

第七章 面向对象

C++ 简明双链教程（李昕著，清华大学出版社）

作者: 李昕

qingline.net/cppbook

目录

01 类和对象

02 动态对象和this指针

03 动态属性和析构函数

04 封装

09 综合练习-构建链表

05 继承

06 多态

07 操作符重载

08 静态属性

01

类和对象

中国石油大学(华东)

qingline.net/cppbook

类和对象

面向对象的程序设计语言

以对象为基础进行程序设计，叫做面向对象编程。C++是面向对象语言，但是因为对C的兼容，并不是纯粹的面向对象语言，但是JAVA和Python都是纯面向对象语言，在纯面向对象语言中，一切都是对象。

1类和对象

表7.1 是NBA2022年东部赛区的一些统计数据。

一行数据是构成一个球队的整体，每列都是这个球队的一个属性，C++中可以通过“类”或“结构体”封装具有不同数据类型多个属性的数据。

表7.1 NBA2022年东部赛区统计数据

排名	球队	胜	负	胜率	得分	失分	分差
1	热火	53	29	64.60%	110.02	105.57	4.45
2	凯尔特人	51	31	62.20%	111.76	104.48	7.28
3	雄鹿	51	31	62.20%	115.49	112.13	3.35
4	76人	51	31	62.20%	109.94	107.33	2.61
5	猛龙	48	34	58.50%	109.39	107.1	2.29
6	公牛	46	36	56.10%	111.61	112	-0.39
7	篮网	44	38	53.70%	112.9	112.12	0.78
8	老鹰	43	39	52.40%	113.94	112.38	1.56
9	骑士	44	38	53.70%	107.79	105.67	2.12
10	黄蜂	43	39	52.40%	115.33	114.89	0.44
11	尼克斯	37	45	45.10%	106.48	106.6	-0.12
12	奇才	35	47	42.70%	108.62	112	-3.38
13	步行者	25	57	30.50%	111.46	114.94	-3.48
14	活塞	23	59	28.00%	104.83	112.55	-7.72
15	魔术	22	60	26.80%	104.23	112.23	-8

1类和对象

结构体是C语言中提供的复合数据类型，因为类包含了结构体中的所有功能，因此只需要掌握类的使用方法即可。以下为构造类的具体代码实现：

样例输入	样例输出
(无)	Heat 0.646341 4.45 Bulls 0.560976 - 0.399

代码7.1 类与对象样例

```
1  #include <iostream>
2  using namespace std;
3  class Team {
4  public:
5      string name;
6      int win, lose;
7      double points, opoints;
8  Team(const string &n, int w, int l, double p, double op) {
9      name = n;
10     win = w;
11     lose = l;
12     points = p;
13     opoints = op;
14 }
15 double rate() {
16     return win / double(win + lose);
17 }
18 double gap() {
19     return points - opoints;
20 }
21 };
22 int main()
23 {
24     Team t1 = Team("Heat", 53, 29, 110.02, 105.57);
25     Team t2 = Team("Bulls", 46, 36, 111.61, 112);
26     cout << t1.name << "\t" << t1.rate() << "\t" << t1.gap() << endl;
27     cout << t2.name << "\t" << t2.rate() << "\t" << t2.gap() << endl;
28     return 0;
29 }
```

第3-21行构造了一个新的类Team，第24-25行用新的类构造了两个变量t1和t2，也称为对象。

第5-7行定义了5个属性，这些属性可以是各种数据类型。

15-20行定义了两个函数，属于一个类的函数称为这个类的成员函数，表示这个类所能执行的行为。第8-14行定义了一个跟类同名的函数，称为构造函数。

知识点

索引	要点	正链	反链
T711	掌握类、对象、构造函数、成员变量、成员函数		T791

qingline.net/cppbook

02

动态对象和this指针

中国石油大学(华东)

qingline.net/cppbook

动态对象和this指针

类就是一种新的数据类型，也可以用创建动态对象。

相同类型构造的多个对象，在执行成员函数时，都是执行同一个成员函数。类的每个成员函数（静态函数除外），包括构造函数，都包含一个隐藏的名为this的形参，这个this参数为指向该对象的指针，用于指明当前正在被处理的对象。

1 动态对象和this指针

当访问指针变量的属性或方法时，可也采用第5行的写法，但是书写时比较繁琐，因此C/C++通常采用第6行的形式进行代替，箭头符号->也产生了指针的含义。

代码7.2 类的动态对象

```
1 int main()
2 {
3     Team* t1 = new Team("Heat", 53, 29, 110.02, 105.57);
4     Team* t2 = new Team("Bulls", 46, 36, 111.61, 112);
5     cout << (*t1).name << '\t' << (*t1).rate() << '\t' << (*t1).gap() << endl;
6     cout << t2->name << '\t' << t2->rate() << '\t' << t2->gap() << endl;
7     delete t1;
8     delete t2;
9     return 0;
10 }
```

1动态对象和this指针

相同类型构造的多个对象，在执行成员函数时，都是执行同一个成员函数。

类的每个成员函数（静态函数除外），包括构造函数，都包含一个隐藏的名为this的形参，这个this参数为指向该对象的指针，用于指明当前正在被处理的对象。

样例输入	样例输出
(无)	0x64fda0 0x64fdc0 Heat t1 0x64fda0 Bulls t2 0x64fdc0 Heat Heat 0x64fda0 Bulls Bulls 0x64fdc0

代码7.2 类的动态对象

```
1  #include <iostream>
2  using namespace std;
3  class Team {
4  public:
5      string name;
6      Team(const string &n) {
7          name = n;
8      }
9      void test(string name) {
10         cout << this->name << '\t' << name << '\t' << this << endl;
11     }
12     void test1(string n) {
13         cout << this->name << '\t' << n << '\t' << this << endl;
14     };
15 };
16 int main()
17 {
18     Team t1 = Team("Heat");
19     Team t2 = Team("Bulls");
20     cout << &t1 << '\t' << &t2 << endl;
21     t1.test("t1");
22     t2.test("t2");
23     t1.test1("t1");
24     t2.test1("t2");
25     return 0;
26 }
```

第20行输出2个对象的地址，说明两个对象存放在两个不同的空间，存放了不同的数据。

第21-24行用两个不同的对象分别调用成员函数test和test1时，输出的this显示了不同的地址，分别对应t1和t2。

第13行，当调用属性或成员函数时，可以添加this指针，也可以省略，二者作用相同。但是如果跟局部变量有命名冲突时，例如第10行的输出结果，省略时表示局部变量，有this指针时表示类的属性或成员函数。

知识点

索引	要点	正链	反链
T721	掌握动态对象和this指针的使用方法	T631	T791

qingline.net/cppbook

03

动态属性和析构函数

中国石油大学（华东）

qingline.net/cppbook

动态属性和析构函数

- 指针数据成员的赋值需要通过动态内存分配。
- 析构函数用于是否动态分配的内存。
- 如果在对象生命周期结束时，如果有需要保存的文件，需要释放的网络链接等资源，也可以在析构函数中完成。
- 析构函数与构造函数类似，没有返回值，与类名相同，前面加一个~运算符。
- 析构函数都是无参的，一个类只能最多有一个析构函数。

3动态属性和析构函数

每个类都提供了一个析构函数，当对象生命周期结束，或被delete删除时，自动释放它的所有成员所占据的内存。以下为析构函数的使用的具体代码实现：

样例输入	样例输出
(无)	101

代码7.3 析构函数的使用

C++

Yuque Light

...

```
1  #include <iostream>
2  using namespace std;
3  class Team {
4  public:
5      int* score=NULL;
6      Team(int n) {
7          score = new int(n);
8      }
9      void test(string name) {
10         cout << *score << endl;
11     }
12     ~Team() {
13         if(score!=NULL)
14             delete score;
15     };
16 };
17 int main()
18 {
19     Team *t1 = new Team(101);
20     t1->test("t1");
21     delete t1;
22     return 0;
23 }
```

第12-15行就是析构函数。属性score在构造函数中被动态分配，当第21行清除对象t1时，自动调用析构函数，属性score所分配的堆内存被释放。

知识点

索引	要点	正链	反链
T731	掌握动态属性和析构函数使用方法	T631	

qingline.net/cppbook

04

封装

中国石油大学(华东)

qingline.net/cppbook

封装

面向对象的三大基本特征就是封装（Encapsulation）、继承（Inheritance）和多态（Polymorphism）。封装的一个非常好的实践准则就是将所有的属性成员定义为私有，这样可以更好地控制数据，当数据访问发生变更时，只需要简单地修改一个地方，就可以满足需求的变更，而不需要大规模的修改代码，从而提高数据的安全性。C++中提供了三种访问控制符进行不同级别的访问控制：

访问控制符	说明
private	私有，说明该成员（数据/函数）仅允许在类的内部进行访问
protected	保护，该成员可以在类的内部访问，也运行在该类的继承类中访问
public	公有，该成员可以随意访问

4 访问控制

属性score设定为private，这样类外就无法直接访问。

第15行执行会报错。但是可以在类内进行访问，例如第7-8行函数体中所示。

代码7.4 访问控制

```
1  #include <iostream>
2  using namespace std;
3  class Team {
4  private:
5      int score;
6  public:
7      int getScore() { return score; }
8      void setScore(int score) { this->score = score; }
9  };
10 int main()
11 {
12     Team t1 = Team();
13     t1.setScore(101);
14     cout << t1.getScore() << endl;
15     //cout << t1.score << endl;
16     return 0;
17 }
```

知识点

索引	要点	正链	反链
T741	掌握对象的封装和属性的访问控制		T791

qingline.net/cppbook

05

继承

中国石油大学(华东)

qingline.net/cppbook

封装

在 C++ 中，继承是一个对象自动获取其父对象的所有属性和行为的过程。这样可以重用、扩展或修改其他类中定义的属性和行为。在 C++ 中，继承另一个类成员类称为派生类，其成员被继承的类称为基类。派生类是基类的专用类。C++ 中支持五种形式的继承：1) 单继承；2) 多重继承；3) 分层继承；4) 多级继承；5) 混合继承。

5 类的继承

以下是类的继承的代码实现：

派生类构造的对象myCar可以访问父类的属性brand和成员函数honk。

父类的public和protected成员都会被继承，注意private成员是不能被继承的，protected成员可以被继承，但是不能被外部变量访问。

代码7.5 类的继承

```
1  #include <iostream>
2  using namespace std;
3  // 基类
4  class Vehicle {
5      public:
6          string brand = "Ford";
7          void honk() {
8              cout << "Tuut, tuut! \n" ;
9          }
10 };
11 // 派生类
12 class Car: public Vehicle {
13     public:
14         string model = "Mustang";
15 };
16
17 int main() {
18     Car myCar;
19     myCar.honk();
20     cout << myCar.brand + " " + myCar.model;
21     return 0;
22 }
```

父类中可以定义多个子类（派生类）中的公有属性或成员函数，相当于定义了一个标准的接口。子类中需要书写的代码也会大量的减少。

样例输入	样例输出
(无)	Tuut, tuut! Ford Mustangg

知识点

索引	要点	正链	反链
T751	掌握对象的继承的使用方法		T771

qingline.net/cppbook

06

多态

中国石油大学(华东)

qingline.net/cppbook

继承

多态意味着“多种形式”。继承从另一个类继承属性和方法。多态性使用这些方法来执行不同的任务，能够以不同的方式执行单个操作。多态按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

6.1 virtual关键字的使用

派生类中如果重新定义了父类的成员函数，称为函数重载。

通过函数重载，子类可以修改父类的同名函数。

例子中的show函数就是一个重载函数

样例输入	样例输出
(无)	print derived class show base class

代码7.6 virtual关键字的使用

```
1  #include<iostream>
2  using namespace std;
3
4  class base {
5  public:
6      virtual void print() {
7          cout << "print base class\n";
8      }
9      void show() {
10         cout << "show base class\n";
11     }
12 };
13
14 class derived : public base {
15 public:
16     void print() {
17         cout << "print derived class\n";
18     }
19     void show() {
20         cout << "show derived class\n";
```

6.1 virtual关键字的使用

- 运行时多态性只能通过基类类型的指针（或引用）来实现。
- 后期绑定根据指针的内容进行，早期绑定根据指针的类型进行。

▼ 代码7.6 virtual关键字的使用

```
1  #include<iostream>
2  using namespace std;
3
4  ▼ class base {
5      public:
6          ▼ virtual void print() {
7              cout << "print base class\n";
8          }
9          ▼ void show() {
10             cout << "show base class\n";
11         }
12     };
13
14  ▼ class derived : public base {
15      public:
16          ▼ void print() {
17              cout << "print derived class\n";
18          }
19          ▼ void show() {
20              cout << "show derived class\n";
```

6.2 函数重载

考虑一个名为Container的基类，它有一个名为pop()的方法。

容器的派生类可以是队列（Queue）或堆栈（Stack）。

对于pop()这个行为，它们有各自不同的表现，实现了函数重载。

第6行在基类中定义了一个纯虚函数pop()。

第35-38行显示不同派生类的对象会执行不同的行为。

代码7.7 函数重载

```
1  #include <iostream>
2  using namespace std;
3  // 基类
4  class Container {
5  public:
6      virtual void pop()=0;    //纯虚函数，定义一个接口，没有具体实现
7      // virtual void pop(){    //虚函数，可以有自己的定义实现
8      //     cout << "Container Pop!\n" ;
9      // }
10 };
11 // 派生类
12 class Queue : public Container {
13 public:
14     void pop() {
15         cout << "Queue: Pop the first element!\n" ;
16     }
17 };
18 // 派生类
19 class Stack : public Container {
20 public:
21     void pop() {
22         cout << "Stack: Pop the last element!\n" ;
23     }
24 };
```

6.2 函数重载

样例输出

Queue: Pop the first element!
Stack: Pop the last element!
Queue: Pop the first element!
Stack: Pop the last element!
Queue: Pop the first element!
Stack: Pop the last element!

最后看第26-28行定义了一个函数 polymorphism(), 因为基类指针既可以指向基类的对象, 也可以指向派生类的对象, 所以Container以及其所有的子类都可以用指针的形式传入.

代码7.7 函数重载

```
1  #include <iostream>
2  using namespace std;
3  // 基类
4  class Container {
5  public:
6      virtual void pop()=0;    //纯虚函数, 定义一个接口, 没有具体实现
7      // virtual void pop(){    //虚函数, 可以有自己的定义实现
8      //     cout << "Container Pop!\n" ;
9      // }
10 };
11 // 派生类
12 class Queue : public Container {
13 public:
14     void pop() {
15         cout << "Queue: Pop the first element!\n" ;
16     }
17 };
18 // 派生类
19 class Stack : public Container {
20 public:
21     void pop() {
22         cout << "Stack: Pop the last element!\n" ;
23     }
24 };
```

知识点

索引	要点	正链	反链
T761	掌握virtual关键字的使用		
T762	掌握函数重载的实现		

07

操作符重载

中国石油大学(华东)

qingline.net/cppbook

7.1 操作符重载

用户可以重新定义或重载 C++ 中可用的大多数内置运算符，可以给运算符赋予新的自定义行为，并且书写和理解上都更加清晰。重载运算符是具有特殊名称的函数：关键字 “operator”后跟正在定义的运算符的符号。与任何其他函数一样，重载运算符具有返回类型和参数列表。操作符重载和普通成员函数重载的含义是完全相同的，只是在定义和使用上略有区别。

7 操作符重载

以下是操作符重载的代码实现

第12-14行重载了操作符+。

第24行对Box对象执行+操作时，就会调用这个函数。

代码7.8 操作符重载

```
1  #include <iostream>
2  using namespace std;
3  class Box {
4  public:
5      Box(double l, double b, double h) {
6          length = l; breadth = b; height = h;
7      }
8      double getVolume(void) {
9          return length * breadth * height;
10     }
11     // 重载操作符+
12     Box operator+(const Box &b) {
13         return Box(this->length+b.length, this->breadth+b.breadth, this->height+b.height);
14     }
15 private:
16     double length, breadth, height; // 长宽高
17 };
18 int main()
19 {
20     Box box1(6., 7., 5.);
21     Box box2(12., 13., 10.);
22     cout << "Volume of Box1 : " << box1.getVolume() << endl;
23     cout << "Volume of Box2 : " << box2.getVolume() << endl;
24     Box box3 = box1 + box2;
25     cout << "Volume of Box3 : " << box3.getVolume() << endl;
26     return 0;
27 }
```

知识点

索引	要点	正链	反链
T771	掌握操作符重载的使用	T751	T791

08

静态属性

中国石油大学(华东)

qingline.net/cppbook

8 静态属性

在变量定义前加上关键字static，就转换为静态变量。

静态变量和全局变量都存储在全局静态数据区，其生命周期都从定义开始持续到程序运行结束。

静态变量的初始化只会被执行一次。

样例输入	样例输出
(无)	1 11 12 17 16 16 11 12

代码7.9 静态变量的使用

```
1  #include <iostream>
2  using namespace std;
3  // 基类
4  class Container {
5  public:
6      static int count;
7      Container() { ++count; }
8      ~Container() { --count; }
9  };
10 // 派生类
11 class Queue : public Container {
12 public:
13     void pop() {
14         cout << "Pop the first element!\n" ;
15     }
16 };
17 int Container::count = 0;
18 int main() {
19     Container c;
20     cout << Container::count << endl;
21     cout << c.count << endl;
22     Container arr[10];
23     cout << Container::count << endl;
```

8 静态属性

第13行定义了一个静态成员属性，在第17行进行初始化。

因为count是成员属性，调用时必须添加类修饰符，例如Container::

第19-31行显示，各种变量构建时都会调用构造函数，释放时都会调用析构函数。

第32-33行的运行结果发现对派生类进行调用时，父类的构造函数也会被调用。

代码7.9 静态变量的使用

```
1  #include <iostream>
2  using namespace std;
3  // 基类
4  class Container {
5  public:
6      static int count;
7      Container() { ++count; }
8      ~Container() { --count; }
9  };
10 // 派生类
11 class Queue : public Container {
12 public:
13     void pop() {
14         cout << "Pop the first element!\n" ;
15     }
16 };
17 int Container::count = 0;
18 int main() {
19     Container c;
20     cout << Container::count << endl;
21     cout << c.count << endl;
22     Container arr[10];
23     cout << Container::count << endl;
```

知识点

索引	要点	正链	反链
T781	掌握静态属性的定义和使用	T335	

qingline.net/cppbook

09

综合练习-构建链表

中国石油大学(华东)

qingline.net/cppbook

9 综合练习-构建链表

本节通过构建链表，练习类和对象的构造，动态对象和this指针、封装和操作符重载。更重要的是理解链表这种特殊的数据结构。链表可以形成一个序列，但是序列中的每个节点的存储空间都是不连续的。这样可以更加方便的插入和删除，但是不能进行随机访问。

9.1 构建节点类

首先，构建一个节点类，每个节点除了保存数据外，更重要的是形成一个指针，用与指向下一个节点，这样才能形成序列。

代码7.10 构建节点类

```
1 class Node
2 {
3 public:
4     int val;
5     Node *next;
6     Node(int val, Node *next = nullptr)
7     {
8         this->val = val;
9         this->next = next;
10    }
11 };
```

//存储数据，此处以int为例

//用于存储下一个节点

//成员变量和形参同名，用this指针进行区分

9.2 构建链表类

然后构建一个链表类，其中的head始终指向头节点。

begin和end操作能够得到迭代器。然后分别实现了尾部添加、插入和删除操作。

代码7.12 构建链表类

```
1 class List{
2     private:
3         Node* head;
4     public:
5         List(){
6             head = nullptr;
7         }
8         Iterator begin(){
9             return Iterator(this->head);           //由头结点封装的迭代器
10        }
11        Iterator end(){
12            return Iterator(nullptr);               //尾结点是空指针
13        }
14        void push_back(int val){
15            Node* node = new Node(val);             //构造一个新节点
16            if(!head)                               //如果链表为空
17                head = node;                         //头结点就是新节点，构建了只有一个节点的链表
18            else{
19                Node* ptr = head;
20                while(ptr->next)                     //如果没有到尾部
21                    ptr = ptr->next;                 //转入下一个节点
22                ptr->next = node;                    //新节点作为尾节点
23            }
24        }
25        bool insert(const Iterator& it, int val){
26            Node* tmp = head;
27            while(Iterator(tmp)!=end() && Iterator(tmp->next)!=it) //以迭代器形式比较
28                tmp = tmp->next;                       //转入下一个节点
29            if(Iterator(tmp)==end()) return false; //没有找到迭代器指向的位置
30            Node* node = new Node(val);               //构造一个新节点
31            node->next = tmp->next;                   //新节点和下一个节点链接
32            tmp->next = node;                         //新节点和上一个节点链接，注意和上一行顺序不能变
33            return true;
```

9.3演示链表类的使用

最后，演示了使用方法。只能通过迭代器访问节点，用户不能直接对节点进行操作，从而实现了链表的安全访问。

因为List支持了begin、end和迭代器，因此可以执行for(auto 变量:容器)操作，进一步说明迭代器可以实现算法和具体数据的分离。

样例输入	样例输出
5	1 2 3 4 5 1 10 2 3 4 5 1 10 3 4 5

代码7.13 演示链表类的使用

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      int m;
6      cin>>m;
7      List ls;
8      for(int i=0;i<m;i++)
9          ls.push_back(i+1);          //构建链表的数据
10     for(auto it = ls.begin();it!=ls.end();it++)
11         cout<<*it<<' ';
12     cout<<endl;
13     auto it = ls.begin();          //指向链表的开始
14     it++;                          //指向下一个节点
15     ls.insert(it,10);              //插入新节点
16     for(auto e:ls)                 //遍历
17         cout<<e<<' ';
18     cout<<endl;
19     ls.erase(it);                  //删除指定的节点，这时依旧指向2对应的节点
20     for(auto e:ls)                 //遍历
21         cout<<e<<' ';
22     cout<<endl;
23     return 0;
24 }
```

知识点

索引	要点	正链	反链
T791	掌握空间不连续序列的使用，了解迭代器的作用	T711,T721, T741,T771	T822, T863

THANKS

中国石油大学(华东)

李昕

qingline.net/cppbook