Программирование на C++. Часть 2
Лекция 7

ПМИ Семестр 2

Демяненко Я.М.

2026

Классы для рекурсивных типов данных

Рекурсивное определение данных возникает, когда структура данных содержит ссылку на объект того же типа.

Примеры рекурсивных структур:

- **Односвязный список** — содержит одну ссылку на следующий элемент. Для таких структур обычно эффективнее использовать **итеративные алгоритмы**.

- Бинарное **дерево** — содержит две ссылки (на левое и правое поддеревья). Для таких структур **рекурсивные алгоритмы** естественнее и проще в реализации.

В данной лекции рассматривается реализация бинарного дерева поиска на C++.

Рекурсивное определение бинарного дерева

- Первый способ определения класса бинарного дерева состоит в том, чтобы определить в нём **указатели на два поддерева** и **открытые рекурсивные функции** для работы с деревом.

- Второй способ. Рекурсивное определение бинарного дерева можно реализовать как класс, содержащий внутри себя **указатель на узел дерева** (корень) и **рекурсивное описание узла дерева**.
  При этом **открытые функции-члены** класса **не будут рекурсивными**.

Пример. Реализовать класс бинарного дерева поиска, определив в нём

указатели на два поддерева и
открытые рекурсивные функции для работы с деревом:

конструктор,
конструктор копии,
деструктор,
добавление элемента в дерево и
вывод всех элементов дерева в поток.

```
class TreeR{
private:
  int data;
  TreeR *lt;
  TreeR *rt;

public:
  TreeR (int val=0, TreeR *l = nullptr, TreeR *r = nullptr):
        data(val), lt(l), rt(r) {}

  TreeR(const TreeR * t){
     if (t->lt) lt = new TreeR(t->lt);
     data = t->data;
     if (t->rt) rt = new TreeR(t->rt);
   }

 ~TreeR(){
        if (lt != nullptr)  delete lt;
        if (rt != nullptr) delete rt;
 }
```

```
void add(int a){
   if (data > a){
     if (lt) lt->add(a);
     else lt = new TreeR(a);
   }
   else{
     if (rt) rt->add(a);
     else rt = new TreeR(a);
   }
 }
```

```
friend ostream& operator<<(ostream& os, const TreeR *t){
    if (t->lt) os << t->lt;
    os <<  t->data << "  ";
    if (t->rt) os<< t->rt;
    return os;
  }

  void printRKL(){
    if (rt) rt->printRKL();
    cout << "  " << data << " ";
    if (lt) lt->printRKL();
  }
};
```

```
if (rt) rt->printRKL();
```
эквивалентно
```
if (this->rt) this->rt->printRKL();
```

Особенности первого подхода

TreeR *t= new TreeR(); // это непустое дерево

```
t->add(5);
t->add(-7);
t->add(3);
t->add(-4);
cout <<"T "<< t << endl;
t->printRKL();
TreeR *t1 = new TreeR(t);
cout << "T1 "<<t1 << endl;
```

Для класса TreeR пустота дерева определяется не на уровне класса, а на уровне указателя на объект класса.

TreeR* t0= nullptr; // это пустое дерево

Особенности первого подхода: проверка на пустоту

TreeR* t0= nullptr; // это пустое дерево

Поэтому **нельзя** реализовать функцию-**член класса** для проверки **на пустоту**.

Для безопасной работы перед вызовом функций-членов класса следует выполнять проверку на пустоту следующим образом:

```
if (t0!=nullptr) t0->add(3);
if (t0!=nullptr) t0->RKL();
if (t0!=nullptr) cout <<"T "<< t0 << endl;
```

Особенности первого подхода: удаление

```
~TreeR(){
    if (lt != nullptr)
        delete lt;
    if (rt != nullptr)
        delete rt;
}
```

При использовании операции удаления отдельных узлов из дерева возникает **проблема при удалении последней вершины**, которая **не может быть** решена **в самой операции**.

Пример. Реализовать класс бинарного дерева поиска, содержащий внутри себя

указатель на корень дерева и
рекурсивное описание узла дерева.

Определить **открытые функции-члены класса**:

конструктор,
конструктор копии,
деструктор,
добавление элемента в дерево и
вывод всех элементов в поток.

При их реализации **использовать рекурсивные функции в закрытой** части описания класса.

```
class Tree{
private:
  struct TNode;
  typedef TNode* node_ptr;

  struct TNode{
    int data;
    node_ptr lt, rt;
    TNode (int val, node_ptr l=nullptr, node_ptr r=nullptr):
          data(val), lt(l), rt(r){}
  };
   node_ptr root;

     void delTree(node_ptr t){
       if (t!=nullptr){
         delTree(t->lt);
         delTree(t->rt);
         delete t;
       }
     }
```

Продолжение: закрытые функции

```
void add(node_ptr& t, int a){
   if (t == nullptr)
       t = new TNode(a);
   else if (t->data > a)
       add(t->lt, a);
   else
       add(t->rt, a);
 }

 void printLKR(node_ptr t, ostream& os) const{
   if (t){
     printLKR(t->lt, os);
     os << t->data << " ";
     printLKR(t->rt, os);
   }
 }
```

```
void copy(node_ptr t, node_ptr &newT) const {
   if (t != nullptr){
     newT = new TNode(t->data, 0, 0);
     copy(t->lt, newT->lt);
     copy(t->rt, newT->rt);
   }
   else newT = nullptr;
 }
```

Продолжение: открытые функции

```
public:
  Tree(): root(nullptr) {}
  Tree(const Tree& t){
    copy( t.root, root);
  }

  ~Tree(){
    delTree (root);
  }
  void addNode(int a){
    add(root, a);
  }

  friend ostream& operator<<(ostream& os, const Tree &t){
    t.printLKR(t.root,os);
    return os;
  }
};
```

Особенности второго подхода

В примере типы TNode (узел) и node_ptr (указатель на узел) определены внутри класса, что подчёркивает **инкапсуляцию внутренней** организации класса Tree.

Именно в определении типа **TNode** присутствует **рекурсия**.

Поэтому **рекурсивные функции** оперируют с указателями на узел дерева и должны быть объявлены как **private**.

Для доступа к ним используются **открытые функции** класса, которые **вызывают рекурсивные** функции, передавая в качестве параметра указатель на корень дерева.

Особенности второго подхода: пустота и деструктор

Такое описание класса допускает существование пустого дерева.

Конструктор без параметров создаёт пустое дерево.

Если предусмотреть операцию удаления узла дерева, то можно получить в процессе её выполнения пустое дерево.

Для безопасной работы с таким деревом рекомендуется иметь операцию проверки дерева на пустоту.

```
Tree t;
t.addNode(5);
t.addNode(7);
t.addNode(3);
t.addNode(4);
cout <<"T "<< t << endl;
Tree t1(t);
cout << "T1 "<<t1 << endl;
```

Рекомендации

Реализация класса бинарного дерева вторым способом лишена недостатков, перечисленных для первого способа.

Именно
второй способ реализации рекомендуется использовать при создании классов для рекурсивных структур.

Callback-функции (функции обратного вызова)

Callback — это функция, которая передается как параметр в другую функцию, чтобы быть вызванной в определенный момент.
Иными словами, callback — это действие передаваемое параметром

```
// Пример: применение операции ко всем элементам списка
template <typename T>
// action — переменная типа «указатель на функцию»
void for_each(Node<T>* p, void (*action)(T&)) {
   while (p != nullptr) {
      action(p->data);   // вызываем callback для каждого элемента
      p = p->next;
   }
}
```

```
// Использование
Node<int>* head = ...;  // указатель на начало списка

for_each(head, print);      // выводим все элементы
for_each(head, increment);  // увеличиваем все элементы
for_each(head, print);      // выводим обновленные значения
```

```
// Функции, которые можно передать как callback
void print(int& x) {
   cout << x << ' ';
}

void increment(int& x) {
   x++;
}
```

Преимущества callback-функций

- Переиспользование кода (одна функция for_each работает с разными операциями)
- Гибкость (можно передать любую подходящую функцию)
- Основа функционального программирования

Как объявлять?

```
// Указатель на функцию с таким прототипом
typedef double (*BinOp) (double, double);

BinOp bop = &add;
(*bop)(3, 5);


// Или как и описание переменной
double (*op)(double, double)

op = mult;
op(3, 5);
```

В C++11 и новее вместо указателей на функции можно использовать лямбда-выражения:

```
for_each(head, [](int& x) { x *= 2; });  // умножить все на 2
```

Пример использования callback с деревом

```
template <typename T>
void for_each_node(node<T>* p, void (*action)(T&)) {
    if (p == nullptr) return;
    for_each_node(p->left, action);
    action(p->data);
    for_each_node(p->right, action);
}

// Применение для дерева целых чисел
void printInt(int& x) {
    cout << x << " ";
}

void doubleInt(int& x) {
    x *= 2;
}
```

```
int main() {
    // ... создание дерева ...
    for_each_node(root, printInt);   // вывод: 3 5 7 10 12 15 17
    for_each_node(root, doubleInt);  // удвоение всех значений
    for_each_node(root, printInt);   // вывод: 6 10 14 20 24 30 34

    return 0;
}
```