【7】一棵 n 个结点的完全二叉树以顺序存储的方式存储在一个树组中。试设计三个算法分别实现前序、中序和后序遍历。

【解】为了测试这三个算法，我们设计了一个基于顺序存储的二叉树类。完全二叉树的顺序存储需要一个动态数组，因而数据成员有两个：指向结点类型的指针和结点的个数。三种遍历仍然可以采用递归的方式实现，因而有三个公有的遍历函数和三个私有的遍历函数。为了创建一棵二叉树，在类中增加了一个构造函数。因为二叉树的顺序存储是用动态数组实现的，所以还需要添加一个析构函数。

构造函数的参数是一个数组，它表示了某棵树的顺序存储。构造函数将该数组的内容复制到数据成员 array。析构函数删除保存树的动态数组。

三个公有的遍历函数对根结点调用相应的私有的遍历函数。三个私有的遍历函数的实现完全按照三种遍历的递归定义。

基于上述讨论的顺序实现的二叉树类的定义见代码清单 5-17。所有的成员函数都作为内联函数直接定义在类中。

代码清单 5-17 顺序实现的二叉树类

```
1.  template <class T>
2.  class BinaryTree {
3.        T *array;                          // 保存结点值的数组
4.        int size;                              // 树的规模
5.
6.   public:
7.        BinaryTree( T a[], int n)                  // 构造函数
8.        {  array = new T[n + 1];
9.           size = n;
10.             for (int i = 0; i < n; ++i)     // 根结点存储在 1 号单元中
11.                 array[i+1] = a[i];
12.       }
13.
14.       void pre_order()       { pre_order(1); }
15.       void in_order()    { in_order(1); }
16.       void post_order()      { post_order(1); }
17.
18.    ~BinaryTree()     { delete [] array; }
19.
20.  private:
21.       void pre_order(int root)
22.       {  if (root > size ) return;
23.            cout << array[root];             // 访问根结点
24.            pre_order(root * 2);          // 前序遍历左子树
25.            pre_order(root * 2 + 1);      // 前序遍历右子树
```

```
26.         }
27.
28.       void in_order(int root)
29.          {  if (root > size ) return;
30.                in_order(root * 2);            // 中序遍历左子树
31.                cout << array[root];              // 访问根结点
32.                in_order(root * 2 + 1);         // 中序遍历右子树
33.         }
34.
35.       void post_order(int root)
36.          {  if (root > size ) return;
37.                post_order(root * 2);          // 后序遍历左子树
38.                post_order(root * 2 + 1);       // 后序遍历右子树
39.                cout << array[root];              // 访问根结点
40.         }
41.  };
```