311Studnet {  
    int    sid;  
    float  quiz;  
    float  asg1;  
    float  asg2;  
};
```

```
int main() {  
    CS2311Studnet sr;  
    cout << "Enter your id, quiz, a1, and a2 marks\n";  
    cin >> sr.id;  
    cin >> sr.quiz;  
    cin >> sr.asg1;  
    cin >> sr.asg2;  
    float grade = (sr.quiz + sr.asg1 + sr.asg2)/3;  
    cout << sr.id << " grade:" << grade << endl;  
    return 0;  
}
```

Struct Assignment

- You can assign structure values to a structure variable:

```
CS2311Student danny;  
CS2311Student kitty;  
danny = kitty;
```

which is equivalent to:

```
danny.sid    = kitty.sid;  
danny.quiz   = kitty.quiz;  
danny.asg1   = kitty.asg1;  
danny.asg2   = kitty.asg2;
```

```
struct CS2311Student {  
    int    sid;  
    float  quiz;  
    float  asg1;  
    float  asg2;  
};
```

Hierarchical structures

- A member of a structure can be another structure:

```
struct Date {  
    int month, day, year;  
};  
  
struct PersonInfo {  
    double height, weight;  
    Date    birthday;  
};  
  
PersonInfo peter;  
peter.birthday.year = 2001;
```

Struct Pointer

- Struct pointer stores the memory address of the first byte of a struct variable

```
Date d;
```

```
d.year = 2022;
```

```
d.month = 11;
```

```
d.day = 7;
```

```
Date *dPtr = &d;
```

Address	Value	
0xa12	2022	d.year
0xa16	11	d.month
0xa1a	7	d.day
0x7c	0xa12	dPtr

Pointer and Arrow Syntax

- Pointers can point to a struct
- One way to do this would be to dereference and then use dot notation:

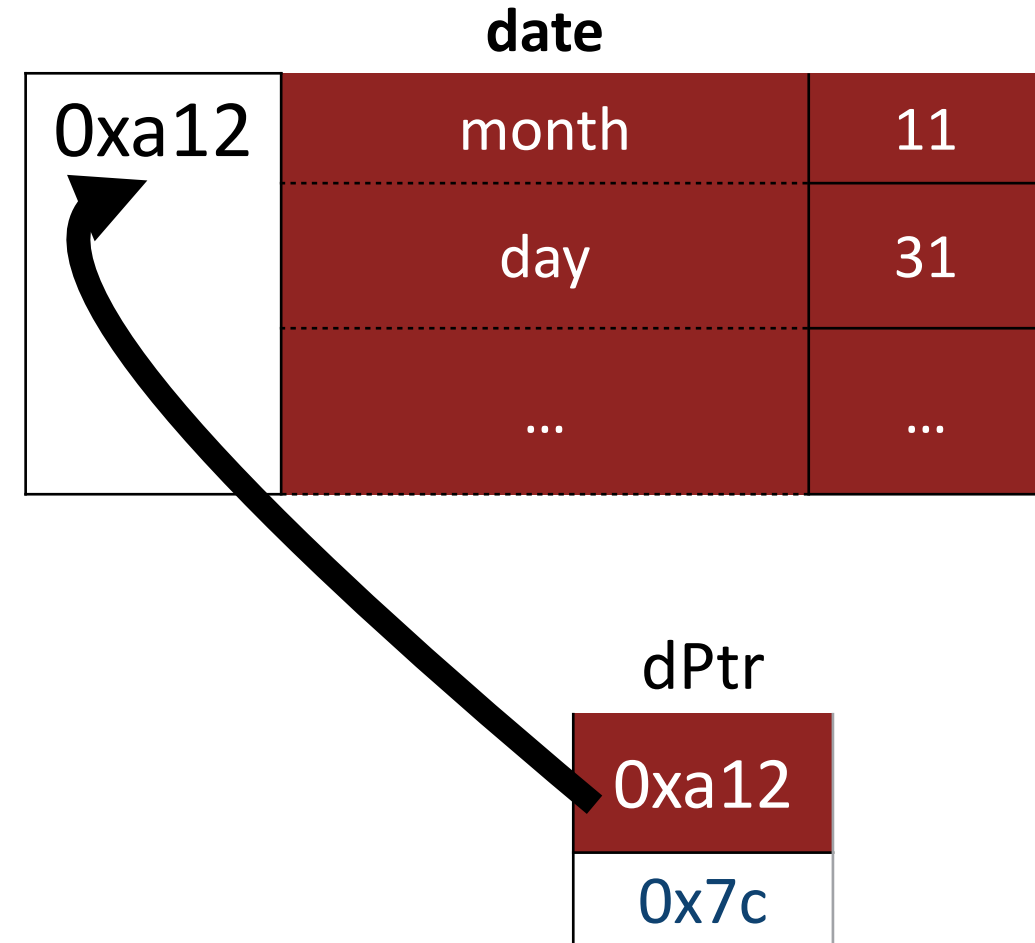
```
Date d;
```

```
d.month = 7;
```

```
Date* dPtr = &d;
```

```
cout << (*dPtr).month << endl;
```

- But, this notation is cumbersome, and the parenthesis are necessary because the "dot" has a higher precedence than the *.



Pointer and Arrow Syntax

- So, we have a different, and more intuitive syntax, called the "arrow" syntax, -> :

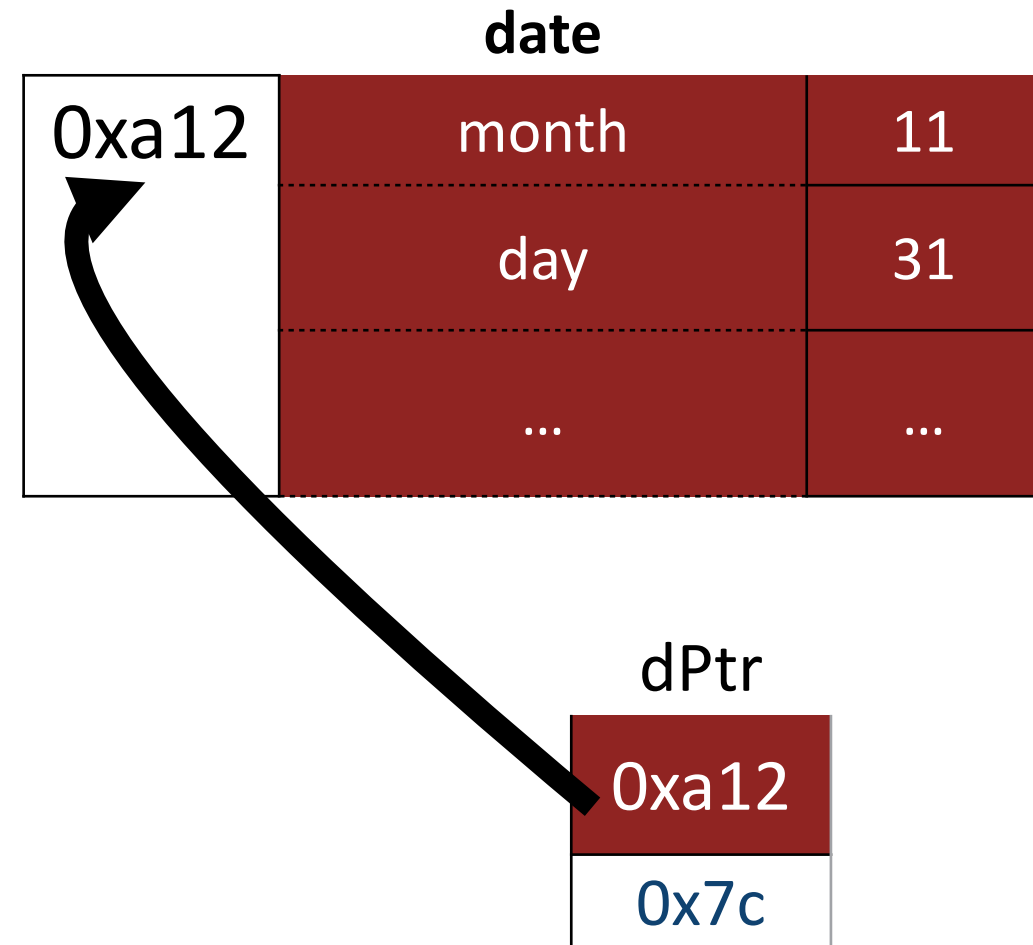
```
Date d;
```

```
d.month = 7;
```

```
Date* dPtr = &d;
```

```
cout << dPtr->month << endl;
```

- The arrow syntax can be used to set a value as well



Dynamic Memory for Struct: new and delete

// allocate a Date struct variable, default initializer

```
Date * d1 = new Date;
```

// allocate a Date struct variable, initialize the members

```
Date * d2 = new Date {2023, 11, 30}; // C++11
```

Dynamic Memory for Struct: new and delete

// allocate a Date struct variable, default initializer

```
Date * d1 = new Date;
```

// allocate a Date struct variable, initialize the members

```
Date * d2 = new Date {2023, 11, 30}; // C++11
```

// deallocate memory

```
delete d1;
```

// deallocate memory

```
delete d2;
```

Dynamic Memory for Struct: new and delete

// allocate memory for 16 Date objects, default initializer

```
Date * da1 = new Date [16];
```

// allocate memory for 16 Date objects, the first two are explicitly initialized

```
Date * da2 = new Date [16]{{2023, 11, 30}, {2023, 12, 1}}; // C++11
```

Dynamic Memory for Struct: new and delete

```
// allocate memory for 16 Date objects, default initializer
```

```
Date * da1 = new Date [16];
```

```
// allocate memory for 16 Date objects, the first two are explicitly initialized
```

```
Date * da2 = new Date [16]{{2023, 11, 30}, {2023, 12, 1}}; // C++11
```

```
// deallocate the memory of da1
```

```
delete []da1;
```

```
// deallocate the memory of da2
```

```
delete []da2;
```

Pass Structure to Function

- Pass structure **by value**

```
void foo(CS2311Student sr) {  
    sr.sid = 0;  
}  
  
int main() {  
    CS2311Student sr;  
    sr.sid = 2311;  
    foo(sr);  
    cout << sr.sid << endl;  
    return 0;  
}
```

Pass Structure to Function

- Pass structure **by pointer**

```
void foo(CS2311Student *sr) {  
    sr->sid = 0;  
}
```

```
int main() {  
    CS2311Student sr;  
    sr.sid = 2311;  
    foo(&sr);  
    cout << sr.sid << endl;  
    return 0;  
}
```

Return Structure from Function

- A function can return a value of structure type:

```
CS2311Student newStudent(int sid) {  
    CS2311Student stu;  
    stu.sid = sid;  
    return stu;  
}
```

Return Structure from Function

- A function can return a value of structure type:

```
CS2311Student newStudent(int sid) {  
    CS2311Student stu;  
    stu.sid = sid;  
    return stu;  
}
```

```
CS2311Student* newStudent(int sid) { // More efficient, why?  
    CS2311 *stu = new CS2311Student;  
    stu->sid = sid;  
    return stu;  
}
```

Array of Structure

```
const int MAX_CLASS_SIZE 1024
```

```
CS2311Student sr[MAX_CLASS_SIZE];
```

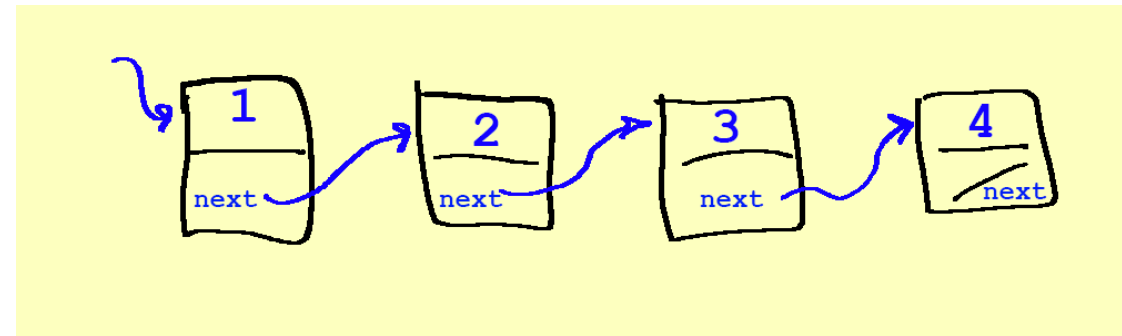
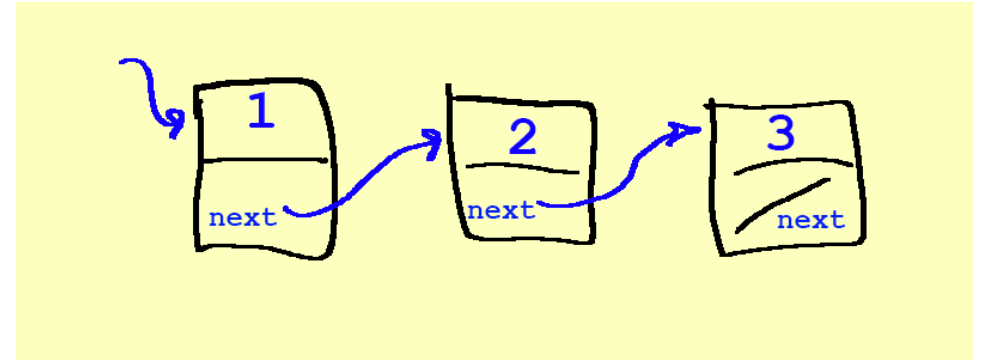
```
int num_of_students = 80;
```

```
// assume sr is sorted based on sr->sid;
```

```
// how to insert a new student?
```

Struct and Pointer: Singly Linked List

- A singly linked list is a chain of *nodes*
- Each node contains two pieces of information:
 - Some piece of **data** that is stored in the sequence
 - A **link to the next node** in the list
- We can *traverse* the list by starting at the first node and repeatedly following its link
- Each element is **stored separately** from the rest.
- The elements are then chained together into a sequence



Singly Linked List: Insert In Front

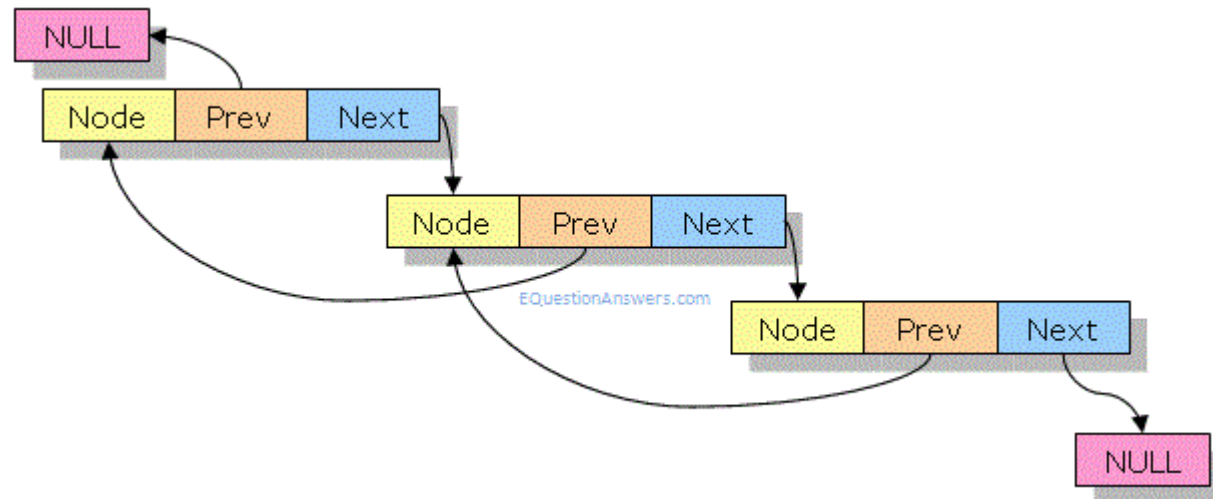
```
struct Node {  
    int data;  
    Node* next;  
};
```

```
// Function to print the linked list  
void printList(Node* head) {  
    while (head != NULL) {  
        cout << head->data << " -> ";  
        head = head->next;  
    }  
    cout << "NULL\n";  
}
```

```
void insertAtBeginning(Node*& head, int newData) {  
    Node* newNode = new Node();  
    newNode->data = newData;  
    newNode->next = head;  
    head = newNode;  
}  
  
void main() {  
    Node* head = NULL; // Start with an empty list  
    insertAtBeginning(head, 10);  
    insertAtBeginning(head, 20);  
    cout << "Linked List after insertion: ";  
    printList(head);  
    // Free the remaining nodes  
    while (head != NULL) {  
        Node* temp = head;  
        head = head->next;  
        delete temp;  
    }  
}
```

Struct and Pointer: Doubly Linked List

- In a doubly linked list, each node contains three pieces of information:
 - Some piece of **data** that is stored in the sequence
 - A **link to the next node** in the list
 - A **link to the prev node** in the list
- We can *traverse* the list in both forward and backward directions



Linked List vs Array

- Storage overhead?
- Insert/delete efficiency?
- Search efficiency?

Linked List vs Array

- Storage overhead? Array is better
- Insert/delete efficiency? Linked list is better
- Search efficiency? Array is better