

# **从结构体到类**

## **理解 C++ 封装**

**主讲：陈笑沙**

# 复习

以下程序，哪行代码有错误？

```
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
```

# 复习

以下程序，哪行代码有错误？

```
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
```

# 复习

以下程序，哪行代码有错误？

```
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
```

# 目录

- 3.1 结构的概念与操作
- 3.2 结构与指针
- 3.3 结构与数组
- 3.4 链表
- 3.5 从结构到类
- 3.7 成员函数
- 3.8 屏蔽类的内部实现

# 3.1 结构的概念与操作

# 3.1 结构的概念与操作

- 单一数值类型的缺陷

# 3.1 结构的概念与操作

- 单一数值类型的缺陷
  - 客体属性描述往往非单值，批量处理更显复杂。

# 3.1 结构的概念与操作

- 单一数值类型的缺陷
  - 客体属性描述往往非单值，批量处理更显复杂。
  - 如：日期、排序等等。

# 3.1 结构的概念与操作

- 单一数值类型的缺陷
  - 客体属性描述往往非单值，批量处理更显复杂。
  - 如：日期、排序等等。
- 解决方法：结构体

# 3.1 结构的概念与操作

- 单一数值类型的缺陷
  - 客体属性描述往往非单值，批量处理更显复杂。
  - 如：日期、排序等等。
- 解决方法：结构体

```
1 struct Date
2 {
3     int month;
4     int day;
5     int year;
6 }; // 注意尾部分号
```

# 3.1 结构的概念与操作

- 单一数值类型的缺陷
  - 客体属性描述往往非单值，批量处理更显复杂。
  - 如：日期、排序等等。
- 解决方法：结构体

```
1 struct Date
2 {
3     int month;
4     int day;
5     int year;
6 }; // 注意尾部分号
```

# 3.1 结构的概念与操作

- 单一数值类型的缺陷
  - 客体属性描述往往非单值，批量处理更显复杂。
  - 如：日期、排序等等。
- 解决方法：结构体

```
1 struct Date
2 {
3     int month;
4     int day;
5     int year;
6 }; // 注意尾部分号
```

# 3.1 结构的概念与操作

- 单一数值类型的缺陷
  - 客体属性描述往往非单值，批量处理更显复杂。
  - 如：日期、排序等等。
- 解决方法：结构体

```
1 struct Date
2 {
3     int month;
4     int day;
5     int year;
6 }; // 注意尾部分号
```

# 3.1 结构的概念与操作

先声明结构体类型，再定义该类型的变量。

```
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 结构的概念与操作

先声明结构体类型，再定义该类型的变量。

```
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 结构的概念与操作

先声明结构体类型，再定义该类型的变量。

```
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 结构的概念与操作

先声明结构体类型，再定义该类型的变量。

```
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 结构的概念与操作

先声明结构体类型，再定义该类型的变量。

```
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 结构的概念与操作

# 3.1 结构的概念与操作

- 结构体类型与结构体变量是不同的概念

# 3.1 结构的概念与操作

- 结构体类型与结构体变量是不同的概念
  - 什么是类型？什么是变量？

# 3.1 结构的概念与操作

- 结构体类型与结构体变量是不同的概念
  - 什么是类型？什么是变量？
  - 可以简单地理解为：类型是集合，变量是集合中的元素

# 3.1 结构的概念与操作

- 结构体类型与结构体变量是不同的概念
  - 什么是类型？什么是变量？
  - 可以简单地理解为：类型是集合，变量是集合中的元素
- 结构体类型中的成员名可以和变量名相同，但是意义不同

# 3.1 结构的概念与操作

- 结构体类型与结构体变量是不同的概念
  - 什么是类型？什么是变量？
  - 可以简单地理解为：类型是集合，变量是集合中的元素
- 结构体类型中的成员名可以和变量名相同，但是意义不同
- 成员变量可以单独使用，其作用和地位与普通变量相当

# 3.1 结构的概念与操作

```
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 结构的概念与操作

```
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 结构的概念与操作

```
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 结构的概念与操作

```
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 结构的概念与操作

另一种方法（不推荐）

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

# 3.1 结构的概念与操作

另一种方法（不推荐）

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

## 3.2 结构与指针

结构体只是一种组织数据的方式：实际数据保存在成员变量中。但是可以有指向结构体的指针

# 3.2 结构与指针

结构体只是一种组织数据的方式：实际数据保存在成员变量中。但是可以有指向结构体的指针

```
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 结构与指针

结构体只是一种组织数据的方式：实际数据保存在成员变量中。但是可以有指向结构体的指针

```
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 结构与指针

结构体只是一种组织数据的方式：实际数据保存在成员变量中。但是可以有指向结构体的指针

```
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 结构与指针

结构体只是一种组织数据的方式：实际数据保存在成员变量中。但是可以有指向结构体的指针

```
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 结构与指针

结构体只是一种组织数据的方式：实际数据保存在成员变量中。但是可以有指向结构体的指针

```
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 结构与指针

结构体只是一种组织数据的方式：实际数据保存在成员变量中。但是可以有指向结构体的指针

```
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 结构与指针

结构体只是一种组织数据的方式：实际数据保存在成员变量中。但是可以有指向结构体的指针

```
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 结构与指针

如果需要结构体当作函数参数传递，一般采用指针或者引用。否则容易出现性能问题。

# 3.3 结构与数组

与基本数据类型相同的用法。

```
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 结构与数组

与基本数据类型相同的用法。

```
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 结构与数组

与基本数据类型相同的用法。

```
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 }
```

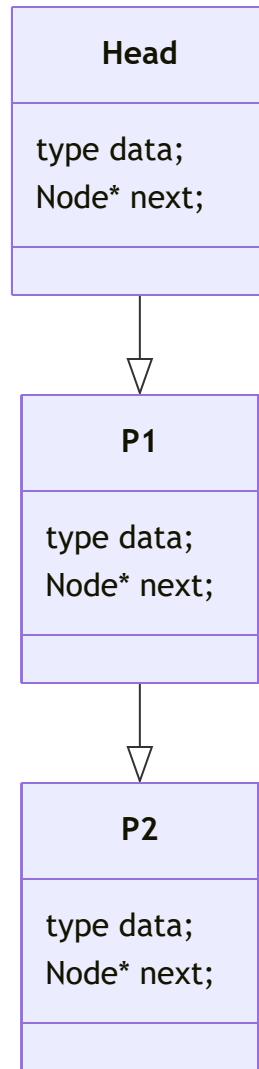
# 3.3 结构与数组

与基本数据类型相同的用法。

```
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 链表

## 什么是链表？



# **3.4 链表**

## **什么是链表？**

## 3.4 链表

### 什么是链表？

- 链表由若干个结点组成

## 3.4 链表

### 什么是链表？

- 链表由若干个结点组成
- 每个节点包含两个成员：

## 3.4 链表

### 什么是链表？

- 链表由若干个结点组成
- 每个节点包含两个成员：
  - 数据（可以是其他结构）

## 3.4 链表

### 什么是链表？

- 链表由若干个结点组成
- 每个节点包含两个成员：
  - 数据（可以是其他结构）
  - 指向下一个节点的指针

## 3.4 链表

### 什么是链表？

- 链表由若干个结点组成
- 每个节点包含两个成员：
  - 数据（可以是其他结构）
  - 指向下一个节点的指针
- 空指针代表空链表

## 3.4 链表

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

# 3.4 链表

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

# 3.4 链表

创建链表：

```
Node* makeList() {  
    return nullptr; // 不要用 NULL  
}
```

## 3.4 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

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 链表

## 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 链表

## 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 链表

## 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 链表

## 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 链表

## 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 链表

## 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 链表

remove, length, find ...

# 3.4 链表

remove, length, find ...

- 如果需要链表， 推荐使用std::list (双向链表)

# 3.4 链表

remove, length, find ...

- 如果需要链表，推荐使用`std::list` (双向链表)
- 如果需要手工实现，推荐使用  
`std::shared_ptr` (智能指针)

# 3.4 链表

remove, length, find ...

- 如果需要链表，推荐使用 `std::list` (双向链表)
- 如果需要手工实现，推荐使用  
`std::shared_ptr` (智能指针)
- 扩展阅读：[Cons List](#)，参考实现：`demo/consList`

# 3.4 链表

remove, length, find ...

- 如果需要链表，推荐使用 `std::list` (双向链表)
- 如果需要手工实现，推荐使用  
`std::shared_ptr` (智能指针)
- 扩展阅读：[Cons List](#)，参考实现：`demo/consList`

`example/lec03/list`

# 3.4 链表

remove, length, find ...

- 如果需要链表，推荐使用 `std::list` (双向链表)
- 如果需要手工实现，推荐使用  
`std::shared_ptr` (智能指针)
- 扩展阅读：[Cons List](#)，参考实现：`demo/consList`

`example/lec03/list`

`example/lec03/smartList`

# 3.5 从结构到类

## 3.5 从结构到类

- 某些函数需要操控特定数据，每次需要将数据传入：

## 3.5 从结构到类

- 某些函数需要操控特定数据，每次需要将数据传入：
  - `Node *prepend(Node *list, ...)`

## 3.5 从结构到类

- 某些函数需要操控特定数据，每次需要将数据传入：
  - `Node *prepend(Node *list, ...)`
  - `Node *insert(Node *list, ...)`

## 3.5 从结构到类

- 某些函数需要操控特定数据，每次需要将数据传入：
  - `Node *prepend(Node *list, ...)`
  - `Node *insert(Node *list, ...)`
  - `void freeList(Node *list)`

## 3.5 从结构到类

- 某些函数需要操控特定数据，每次需要将数据传入：
  - `Node *prepend(Node *list, ...)`
  - `Node *insert(Node *list, ...)`
  - `void freeList(Node *list)`
- 语言层次，数据与其操作没有关联

# 3.5 从结构到类

## 3.5 从结构到类

- 某些数据不想暴露给调用者（内部实现）

## 3.5 从结构到类

- 某些数据不想暴露给调用者（内部实现）
  - 内部更改不影响外部调用

## 3.5 从结构到类

- 某些数据不想暴露给调用者（内部实现）
  - 内部更改不影响外部调用
  - 外部出错更加容易排查错误

## **3.5 从结构到类**

**引入类的概念：**

# 3.5 从结构到类

引入类的概念：

- 数据与操作绑定

# 3.5 从结构到类

引入类的概念：

- 数据与操作绑定
- 数据的访问权限控制

# 3.5 从结构到类

```
class Point {  
    // 数据，默认为 private  
    int x;  
    int y;  
public:  
    // 在类内部定义函数(方法)  
    void print() {  
        cout << '(' << x << ", " << ')';  
    }  
};
```

# 3.5 从结构到类

## 3.5 从结构到类

- 类是一种数据类型，对象是对应类的实例

## 3.5 从结构到类

- 类是一种数据类型，对象是对应类的实例
  - 对象是对应数据类型的变量

## 3.5 从结构到类

- 类是一种数据类型，对象是对应类的实例
  - 对象是对应数据类型的变量
- 类是一张蓝图

## 3.5 从结构到类

- 类是一种数据类型，对象是对应类的实例
  - 对象是对应数据类型的变量
- 类是一张蓝图
  - 对象是根据蓝图制作出来的具体实体

# 3.5 从结构到类

## 3.5 从结构到类

- 出于兼容性考虑，c++ 中，`struct` 也可以加入方法。

## 3.5 从结构到类

- 出于兼容性考虑，c++ 中， struct 也可以加入方法。
- struct 中的成员变量，默认是public 权限

# 3.6 成员函数

## 3.6 成员函数

- 一个类中，包含有成员变量与成员函数

## 3.6 成员函数

- 一个类中，包含有成员变量与成员函数
- 成员变量与成员函数，统称为类的成员

## 3.6 成员函数

- 一个类中，包含有成员变量与成员函数
- 成员变量与成员函数，统称为类的成员
- 成员变量有时被称为属性，成员函数有时被称为方法

## 3.6 成员函数

- 一个类中，包含有成员变量与成员函数
- 成员变量与成员函数，统称为类的成员
- 成员变量有时被称为属性，成员函数有时被称为方法
- 它们都不是必需的

# 3.6 成员函数

## 3.6 成员函数

- 在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：

## 3.6 成员函数

- 在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：
  - `private`: 私有成员，只能在成员函数内访问

## 3.6 成员函数

- 在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：
  - **private**: 私有成员，只能在成员函数内访问
  - **public**: 公有成员，可以在任何地方访问

## 3.6 成员函数

- 在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：
  - **private**: 私有成员，只能在成员函数内访问
  - **public**: 公有成员，可以在任何地方访问
  - **protected**: 保护成员，以后再说

## 3.6 成员函数

- 在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：
  - **private**: 私有成员，只能在成员函数内访问
  - **public**: 公有成员，可以在任何地方访问
  - **protected**: 保护成员，以后再说
- 以上三种关键字出现的次数和先后次序都沒有限制。

# 3.6 成员函数

如何定义一个类？

```
class className {  
private:  
    //私有属性和函数  
public:  
    //公有属性和函数  
protected:  
    //保护属性和函数  
}; // 不要忘了分号
```

# 3.6 成员函数

```
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 成员函数

```
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 成员函数

```
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 成员函数

```
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 成员函数

```
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 成员函数

```
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 成员函数

```
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 成员函数

```
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 成员函数

```
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 成员函数

```
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    };
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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    }  
17}
```

# 3.8 屏蔽类的内部实现

```
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    }
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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}
```

# 3.8 屏蔽类的内部实现

```
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;
```

## 3.8 屏蔽类的内部实现

## 3.8 屏蔽类的内部实现

- 内部更改，接口不变，不会影响外部程序

## 3.8 屏蔽类的内部实现

- 内部更改，接口不变，不会影响外部程序
  - 解耦合

## 3.8 屏蔽类的内部实现

- 内部更改，接口不变，不会影响外部程序
  - 解耦合
- 更加方便定位错误

## 3.8 屏蔽类的内部实现

- 内部更改，接口不变，不会影响外部程序
  - 解耦合
- 更加方便定位错误
  - 不用考虑外部在不未知情况下改动了私有属性