

FROM STRUCT TO CLASS

UNDERSTANDING C++ ENCAPSULATION

Lecturer: 陈笑沙

REVIEW

Which line of code has an error in the following program?

```
1 int a = 100, b = 10;
2 const int *p1 = &a;
3 p1 = &b;
4 *p1 = 0;
5 int * const p2 = &a;
6 *p2 = 0;
7 p2 = &b;
8 const int * const p3 = &a;
9 *p3 = 0;
10 p3 = &b
```

REVIEW

Which line of code has an error in the following program?

```
1 int a = 100, b = 10;
2 const int *p1 = &a;
3 p1 = &b;
4 *p1 = 0;
5 int * const p2 = &a;
6 *p2 = 0;
7 p2 = &b;
8 const int * const p3 = &a;
9 *p3 = 0;
10 p3 = &b
```

REVIEW

Which line of code has an error in the following program?

```
1 int a = 100, b = 10;
2 const int *p1 = &a;
3 p1 = &b;
4 *p1 = 0;
5 int * const p2 = &a;
6 *p2 = 0;
7 p2 = &b;
8 const int * const p3 = &a;
9 *p3 = 0;
10 p3 = &b
```

TABLE OF CONTENTS

- 3.1 Concept and Operation of Structures
- 3.2 Structures and Pointers
- 3.3 Structures and Arrays
- 3.4 Linked Lists
- 3.5 From Structures to Classes
- 3.6 Member Functions
- 3.7 Hiding the Internal Implementation of Classes

3.1 CONCEPT AND OPERATION OF STRUCTURES

3.1 CONCEPT AND OPERATION OF STRUCTURES

- The limitation of single-value data types

3.1 CONCEPT AND OPERATION OF STRUCTURES

- The limitation of single-value data types
 - Object attributes are often not single values, and batch processing is more complex.

3.1 CONCEPT AND OPERATION OF STRUCTURES

- The limitation of single-value data types
 - Object attributes are often not single values, and batch processing is more complex.
 - For example: dates, persons, etc.

3.1 CONCEPT AND OPERATION OF STRUCTURES

- The limitation of single-value data types
 - Object attributes are often not single values, and batch processing is more complex.
 - For example: dates, persons, etc.
- Solution: structure

3.1 CONCEPT AND OPERATION OF STRUCTURES

- The limitation of single-value data types
 - Object attributes are often not single values, and batch processing is more complex.
 - For example: dates, persons, etc.
- Solution: structure

```
1 struct Date
2 {
3     int month;
4     int day;
5     int year;
6 }; // Note: semicolon at the end
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

- The limitation of single-value data types
 - Object attributes are often not single values, and batch processing is more complex.
 - For example: dates, persons, etc.
- Solution: structure

```
1 struct Date
2 {
3     int month;
4     int day;
5     int year;
6 }; // Note: semicolon at the end
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

- The limitation of single-value data types
 - Object attributes are often not single values, and batch processing is more complex.
 - For example: dates, persons, etc.
- Solution: structure

```
1 struct Date
2 {
3     int month;
4     int day;
5     int year;
6 }; // Note: semicolon at the end
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

- The limitation of single-value data types
 - Object attributes are often not single values, and batch processing is more complex.
 - For example: dates, persons, etc.
- Solution: structure

```
1 struct Date
2 {
3     int month;
4     int day;
5     int year;
6 }; // Note: semicolon at the end
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

First declare the structure type, then define variables of that type.

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender; // 'M' or 'F'  
5     unsigned int age;  
6     double score;  
7     std::string address;  
8 };  
9 Student student1, student2;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

First declare the structure type, then define variables of that type.

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender; // 'M' or 'F'  
5     unsigned int age;  
6     double score;  
7     std::string address;  
8 };  
9 Student student1, student2;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

First declare the structure type, then define variables of that type.

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender; // 'M' or 'F'  
5     unsigned int age;  
6     double score;  
7     std::string address;  
8 };  
9 Student student1, student2;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

First declare the structure type, then define variables of that type.

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender; // 'M' or 'F'  
5     unsigned int age;  
6     double score;  
7     std::string address;  
8 };  
9 Student student1, student2;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

First declare the structure type, then define variables of that type.

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender; // 'M' or 'F'  
5     unsigned int age;  
6     double score;  
7     std::string address;  
8 };  
9 Student student1, student2;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

3.1 CONCEPT AND OPERATION OF STRUCTURES

- Structure type and structure variable are different concepts

3.1 CONCEPT AND OPERATION OF STRUCTURES

- Structure type and structure variable are different concepts
 - What is a type? What is a variable?

3.1 CONCEPT AND OPERATION OF STRUCTURES

- Structure type and structure variable are different concepts
 - What is a type? What is a variable?
 - It can be simply understood that: type is a collection, variable is an element in the collection

3.1 CONCEPT AND OPERATION OF STRUCTURES

- Structure type and structure variable are different concepts
 - What is a type? What is a variable?
 - It can be simply understood that: type is a collection, variable is an element in the collection
- Member names in a structure type can be the same as variable names, but they have different meanings

3.1 CONCEPT AND OPERATION OF STRUCTURES

- Structure type and structure variable are different concepts
 - What is a type? What is a variable?
 - It can be simply understood that: type is a collection, variable is an element in the collection
- Member names in a structure type can be the same as variable names, but they have different meanings
- Member variables can be used individually, and their role and status are equivalent to ordinary variables

3.1 CONCEPT AND OPERATION OF STRUCTURES

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender;  
5     std::string address;  
6 };  
7  
8 Student a = {"10101", "Alice", 'F', "Taiyuan"};  
9 std::cout << a.id << ", "  
10            << a.name << ", "  
11            << a.gender << ", "  
12            << a.address << std::endl;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender;  
5     std::string address;  
6 };  
7  
8 Student a = {"10101", "Alice", 'F', "Taiyuan"};  
9 std::cout << a.id << ", "  
10            << a.name << ", "  
11            << a.gender << ", "  
12            << a.address << std::endl;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender;  
5     std::string address;  
6 };  
7  
8 Student a = {"10101", "Alice", 'F', "Taiyuan"};  
9 std::cout << a.id << ", "  
10            << a.name << ", "  
11            << a.gender << ", "  
12            << a.address << std::endl;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

```
1 struct Student {  
2     std::string id;  
3     std::string name;  
4     char gender;  
5     std::string address;  
6 };  
7  
8 Student a = {"10101", "Alice", 'F', "Taiyuan"};  
9 std::cout << a.id << ", "  
10            << a.name << ", "  
11            << a.gender << ", "  
12            << a.address << std::endl;
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

Another method (not recommended)

```
1 struct Student {  
2     std::string name;  
3     char gender;  
4 } stu = {"Bob", 'M';
```

3.1 CONCEPT AND OPERATION OF STRUCTURES

Another method (not recommended)

```
1 struct Student {  
2     std::string name;  
3     char gender;  
4 } stu = {"Bob", 'M';
```

3.2 STRUCTURES AND POINTERS

The structure is just a way to organize data: the actual data is stored in member variables. However, there can be pointers to structures

3.2 STRUCTURES AND POINTERS

The structure is just a way to organize data: the actual data is stored in member variables. However, there can be pointers to structures

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct Person {
6     string name;
7     string id;
8     double salary;
9 }
10
11 int main() {
12     Person person1, person2;
13     Person *ptr1 = &person1;
14     ptr1->name = "Eric";
15     ptr1->id = "1234004321";
```

3.2 STRUCTURES AND POINTERS

The structure is just a way to organize data: the actual data is stored in member variables. However, there can be pointers to structures

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct Person {
6     string name;
7     string id;
8     double salary;
9 }
10
11 int main() {
12     Person person1, person2;
13     Person *ptr1 = &person1;
14     ptr1->name = "Eric";
15     ptr1->id = "1234004321";
```

3.2 STRUCTURES AND POINTERS

The structure is just a way to organize data: the actual data is stored in member variables. However, there can be pointers to structures

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct Person {
6     string name;
7     string id;
8     double salary;
9 }
10
11 int main() {
12     Person person1, person2;
13     Person *ptr1 = &person1;
14     ptr1->name = "Eric";
15     ptr1->id = "1234004321";
```

3.2 STRUCTURES AND POINTERS

The structure is just a way to organize data: the actual data is stored in member variables. However, there can be pointers to structures

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct Person {
6     string name;
7     string id;
8     double salary;
9 }
10
11 int main() {
12     Person person1, person2;
13     Person *ptr1 = &person1;
14     ptr1->name = "Eric";
15     ptr1->id = "1234004321";
```

3.2 STRUCTURES AND POINTERS

The structure is just a way to organize data: the actual data is stored in member variables. However, there can be pointers to structures

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct Person {
6     string name;
7     string id;
8     double salary;
9 }
10
11 int main() {
12     Person person1, person2;
13     Person *ptr1 = &person1;
14     ptr1->name = "Eric";
15     ptr1->id = "1234004321";
```

3.2 STRUCTURES AND POINTERS

The structure is just a way to organize data: the actual data is stored in member variables. However, there can be pointers to structures

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct Person {
6     string name;
7     string id;
8     double salary;
9 }
10
11 int main() {
12     Person person1, person2;
13     Person *ptr1 = &person1;
14     ptr1->name = "Eric";
15     ptr1->id = "1234004321";
```

3.2 STRUCTURES AND POINTERS

The structure is just a way to organize data: the actual data is stored in member variables. However, there can be pointers to structures

```
1 #include <iostream>
2
3 using namespace std;
4
5 struct Person {
6     string name;
7     string id;
8     double salary;
9 }
10
11 int main() {
12     Person person1, person2;
13     Person *ptr1 = &person1;
14     ptr1->name = "Eric";
15     ptr1->id = "1234004321";
```

3.2 STRUCTURES AND POINTERS

If you need to pass a structure as a function parameter, generally use pointers or references. Otherwise, performance issues may occur.

3.3 STRUCTURES AND ARRAYS

The same usage as basic data types.

```
1 struct Person {  
2     string name;  
3     unsigned int age;  
4 };  
5  
6 int main() {  
7     Person group[3] = {  
8         {"Bob", 20},  
9         {"Alice", 30},  
10        {"Eric", 40}};  
11    Person *p = group;  
12    return 0;  
13 }
```

3.3 STRUCTURES AND ARRAYS

The same usage as basic data types.

```
1 struct Person {
2     string name;
3     unsigned int age;
4 };
5
6 int main() {
7     Person group[3] = {
8         {"Bob", 20},
9         {"Alice", 30},
10        {"Eric", 40}};
11    Person *p = group;
12    return 0;
13 }
```

3.3 STRUCTURES AND ARRAYS

The same usage as basic data types.

```
1 struct Person {
2     string name;
3     unsigned int age;
4 };
5
6 int main() {
7     Person group[3] = {
8         {"Bob", 20},
9         {"Alice", 30},
10        {"Eric", 40}};
11    Person *p = group;
12    return 0;
13 }
```

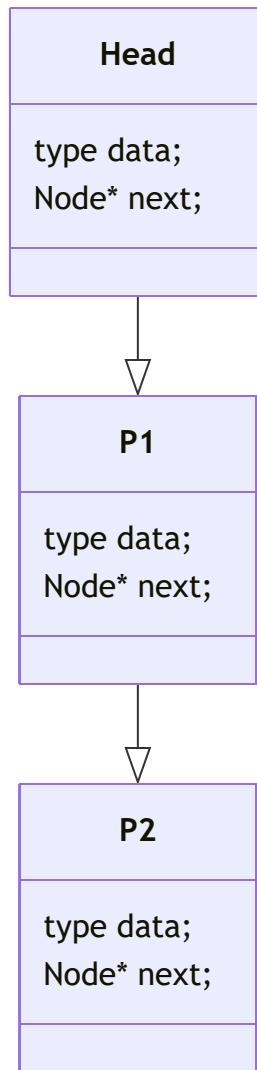
3.3 STRUCTURES AND ARRAYS

The same usage as basic data types.

```
1 struct Person {
2     string name;
3     unsigned int age;
4 };
5
6 int main() {
7     Person group[3] = {
8         {"Bob", 20},
9         {"Alice", 30},
10        {"Eric", 40}};
11    Person *p = group;
12    return 0;
13 }
```

3.4 LINKED LISTS

WHAT IS A LINKED LIST?



3.4 LINKED LISTS

WHAT IS A LINKED LIST?

3.4 LINKED LISTS

WHAT IS A LINKED LIST?

- A linked list consists of several nodes

3.4 LINKED LISTS

WHAT IS A LINKED LIST?

- A linked list consists of several nodes
- Each node contains two members:

3.4 LINKED LISTS

WHAT IS A LINKED LIST?

- A linked list consists of several nodes
- Each node contains two members:
 - Data (can be other structures)

3.4 LINKED LISTS

WHAT IS A LINKED LIST?

- A linked list consists of several nodes
- Each node contains two members:
 - Data (can be other structures)
 - Pointer to the next node

3.4 LINKED LISTS

WHAT IS A LINKED LIST?

- A linked list consists of several nodes
- Each node contains two members:
 - Data (can be other structures)
 - Pointer to the next node
- A null pointer represents an empty linked list

3.4 LINKED LISTS

```
1 struct Node {  
2     int data;  
3     Node* next;  
4 };
```

3.4 LINKED LISTS

```
1 struct Node {  
2     int data;  
3     Node* next;  
4 };
```

3.4 LINKED LISTS

Create a linked list:

```
Node* makeList() {  
    return nullptr; // Do Not use NULL  
}
```

3.4 LINKED LISTS

prepend:

```
1 Node *prepend(Node *list, const int element) {  
2     Node *head = new Node();  
3     head->data = element;  
4     head->next = list;  
5     return head;  
6 }
```

3.4 LINKED LISTS

prepend:

```
1 Node *prepend(Node *list, const int element) {  
2     Node *head = new Node();  
3     head->data = element;  
4     head->next = list;  
5     return head;  
6 }
```

3.4 LINKED LISTS

prepend:

```
1 Node *prepend(Node *list, const int element) {  
2     Node *head = new Node();  
3     head->data = element;  
4     head->next = list;  
5     return head;  
6 }
```

3.4 LINKED LISTS

prepend:

```
1 Node *prepend(Node *list, const int element) {  
2     Node *head = new Node();  
3     head->data = element;  
4     head->next = list;  
5     return head;  
6 }
```

3.4 LINKED LISTS

prepend:

```
1 Node *prepend(Node *list, const int element) {  
2     Node *head = new Node();  
3     head->data = element;  
4     head->next = list;  
5     return head;  
6 }
```

3.4 LINKED LISTS

print:

```
1 void printList(const Node *list) {  
2     std::cout << "[";  
3     const Node *head = list;  
4     while (head != nullptr) {  
5         std::cout << head->data;  
6         if (head->next != nullptr) {  
7             std::cout << ", ";  
8         }  
9         head = head->next;  
10    }  
11    std::cout << "]";  
12 }
```

3.4 LINKED LISTS

print:

```
1 void printList(const Node *list) {  
2     std::cout << "[";  
3     const Node *head = list;  
4     while (head != nullptr) {  
5         std::cout << head->data;  
6         if (head->next != nullptr) {  
7             std::cout << ", ";  
8         }  
9         head = head->next;  
10    }  
11    std::cout << "]";  
12 }
```

3.4 LINKED LISTS

print:

```
1 void printList(const Node *list) {  
2     std::cout << "[";  
3     const Node *head = list;  
4     while (head != nullptr) {  
5         std::cout << head->data;  
6         if (head->next != nullptr) {  
7             std::cout << ", ";  
8         }  
9         head = head->next;  
10    }  
11    std::cout << "]";  
12 }
```

3.4 LINKED LISTS

print:

```
1 void printList(const Node *list) {  
2     std::cout << "[";  
3     const Node *head = list;  
4     while (head != nullptr) {  
5         std::cout << head->data;  
6         if (head->next != nullptr) {  
7             std::cout << ", ";  
8         }  
9         head = head->next;  
10    }  
11    std::cout << "]";  
12 }
```

3.4 LINKED LISTS

print:

```
1 void printList(const Node *list) {  
2     std::cout << "[";  
3     const Node *head = list;  
4     while (head != nullptr) {  
5         std::cout << head->data;  
6         if (head->next != nullptr) {  
7             std::cout << ", ";  
8         }  
9         head = head->next;  
10    }  
11    std::cout << "]";  
12 }
```

3.4 LINKED LISTS

print:

```
1 void printList(const Node *list) {  
2     std::cout << "[";  
3     const Node *head = list;  
4     while (head != nullptr) {  
5         std::cout << head->data;  
6         if (head->next != nullptr) {  
7             std::cout << ", ";  
8         }  
9         head = head->next;  
10    }  
11    std::cout << "]";  
12 }
```

3.4 LINKED LISTS

print:

```
1 void printList(const Node *list) {  
2     std::cout << "[";  
3     const Node *head = list;  
4     while (head != nullptr) {  
5         std::cout << head->data;  
6         if (head->next != nullptr) {  
7             std::cout << ", ";  
8         }  
9         head = head->next;  
10    }  
11    std::cout << "]";  
12 }
```

3.4 LINKED LISTS

print:

```
1 void printList(const Node *list) {  
2     std::cout << "[";  
3     const Node *head = list;  
4     while (head != nullptr) {  
5         std::cout << head->data;  
6         if (head->next != nullptr) {  
7             std::cout << ", ";  
8         }  
9         head = head->next;  
10    }  
11    std::cout << "]";  
12 }
```

3.4 LINKED LISTS

free:

```
1 void freeList(Node *list) {  
2     while (list != nullptr) {  
3         Node *next = list->next;  
4         delete list;  
5         list = next;  
6     }  
7 }
```

3.4 LINKED LISTS

free:

```
1 void freeList(Node *list) {  
2     while (list != nullptr) {  
3         Node *next = list->next;  
4         delete list;  
5         list = next;  
6     }  
7 }
```

3.4 LINKED LISTS

free:

```
1 void freeList(Node *list) {  
2     while (list != nullptr) {  
3         Node *next = list->next;  
4         delete list;  
5         list = next;  
6     }  
7 }
```

3.4 LINKED LISTS

free:

```
1 void freeList(Node *list) {  
2     while (list != nullptr) {  
3         Node *next = list->next;  
4         delete list;  
5         list = next;  
6     }  
7 }
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

insert:

```
1 Node *insert(Node *list, int element, size_t index) {
2     if (list == nullptr) {
3         // empty list
4         if (index == 0) {
5             Node *newNode = new Node();
6             newNode->data = element;
7             newNode->next = nullptr;
8             return newNode;
9         }
10        return list; // Fail, should throw exception
11    }
12
13    Node *prev = nullptr;
14    Node *next = list;
15    while (index > 0) {
```

3.4 LINKED LISTS

delete, length, find ...

3.4 LINKED LISTS

delete, length, find ...

- If you need a linked list, it is recommended to use `std :: list` (double-linked list)

3.4 LINKED LISTS

delete, length, find ...

- If you need a linked list, it is recommended to use **std :: list** (double-linked list)
- If you need to implement it manually, it is recommended to use **std :: shared_ptr** (smart pointer)

3.4 LINKED LISTS

delete, length, find ...

- If you need a linked list, it is recommended to use `std::list` (double-linked list)
- If you need to implement it manually, it is recommended to use `std::shared_ptr` (smart pointer)
- Extended reading: [Cons List](#), reference implementation: `demo/consList`

3.4 LINKED LISTS

delete, length, find ...

- If you need a linked list, it is recommended to use `std::list` (double-linked list)
- If you need to implement it manually, it is recommended to use `std::shared_ptr` (smart pointer)
- Extended reading: [Cons List](#), reference implementation: `demo/consList`

`example/lec03/list`

3.4 LINKED LISTS

delete, length, find ...

- If you need a linked list, it is recommended to use `std::list` (double-linked list)
- If you need to implement it manually, it is recommended to use `std::shared_ptr` (smart pointer)
- Extended reading: [Cons List](#), reference implementation: `demo/consList`

`example/lec03/list`

`example/lec03/smartList`

3.5 FROM STRUCTURES TO CLASSES

3.5 FROM STRUCTURES TO CLASSES

- Some functions need to manipulate specific data, and the data needs to be passed in each time:

3.5 FROM STRUCTURES TO CLASSES

- Some functions need to manipulate specific data, and the data needs to be passed in each time:
 - `Node *prepend(Node *list, ...)`

3.5 FROM STRUCTURES TO CLASSES

- Some functions need to manipulate specific data, and the data needs to be passed in each time:
 - `Node *prepend(Node *list, ...)`
 - `Node *insert(Node *list, ...)`

3.5 FROM STRUCTURES TO CLASSES

- Some functions need to manipulate specific data, and the data needs to be passed in each time:
 - `Node *prepend(Node *list, ...)`
 - `Node *insert(Node *list, ...)`
 - `void freeList(Node *list)`

3.5 FROM STRUCTURES TO CLASSES

- Some functions need to manipulate specific data, and the data needs to be passed in each time:
 - `Node *prepend(Node *list, ...)`
 - `Node *insert(Node *list, ...)`
 - `void freeList(Node *list)`
- At the language level, data and its operations are not associated

3.5 FROM STRUCTURES TO CLASSES

3.5 FROM STRUCTURES TO CLASSES

- Some data do not want to be exposed to the caller
(internal implementation)

3.5 FROM STRUCTURES TO CLASSES

- Some data do not want to be exposed to the caller (internal implementation)
 - Internal changes do not affect external calls

3.5 FROM STRUCTURES TO CLASSES

- Some data do not want to be exposed to the caller (internal implementation)
 - Internal changes do not affect external calls
 - External errors are easier to debug

3.5 FROM STRUCTURES TO CLASSES

Introducing the concept of classes:

3.5 FROM STRUCTURES TO CLASSES

Introducing the concept of classes:

- Data and operations are bound together

3.5 FROM STRUCTURES TO CLASSES

Introducing the concept of classes:

- Data and operations are bound together
- Control over data access

3.5 FROM STRUCTURES TO CLASSES

```
class Point {  
    // data field, default is private  
    int x;  
    int y;  
public:  
    // Functions in class (methods)  
    void print() {  
        cout << '(' << x << ", " << ')';  
    }  
};
```

3.5 FROM STRUCTURES TO CLASSES

3.5 FROM STRUCTURES TO CLASSES

- A class is a data type, and an object is an instance of the corresponding class

3.5 FROM STRUCTURES TO CLASSES

- A class is a data type, and an object is an instance of the corresponding class
 - An object is a variable of the corresponding data type

3.5 FROM STRUCTURES TO CLASSES

- A class is a data type, and an object is an instance of the corresponding class
 - An object is a variable of the corresponding data type
- A class is a blueprint

3.5 FROM STRUCTURES TO CLASSES

- A class is a data type, and an object is an instance of the corresponding class
 - An object is a variable of the corresponding data type
- A class is a blueprint
 - An object is a specific entity made according to the blueprint

3.5 FROM STRUCTURES TO CLASSES

3.5 FROM STRUCTURES TO CLASSES

- For compatibility reasons, `c++` allows methods to be added to `struct`.

3.5 FROM STRUCTURES TO CLASSES

- For compatibility reasons, `c++` allows methods to be added to `struct`.
- Member variables in `struct` have `public` access by default.

3.6 MEMBER FUNCTIONS

3.6 MEMBER FUNCTIONS

- A class contains member variables and member functions

3.6 MEMBER FUNCTIONS

- A class contains member variables and member functions
- Member variables and member functions are collectively referred to as members of the class

3.6 MEMBER FUNCTIONS

- A class contains member variables and member functions
- Member variables and member functions are collectively referred to as members of the class
- Member variables are sometimes called properties, and member functions are sometimes called methods

3.6 MEMBER FUNCTIONS

- A class contains member variables and member functions
- Member variables and member functions are collectively referred to as members of the class
- Member variables are sometimes called properties, and member functions are sometimes called methods
- They are not required

3.6 MEMBER FUNCTIONS

3.6 MEMBER FUNCTIONS

- In the definition of a class, use the following access range keywords to specify the accessibility of class members:

3.6 MEMBER FUNCTIONS

- In the definition of a class, use the following access range keywords to specify the accessibility of class members:
 - **private**: Private members, can only be accessed within member functions

3.6 MEMBER FUNCTIONS

- In the definition of a class, use the following access range keywords to specify the accessibility of class members:
 - **private**: Private members, can only be accessed within member functions
 - **public** : Public members, can be accessed anywhere

3.6 MEMBER FUNCTIONS

- In the definition of a class, use the following access range keywords to specify the accessibility of class members:
 - **private**: Private members, can only be accessed within member functions
 - **public** : Public members, can be accessed anywhere
 - **protected**: Protected members, will be explained later

3.6 MEMBER FUNCTIONS

- In the definition of a class, use the following access range keywords to specify the accessibility of class members:
 - **private**: Private members, can only be accessed within member functions
 - **public** : Public members, can be accessed anywhere
 - **protected**: Protected members, will be explained later
- The number of occurrences and order of the above three keywords are not limited.

3.6 MEMBER FUNCTIONS

How to define a class?

```
class className {  
private:  
    // Private properties and methods  
public:  
    // Public properties and methods  
protected:  
    // Protected properties and methods  
}; // Remember semicolon
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {
2 private:
3     int w, h;
4 public:
5     int area() {
6         return w * h;
7     }
8     int perimeter() {
9         return 2 * (w + h);
10    }
11    void set(int width, int height) {
12        w = width;
13        h = height;
14    }
15};
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {
2 private:
3     int w, h;
4 public:
5     int area() {
6         return w * h;
7     }
8     int perimeter() {
9         return 2 * (w + h);
10    }
11    void set(int width, int height) {
12        w = width;
13        h = height;
14    }
15};
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {  
2     private:  
3         int w, h;  
4     public:  
5         int area() {  
6             return w * h;  
7         }  
8         int perimeter() {  
9             return 2 * (w + h);  
10        }  
11        void set(int width, int height) {  
12            w = width;  
13            h = height;  
14        }  
15    };
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {  
2 private:  
3     int w, h;  
4 public:  
5     int area() {  
6         return w * h;  
7     }  
8     int perimeter() {  
9         return 2 * (w + h);  
10    }  
11    void set(int width, int height) {  
12        w = width;  
13        h = height;  
14    }  
15};
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {  
2 private:  
3     int w, h;  
4 public:  
5     int area() {  
6         return w * h;  
7     }  
8     int perimeter() {  
9         return 2 * (w + h);  
10    }  
11    void set(int width, int height) {  
12        w = width;  
13        h = height;  
14    }  
15};
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {  
2 private:  
3     int w, h;  
4 public:  
5     int area() {  
6         return w * h;  
7     }  
8     int perimeter() {  
9         return 2 * (w + h);  
10    }  
11    void set(int width, int height) {  
12        w = width;  
13        h = height;  
14    }  
15};
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {  
2 private:  
3     int w, h;  
4 public:  
5     int area() {  
6         return w * h;  
7     }  
8     int perimeter() {  
9         return 2 * (w + h);  
10    }  
11    void set(int width, int height) {  
12        w = width;  
13        h = height;  
14    }  
15};
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {
2 private:
3     int w, h;
4 public:
5     int area() {
6         return w * h;
7     }
8     int perimeter() {
9         return 2 * (w + h);
10    }
11    void set(int width, int height) {
12        w = width;
13        h = height;
14    }
15};
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {  
2 private:  
3     int w, h;  
4 public:  
5     int area() {  
6         return w * h;  
7     }  
8     int perimeter() {  
9         return 2 * (w + h);  
10    }  
11    void set(int width, int height) {  
12        w = width;  
13        h = height;  
14    }  
15};
```

3.6 MEMBER FUNCTIONS

```
1 class Rectangle {  
2 private:  
3     int w, h;  
4 public:  
5     int area() {  
6         return w * h;  
7     }  
8     int perimeter() {  
9         return 2 * (w + h);  
10    }  
11    void set(int width, int height) {  
12        w = width;  
13        h = height;  
14    }  
15};
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2     private:  
3         double x, y;  
4     public:  
5         void set(double xCoord, double yCoord) {  
6             x = xCoord;  
7             y = yCoord;  
8         }  
9  
10        double length() {  
11            return std::sqrt(x * x + y * y);  
12        }  
13  
14        double angle() {  
15            return std::atan2(y, x);  
16        }  
17    }  
18}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2     private:  
3         double x, y;  
4     public:  
5         void set(double xCoord, double yCoord) {  
6             x = xCoord;  
7             y = yCoord;  
8         }  
9  
10        double length() {  
11            return std::sqrt(x * x + y * y);  
12        }  
13  
14        double angle() {  
15            return std::atan2(y, x);  
16        }  
17    }  
18}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double x, y;  
4 public:  
5     void set(double xCoord, double yCoord) {  
6         x = xCoord;  
7         y = yCoord;  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }  
17}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double x, y;  
4 public:  
5     void set(double xCoord, double yCoord) {  
6         x = xCoord;  
7         y = yCoord;  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }  
17}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double x, y;  
4 public:  
5     void set(double xCoord, double yCoord) {  
6         x = xCoord;  
7         y = yCoord;  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }  
17}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double x, y;  
4 public:  
5     void set(double xCoord, double yCoord) {  
6         x = xCoord;  
7         y = yCoord;  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }  
17}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double x, y;  
4 public:  
5     void set(double xCoord, double yCoord) {  
6         x = xCoord;  
7         y = yCoord;  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }  
17}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double r, a;  
4 public:  
5     void set(double xCoord, double yCoord) {  
6         x = xCoord;  
7         y = yCoord;  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }  
17}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double r, a;  
4 public:  
5     void set(double xCoord, double yCoord) {  
6         x = xCoord;  
7         y = yCoord;  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }  
17}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double r, a;  
4 public:  
5     void set(double x, double y) {  
6         r = std::sqrt(x * x + y * y);  
7         a = std::atan2(y, x);  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2 private:  
3     double r, a;  
4 public:  
5     void set(double x, double y) {  
6         r = std::sqrt(x * x + y * y);  
7         a = std::atan2(y, x);  
8     }  
9  
10    double length() {  
11        return std::sqrt(x * x + y * y);  
12    }  
13  
14    double angle() {  
15        return std::atan2(y, x);  
16    }
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2     private:  
3         double r, a;  
4     public:  
5         void set(double x, double y) {  
6             r = std::sqrt(x * x + y * y);  
7             a = std::atan2(y, x);  
8         }  
9  
10        double length() {  
11            return r;  
12        }  
13  
14        double angle() {  
15            return std::atan2(y, x);  
16        }  
17    }  
18}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2     private:  
3         double r, a;  
4     public:  
5         void set(double x, double y) {  
6             r = std::sqrt(x * x + y * y);  
7             a = std::atan2(y, x);  
8         }  
9  
10        double length() {  
11            return r;  
12        }  
13  
14        double angle() {  
15            return std::atan2(y, x);  
16        }  
17    }  
18}
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

```
1 class Point {  
2     private:  
3         double r, a;  
4     public:  
5         void set(double x, double y) {  
6             r = std::sqrt(x * x + y * y);  
7             a = std::atan2(y, x);  
8         }  
9  
10        double length() {  
11            return r;  
12        }  
13  
14        double angle() {  
15            return a;
```

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

- Internal changes, interface unchanged, will not affect external programs

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

- Internal changes, interface unchanged, will not affect external programs
 - Decoupling

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

- Internal changes, interface unchanged, will not affect external programs
 - Decoupling
- Easier to locate errors

3.7 HIDING THE INTERNAL IMPLEMENTATION OF CLASSES

- Internal changes, interface unchanged, will not affect external programs
 - Decoupling
- Easier to locate errors
 - No need to consider that external programs may have changed private properties without knowing

HOMEWORK

Create a linked list that stores chars with structures.

Need to implement following methods:

- `void display(List*)`
- `void prepend(List*, char c)`
- `void append(List*, char c)`
- `int size(List*)`
- `void delete(List*, int index)`
- `void free(List*)`

Do not use AI, I can tell the differences between AI generate codes and yours. Do not copy.