Практическое занятие №3

Густов Владимир Владимирович
gutstuf@gmail.com

Цитата дня: Любо, братцы, любо. Любо, братцы, жить.
В танковой бригаде не приходится тужить… (с) Егор Летов

Дерево

- содержит указатель на потомков;
- удаление/добавление элемента либо упорядочено (для бинарных), либо производится напрямую из локального родителя.

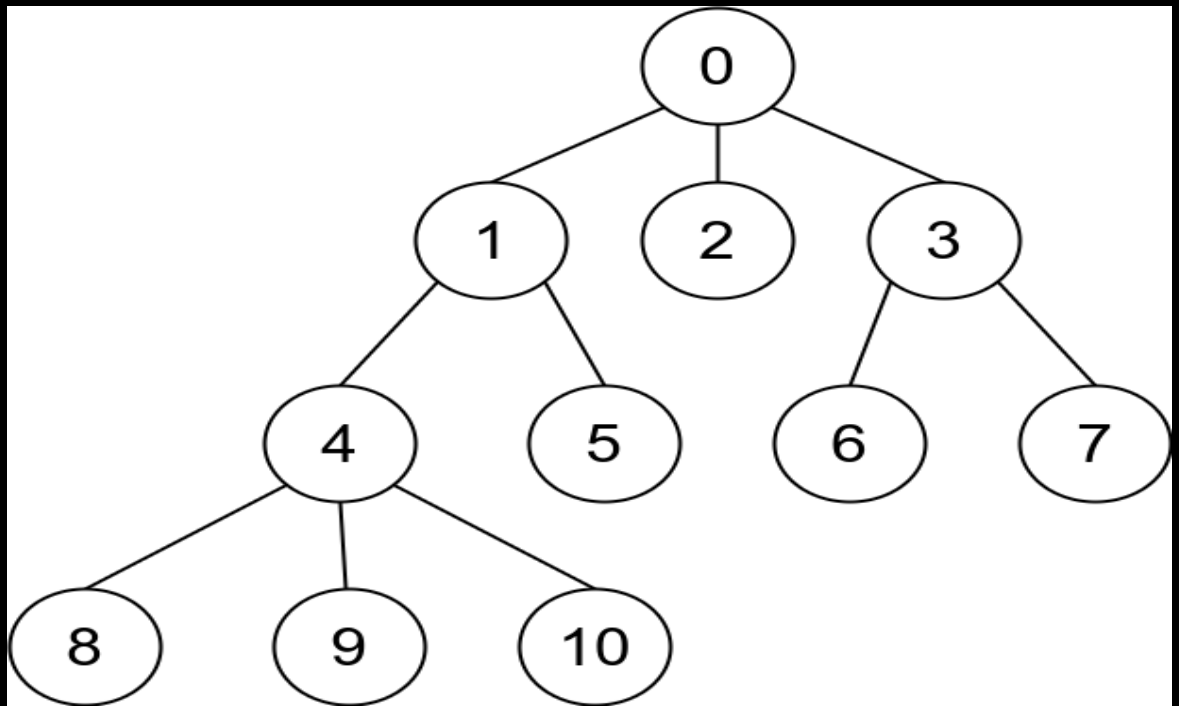
Бинарное (двоичное) дерево строится по правилам:

- в левое поддерево добавляются элементы меньше корня;
- в правое поддерево добавляются элементы больше или равные корню;

Способы обходов дерева:

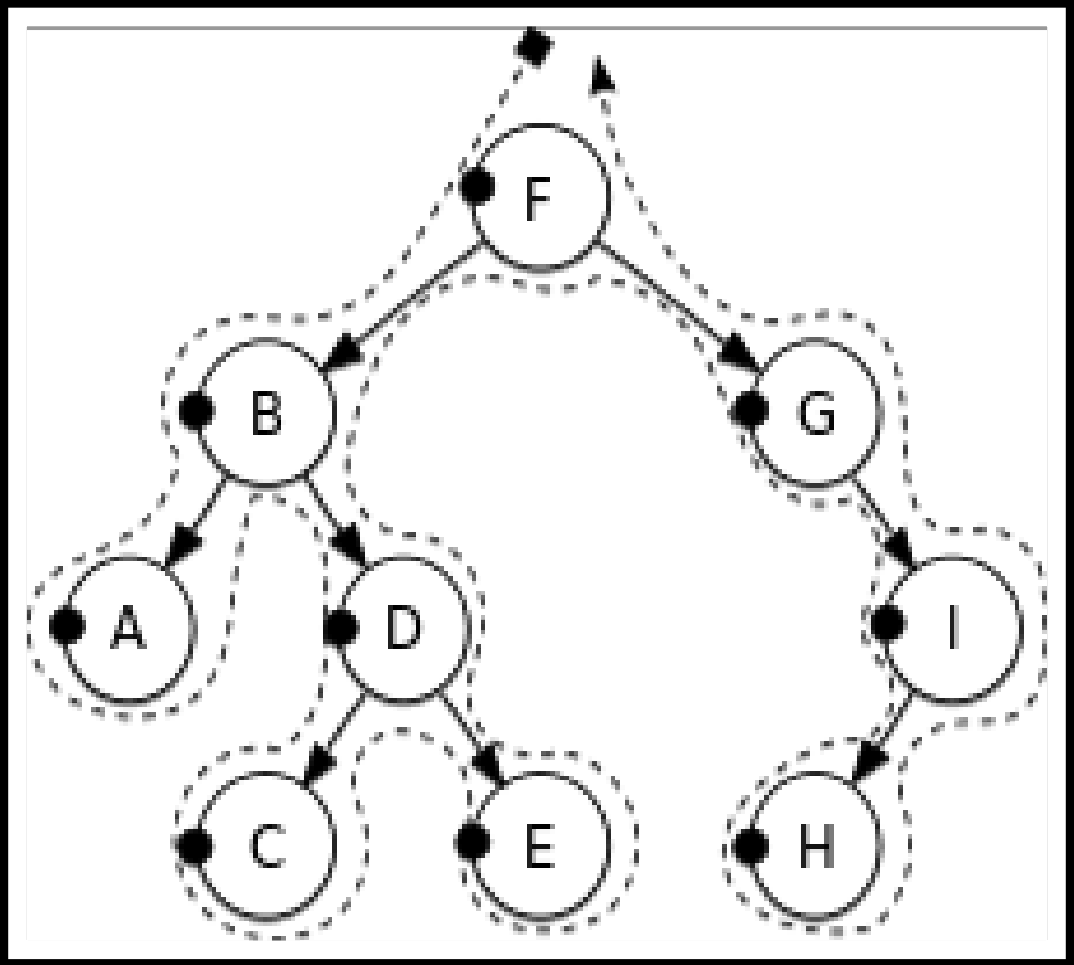
- поиск в глубину (прямой, симметричный, обратный);
- поиск в ширину.

- **Уровень** – «поколение» узла. Если корневой узел находится на уровне 0, то его следующий дочерний узел находится на уровне 1, его внук - на уровне 2, и так далее.
- **Ширина** – максимальное кол-во узлов расположенных на одной высоте;
- **Высота** – длина самого дальнего пути к листу;
- **Глубина узла** – длина пути от текущего узла к корню;
- **Степень** – количество потомков у узла.



Прямой (preorder)

Сначала обрабатывается сам узел, затем его левые потомки, а после правые.

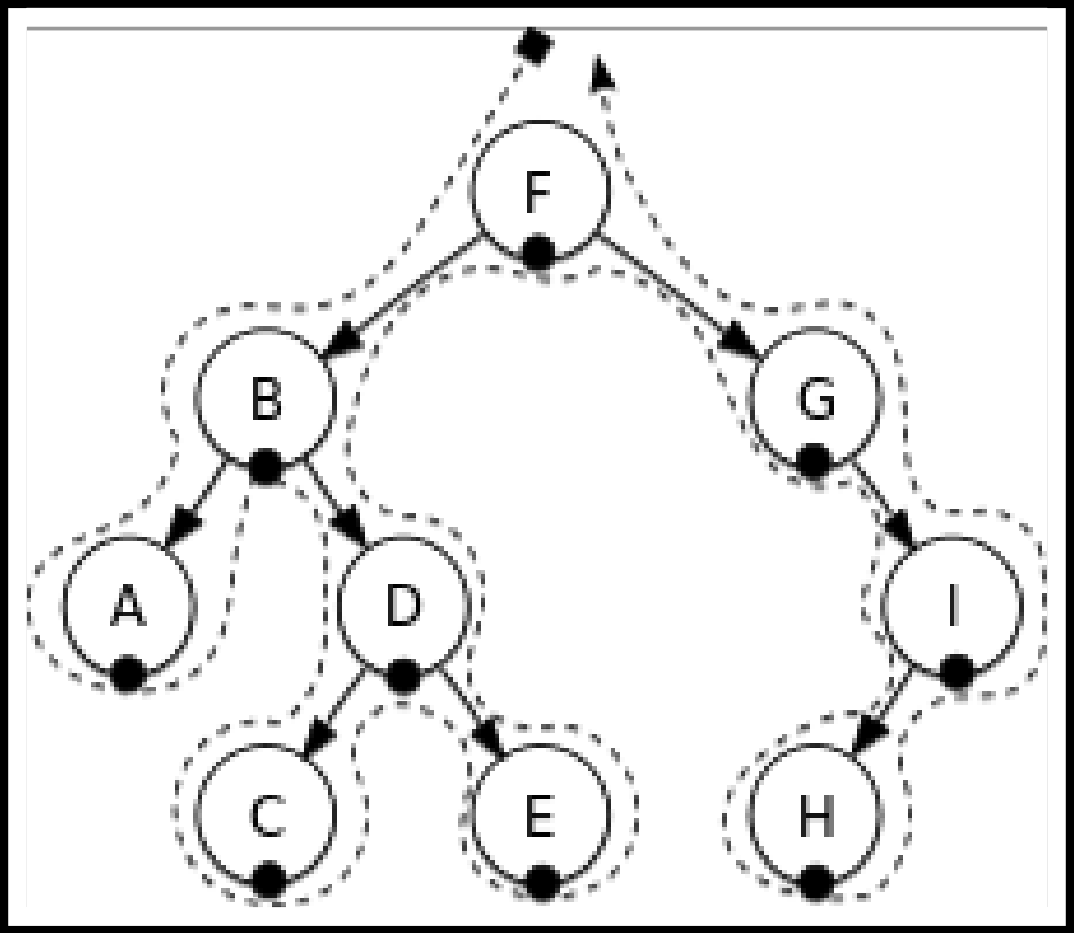


Результатом обхода будет:
F B A D C E G I H

```
preorder(node* nd) {
   if (nd == nullptr)
      return;
   task(nd);
   preorder(nd->left);
   preorder(nd->right);
}
```

Симметричном (inorder)

Сначала уходим вглубь левого поддерева, затем обрабатываем достигнутый узел, потом углубляемся в правое поддерево

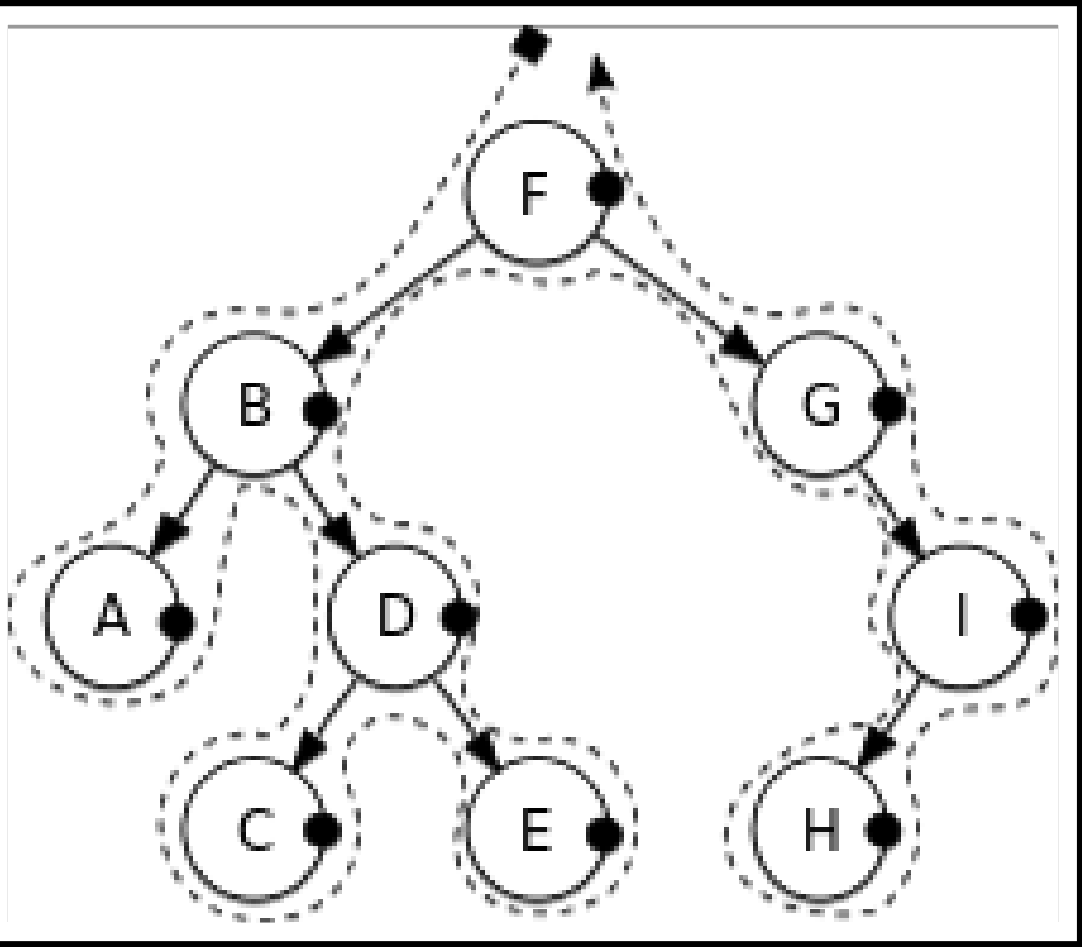


Результатом обхода будет:
A B C D E F G H I

```
inorder(node* nd) {
  if (nd == nullptr)
    return;
  inorder(nd->left);
  task(nd);
  inorder(nd->right);
}
```

Обратном (postorder)

Сначала уходим вглубь левого поддерева, оттуда вглубь правого поддерева и лишь потом обрабатываем достигнутый узел.

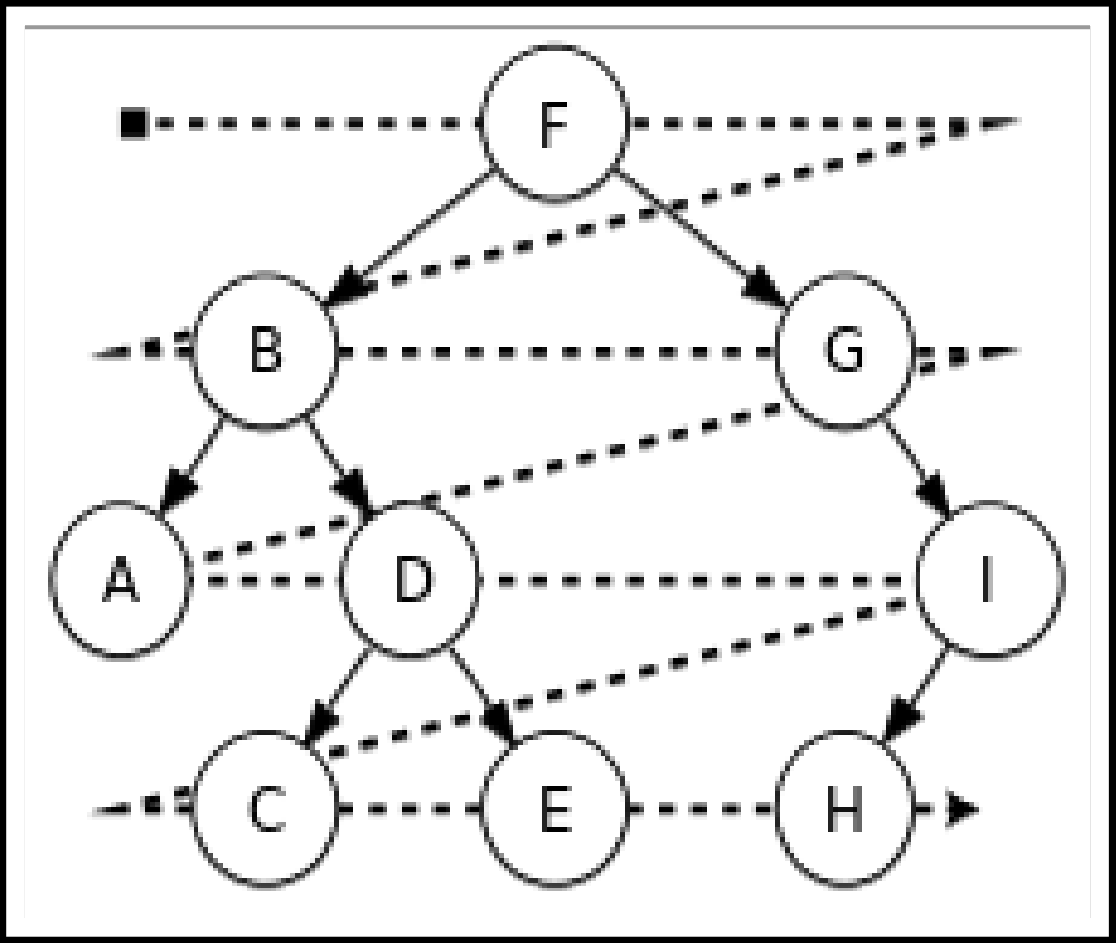


Результатом обхода будет:
A C E D B H I G F

```
postorder(node* nd) {
  if (nd == nullptr)
    return;
  postorder(nd->left);
  postorder(nd->right);
  task(nd);
}
```

Обход в ширину

Сначала обрабатывается корень, затем (слева направо) все узлы первого уровня, потом второго и т.д.



Результатом обхода будет:
F B G A D I C E H

```
widthorder(node* nd) {
  if (nd == nullptr)
    return;
  task(nd);
  queue.push(nd->left);
  queue.push(nd->rigth);
  widthorder(queue.pop);
}
```

Удаление узлов

- если у узла нет потомков — удаляем его;
- если узел имеет 1 потомка — заменяем родителя потомком;
- если узел имеет 2 (и более) потомка — **выбираем узел (из левого поддерева) с максимальным значением и заменяем им родителя;**

- удаляя узел из поддерева необходимо заменить его элементом, который **будет строго меньше правой части** удалённого узла, **и больше** (или равен) **левой части** удалённого узла.

Построим

1) 25 30 20 15 40 45 50 10 5 2 22

2) 57 16 5 35 18 88 96 17 32 71 26 58

3) 57 73 35 92 1 60 88 52 96 98

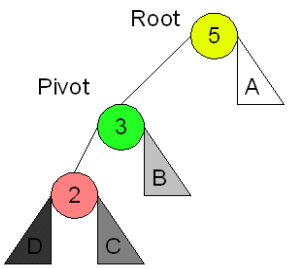
Балансировка

Важно, чтобы разница между высотами левого и правого поддеревьев была не более 1.
Соответственно, для сохранения сбалансированности, необходимо поворачивать (перестраивать) дерево.

Рекурсивный алгоритм балансировки дерева для известного числа вершин (n):

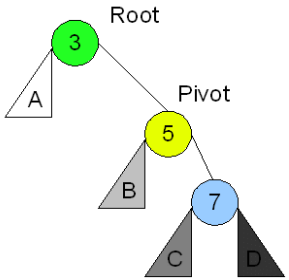
1. берём вершину в качестве корня;
2. построить для него левое поддерево с $nl = n / 2$ вершинами;
3. построить для него правое поддерево с $nr = n - nl - 1$ вершинами.

Left Left Case



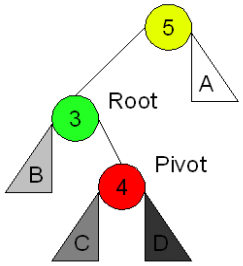
Right Rotation

Right Right Case



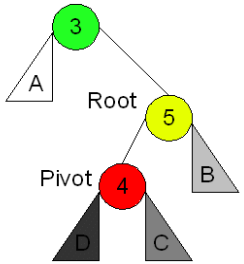
Left Rotation

Left Right Case

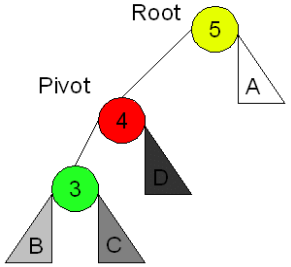


Left Rotation

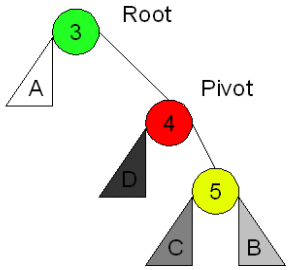
Right Left Case



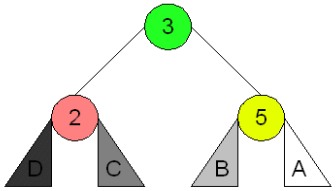
Right Rotation

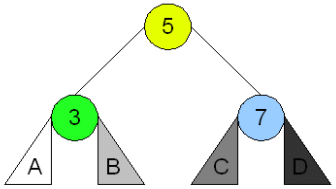


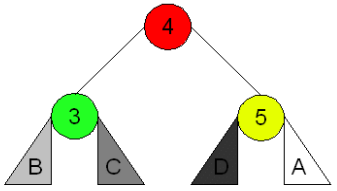
Right Rotation

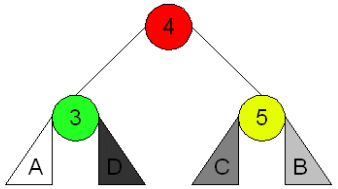


Left Rotation









Построим

1) 20 25 40 45 30 10 15 50 5 2 22 1

2) 57 71 16 5 35 96 17 32 18 88 58 26

3) 60 88 73 35 57 1 92 5 52 2 96 98

Constructors (explicit)

```
class A {
  explicit A(auto) { … }
  explicit bool() const { … };
  int* arr;
};

void PrintWhat(A obj) {
  bool b =
static_cast<bool>(obj);
  // …
}

int main() {
  PrintWhat(53); // error (ctime)
  PrintWhat(A(53));

  return 0;
}
```

```
struct V {
  operator int() const {
    return 0;
  } // I
};

struct U {
  U(V) { } // II
};

void f(U)   {cout << "U" << endl;}
void f(int) {cout << "V" << endl;}

int main() {
  V x;
  f(x); // ???

  U y(x);
  f(y); // ???
  return 0;
}
```

g++ v7.2.0 –O0

```
class Test {
  int i;
};

int main() {
  Test t;
  cout << t.i << endl; // 1.78*10^9
  Test a = t;
  cout << a.i << endl;
  return 0;
}
```

```
class Test {
  int i;
  Test(int a) : i{a} {};
};

int main() {
  Test t; // error
  cout << t.i << endl;
  Test a = t; // error
  cout << a.i << endl;
  return 0;
}
```

```
Test a = t; // Test(const Test&);

Test b;
b = a; // Test& operator= (const Test&);

Test c(std::move(t)); // Test(Test&&);
d = std::move(b);  // Test& operator= (Test&&)
```

Правило пяти

Если класс определяет один из следующих методов, то он должен явным образом определить все пять методов.

- деструктор;
- конструктор копирования;
- оператор копирования;
- конструктор перемещения;
- оператор перемещения.

Т.е. если хотя бы один из методов определён, то это означает, что остальные (методы) не удовлетворяют потребностям класса в одном случае, и вероятно не удовлетворят в остальных случаях.

Правило нуля

Не определяйте самостоятельно ни один из методов (5ки), вместо этого поручите заботу о владении ресурсами специально придуманным для этого классам.

Т.е. нужно поручить их создание компилятору (присвоив значение = default).

`std::unique_ptr, std::shared_ptr, std::weak_ptr`

```
class Test {
public:
  int* arr {nullptr};
  Test() { arr = new int[5]; };

  Test(const Test&) = default;
  Test& operator= (const Test&) =
default;
  Test(Test&&) = default;
  Test& operator= (Test&&) = default;
  ~Test() = default;
};

void main() {
  Test t1;
  for (int i = 0; i < 5; i++)
    t1.arr[i] = i;

  {
    Test t2 = t1;
    for (int i = 0; i < 5; i++)
      cout << t2.arr[i] << endl;
    t2.arr[2] = 255;
  } // ???
  for (int i = 0; i < 5; i++)
    cout << t1.arr[i] << endl;   // ???

  Test t3;
  t3 = t1;
  for (int i = 0; i < 5; i++)
    cout << t3.arr[i] << endl;   // ???
}
```

```
class Test {
public:
// will call delete[], only in C++17
  std::shared_ptr<int[]> arr;

  Test() { arr.reset(new int[5]); }
  // .reset(…); equivalent to swap
};

void main() {
    Test t1;
    for (int i = 0; i < 5; i++)
        t1.arr[i] = i;

    {
      Test t2 = t1;
      for (int i = 0; i < 5; i++)
        cout << t2.arr[i] << endl;

      t2.arr[2] = 255;
    }
    for (int i = 0; i < 5; i++)
        cout << t1.arr[i] << endl;

    Test t3;
    t3 = t1;

    for (int i = 0; i < 5; i++)
        cout << t3.arr[i] << endl;
}
```

RAII

(Resource Acquisition Is Initialization)

«Захват ресурса — есть инициализация»

Класс, чей конструктор получает доступ к ресурсу, отвечает за освобождение ресурса.

SBRM (Scope Bound Resource Management) == RAII.
В качестве ресурса может выступать: блок памяти, сетевой сокет, файловый дескриптор, мьютекс и т. д.

В конструкторе класса **организуется получение доступа** к ресурсу, а в деструкторе — **освобождение** этого ресурса.

Т.к. локальные переменные (объекты) размещаются на стеке, то при его развёртывании (выходе из области, scope), ресурс гарантированно освобождается при уничтожении переменной (вызовом деструктора).

Необходимо контролировать возможность создания объекта через операцию клонирования и корректно переопределить (или запретить) операцию присваивания для подобных объектов.

```
class MThread {
    std::thread m_thread;
public:
    MThread(std::thread&& thread) {
        m_thread.swap(thread);
    }

    ~MThread() {
        if (m_thread.joinable())
            m_thread.join();
    }
};
```

```
#include <thread>

int main() {
    MThread wrp_one(std::move
                    (std::thread(mass_print))
                );
    MThread wrp_two(std::move
                    (std::thread(mass_print))
                );

    return 0;
}
```

```
void mass_print() {
    for (int i = 0; i < 10; i++)
        cout << "ID: "
             << this_thread::get_id()
             << endl; }
```

Тривиальные типы

Класс (структура) который одновременно и тривиально конструируем по умолчанию и тривиально копируем, т.е. подразумевает что:

- использует неявно определенные значения по умолчанию, конструкторы копирования и перемещения, операторы копирования и перемещения, и деструктор;
- не имеет виртуальных членов;
- не имеет нестатических членов с {} или = инициализацией;
- его родительский класс и нестатические члены данных сами также являются тривиальными типами.

```
#include <type_traits>
```

`std::is_trivial<T>` - проверяет, является ли класс тривиальным;

`std::is_trivially_copyable<T>` – проверяет, является ли класс тривиально копируемым.

Тривиальные типы

```
// https://wandbox.org/permlink/Fa9UXci4zy6g5xyv
#include <iostream>
#include <type_traits>
using namespace std;

struct A { int i; };

struct B {
  int i,j;
  //B (const B& x) : i(x.i), j(1) {};  // copy ctor
};

int main() {
  cout << boolalpha;
  cout << "is_trivially_copyable:" << endl;

  cout << "int: " << is_trivially_copyable<int>::value << endl;
  cout << "A: "   << is_trivially_copyable<A>::value << endl;
  cout << "B: "   << is_trivially_copyable<B>::value << endl;
  return 0;
}
```

Aggregate types

Агрегат — массив или класс без user-provided конструктора, без brace-or-equal-initializers для нестатических полей, без приватных или защищённых (protected) нестатических полей. Не имеет родительских классов и виртуальных функций.

`std::is_aggregate<T>` - проверяет, является ли класс агрегатом

```
class Aggr {
public:
  Aggr() = default; // user-defined
  int i, j, e;
};
```

POD types

Plain Old Data — класс/структура без конструктора, деструктора и виртуальных членов. Может быть обменен с библиотеками C напрямую.

POD — агрегатный класс, хранящий такие же PODs члены, не имеющий поля ссылочного типа, не обладает пользовательским конструктором копирования.

std::is_pod<T> - проверяет, является ли тип типом простой структуры данных (POD) или нет.

```
cout << std::is_pod<A>::value;
```

braced-init-list (std::initializer_list)

```
vector v       (3, 0);          // 0, 0, 0
vector v       {3, 0};          // 3, 0
MyClass obj {};                 // call default ctor
MyClass obj {23, 24, 25}; // std::initializer_list<int>
```

Для:

- **агрегатов** — агрегирующая инициализация;
- **классов** — в первую очередь ищет конструктор поддерживающий std::initializer_list<T>; если не находит, вызывает первый подходящий.

```
class Test {
public:
  Test()            { cout << "def" << endl; };
  Test(const Test&) { cout << "copy" << endl;  };
  Test(auto, auto)  { cout << "param" << endl; };
  Test(std::initializer_list<int> list) { cout << "list" << endl; };
};


class aggreg {
  public:
  int i, j, e;
};


int main() {
  Test t {32, 3};           // list
  Test a = {32, 25, 235};   // list
  Test b {32, 2.5, 235};    // error
  Test c {t};               // copy
  Test d {};                // def
  Test e (24, 2.5);         // param
  Test f ();                // ???

  aggreg ag {1, 5};
  cout << ag.i << " " << ag.j << " " << ag.e << endl;  // 1 5 0

  return 0;
}
```

https://youtu.be/2jJumNzcp6Y