数据结构与算法

DATA STRUCTURE

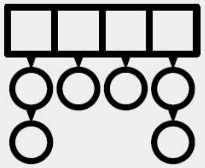
第二十八讲 复习

胡浩栋

信息管理与工程学院

2017 - 2018 第一学期

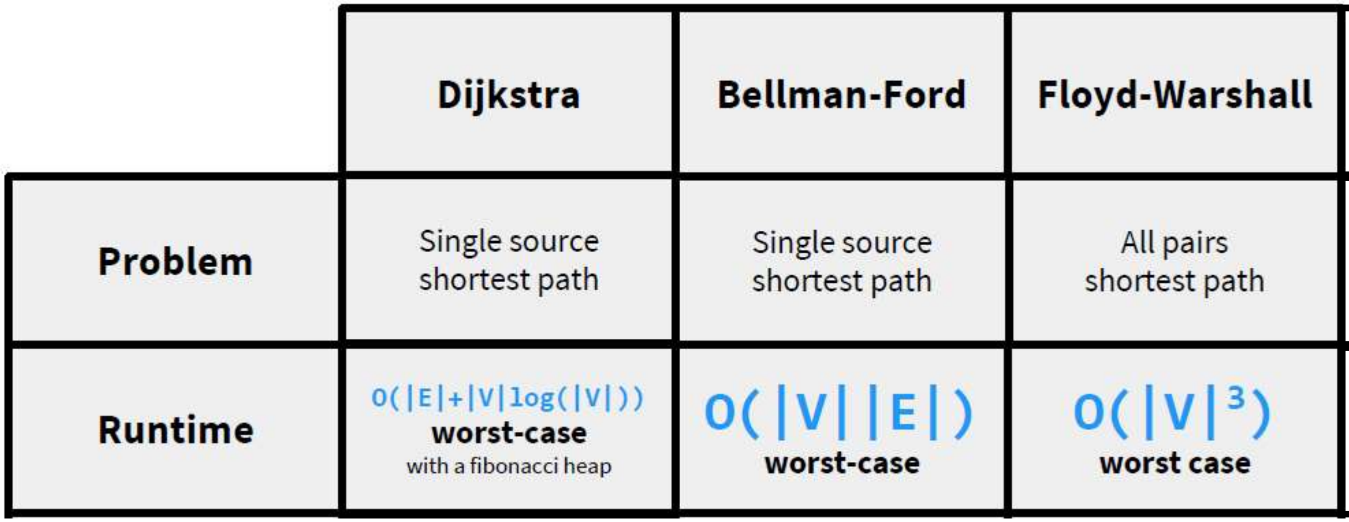
图的表示

For G = (V, E)	$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$	
Edge Membership Is e = {u, v} in E?	$O(1)$	$O(\mathrm{deg}(u))$ **or** $O(\mathrm{deg}(v))$
Neighbor Query What are the neighbors of u?	$O(\|V\|)$	$O(\mathrm{deg}(v))$
Space requirements	$O(\|V\|^2)$	$O(\|V\|+\|E\|)$

图遍历算法

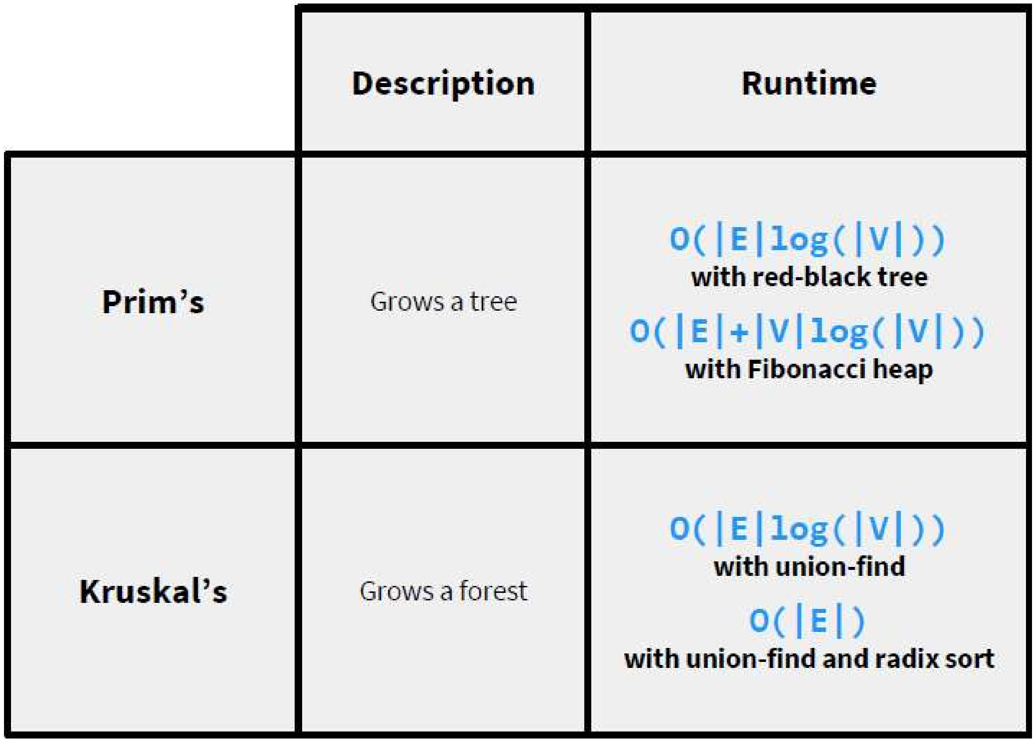
- 深度优先
  - 深度优先+始末时间
  - 拓扑排序
  - 检测图上的cycle
  - 强连通分支 （作业）
- 广度优先
  - 按边算的最短路径
  - 检测二分图
  - 无向图上的连通分支

最短路径算法和动态规划算法



	Dijkstra	Bellman-Ford	Floyd-Warshall	DAG上的 最短路径
Problem	Single source shortest path	Single source shortest path	All pairs shortest path	Single source
Runtime	O(|E|+|V|log(|V|)) **worst-case** with a fibonacci heap	O(|V||E|) **worst-case**	O(|V|³) **worst case**	O(|V|+|E|)

最小生成树MST和贪婪算法



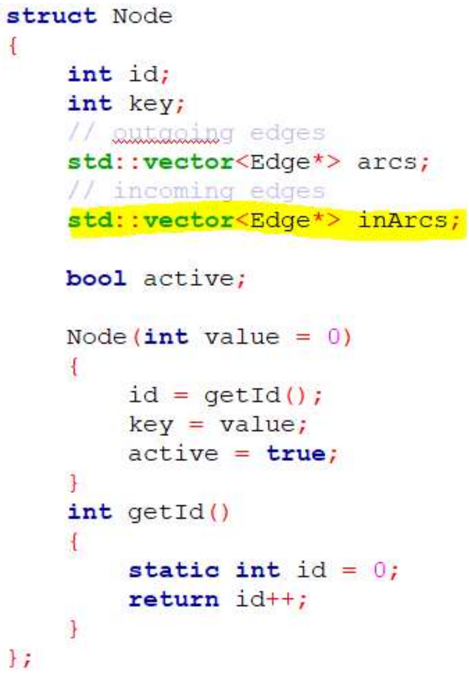
	Description	Runtime

Prim's	Grows a tree	O(|E|log(|V|))
		with red-black tree
		O(|E|+|V|log(|V|))
		with Fibonacci heap

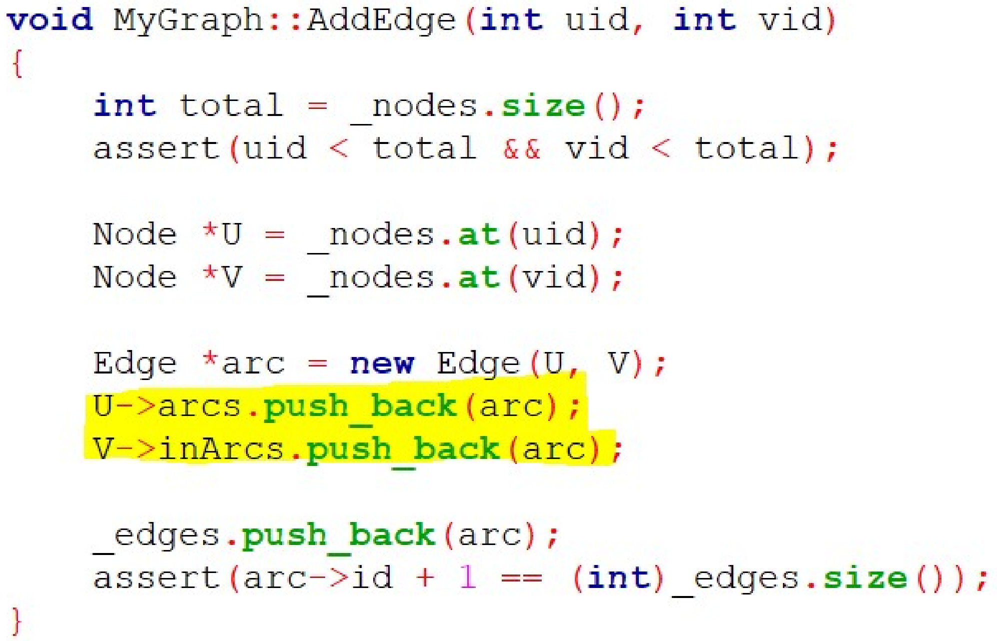
Kruskal's	Grows a forest	O(|E|log(|V|))
		with union-find
		O(|E|)
		with union-find and radix sort

FindScc

- 每个节点保存了出边集合和入边集合
- 每添加一条边，需要在两个顶点结构体里加边



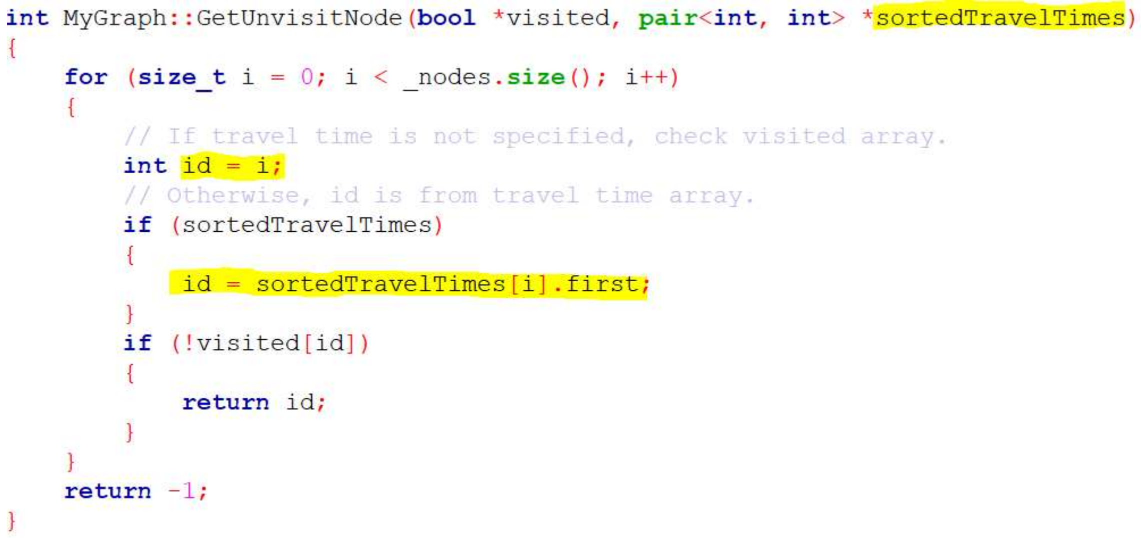
```
struct Node
{
    int id;
    int key;
    // outgoing edges
    std::vector<Edge*> arcs;
    // incoming edges
    std::vector<Edge*> inArcs;

    bool active;

    Node(int value = 0)
    {
        id = getId();
        key = value;
        active = true;
    }
    int getId()
    {
        static int id = 0;
        return id++;
    }
};
```



```
void MyGraph::AddEdge(int uid, int vid)
{
    int total = _nodes.size();
    assert(uid < total && vid < total);

    Node *U = _nodes.at(uid);
    Node *V = _nodes.at(vid);

    Edge *arc = new Edge(U, V);
    U->arcs.push_back(arc);
    V->inArcs.push_back(arc);

    _edges.push_back(arc);
    assert(arc->id + 1 == (int)_edges.size());
}
```

FindScc

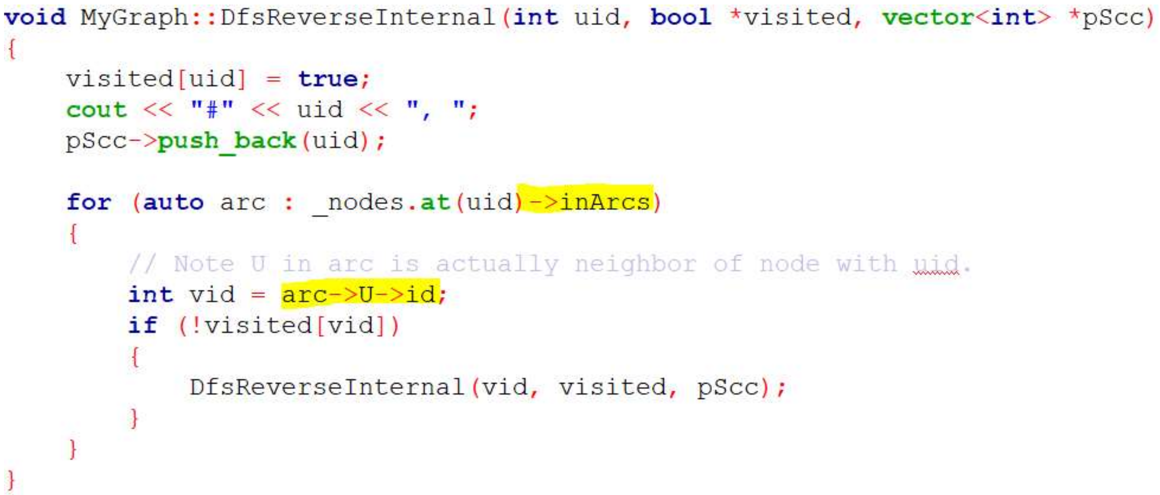
- 辅助函数，找下一个没访问过的节点
- 如果有按结束时间排序的traveltime，需要找结束时间最晚的第一个没访问过的节点



```
int MyGraph::GetUnvisitNode(bool *visited, pair<int, int> *sortedTravelTimes)
{
    for (size_t i = 0; i < _nodes.size(); i++)
    {
        // If travel time is not specified, check visited array.
        int id = i;
        // Otherwise, id is from travel time array.
        if (sortedTravelTimes)
        {
            id = sortedTravelTimes[i].first;
        }
        if (!visited[id])
        {
            return id;
        }
    }
    return -1;
}
```

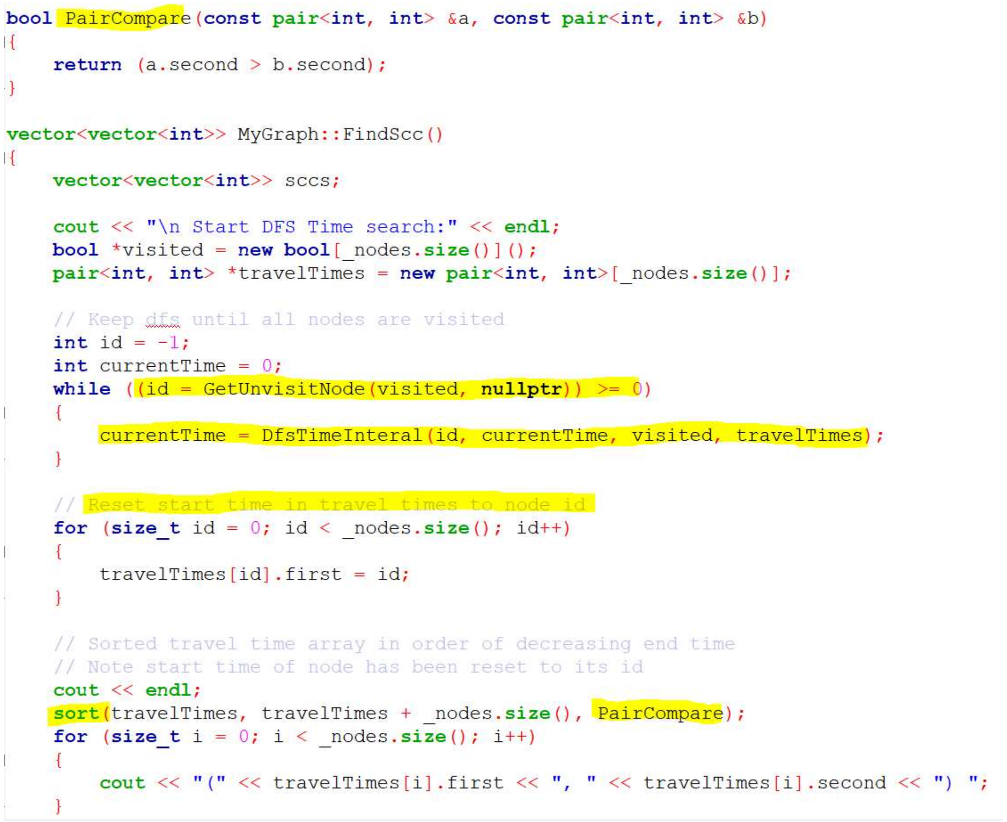
FindScc

- 按反向边的DFS递归函数
- 因为我们在节点里加了入边，所以直接用入边进行DFS
- 但注意邻接顶点是arc里的U

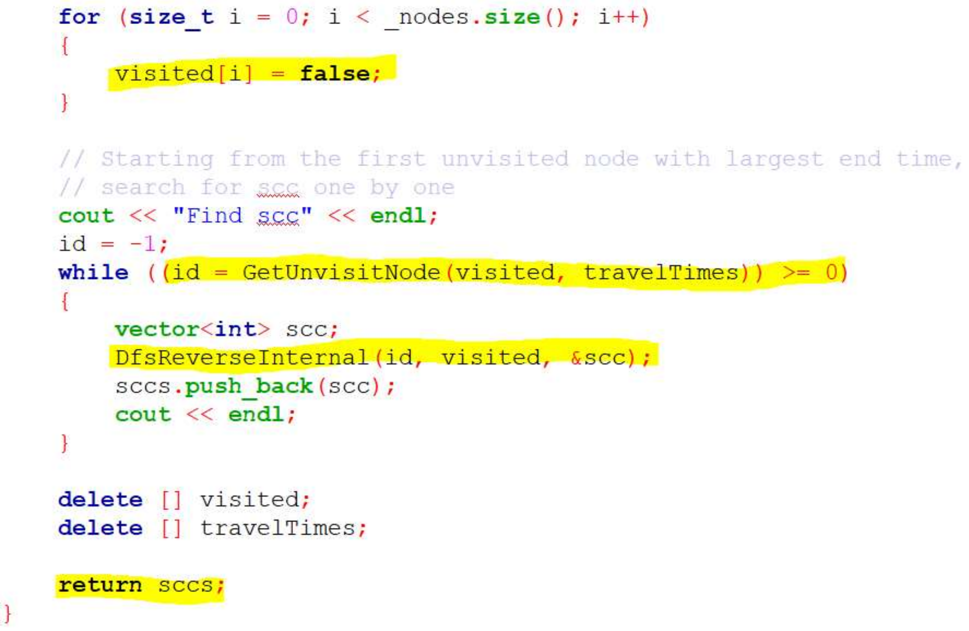


```
void MyGraph::DfsReverseInternal(int uid, bool *visited, vector<int> *pScc)
{
    visited[uid] = true;
    cout << "#" << uid << ", ";
    pScc->push_back(uid);

    for (auto arc : _nodes.at(uid)->inArcs)
    {
        // Note U in arc is actually neighbor of node with uid.
        int vid = arc->U->id;
        if (!visited[vid])
        {
            DfsReverseInternal(vid, visited, pScc);
        }
    }
}
```

FindScc

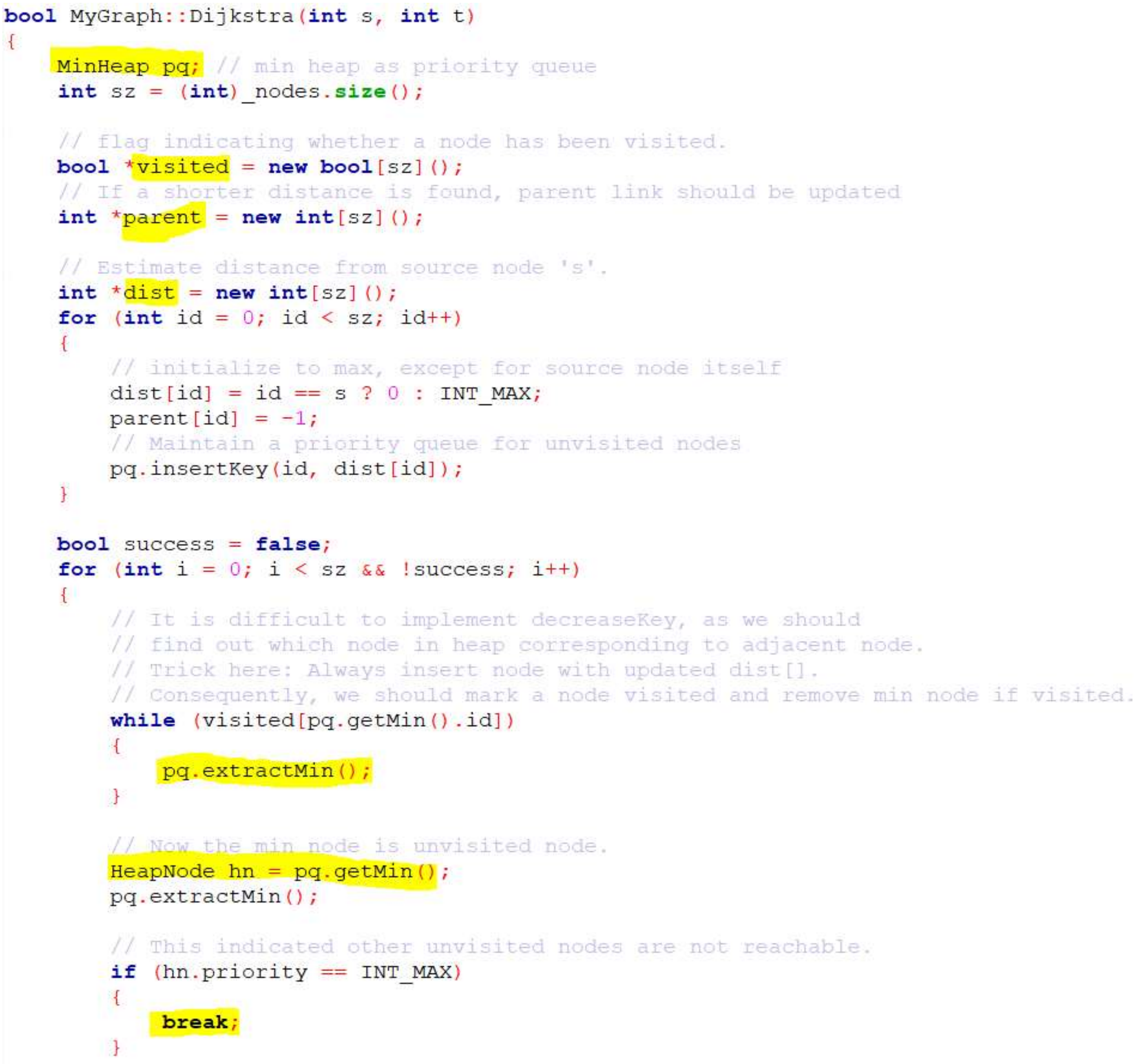


```
bool PairCompare(const pair<int, int> &a, const pair<int, int> &b)
{
    return (a.second > b.second);
}

vector<vector<int>> MyGraph::FindScc()
{
    vector<vector<int>> sccs;

    cout << "\n Start DFS Time search:" << endl;
    bool *visited = new bool[_nodes.size()]();
    pair<int, int> *travelTimes = new pair<int, int>[_nodes.size()];

    // Keep dfs until all nodes are visited
    int id = -1;
    int currentTime = 0;
    while ((id = GetUnvisitNode(visited, nullptr)) >= 0)
    {
        currentTime = DfsTimeInteral(id, currentTime, visited, travelTimes);
    }

    // Reset start time in travel times to node id
    for (size_t id = 0; id < _nodes.size(); id++)
    {
        travelTimes[id].first = id;
    }

    // Sorted travel time array in order of decreasing end time
    // Note start time of node has been reset to its id
    cout << endl;
    sort(travelTimes, travelTimes + _nodes.size(), PairCompare);
    for (size_t i = 0; i < _nodes.size(); i++)
    {
        cout << "(" << travelTimes[i].first << ", " << travelTimes[i].second << ") ";
    }
```



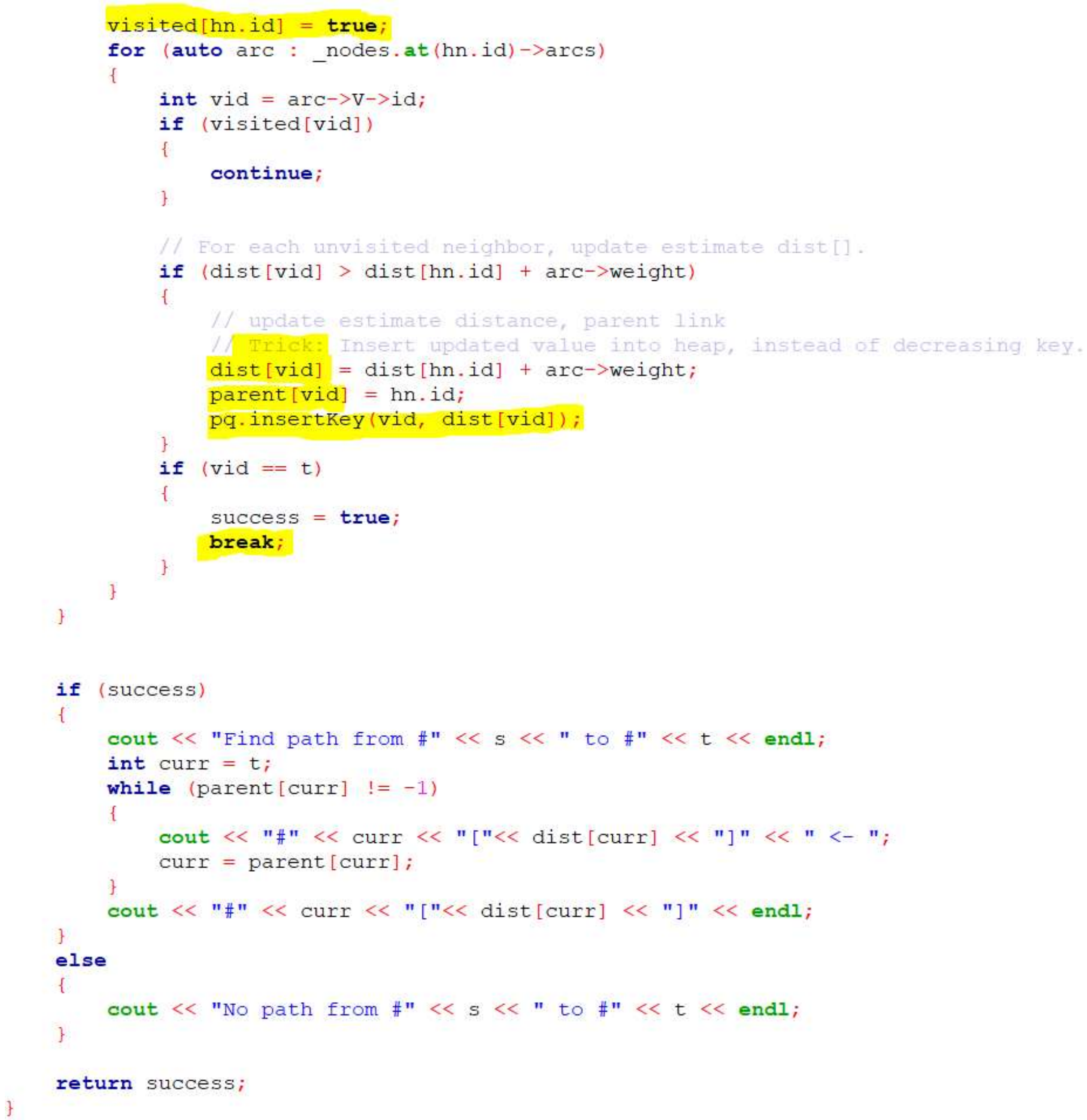
```
    for (size_t i = 0; i < _nodes.size(); i++)
    {
        visited[i] = false;
    }

    // Starting from the first unvisited node with largest end time,
    // search for scc one by one
    cout << "Find scc" << endl;
    id = -1;
    while ((id = GetUnvisitNode(visited, travelTimes)) >= 0)
    {
        vector<int> scc;
        DfsReverseInternal(id, visited, &scc);
        sccs.push_back(scc);
        cout << endl;
    }

    delete [] visited;
    delete [] travelTimes;

    return sccs;
}
```

Dijkstra



```
bool MyGraph::Dijkstra(int s, int t)
{
    MinHeap pq; // min heap as priority queue
    int sz = (int)_nodes.size();

    // flag indicating whether a node has been visited.
    bool *visited = new bool[sz]();
    // If a shorter distance is found, parent link should be updated
    int *parent = new int[sz]();

    // Estimate distance from source node 's'.
    int *dist = new int[sz]();
    for (int id = 0; id < sz; id++)
    {
        // initialize to max, except for source node itself
        dist[id] = id == s ? 0 : INT_MAX;
        parent[id] = -1;
        // Maintain a priority queue for unvisited nodes
        pq.insertKey(id, dist[id]);
    }

    bool success = false;
    for (int i = 0; i < sz && !success; i++)
    {
        // It is difficult to implement decreaseKey, as we should
        // find out which node in heap corresponding to adjacent node.
        // Trick here: Always insert node with updated dist[].
        // Consequently, we should mark a node visited and remove min node if visited.
        while (visited[pq.getMin().id])
        {
            pq.extractMin();
        }

        // Now the min node is unvisited node.
        HeapNode hn = pq.getMin();
        pq.extractMin();

        // This indicated other unvisited nodes are not reachable.
        if (hn.priority == INT_MAX)
        {
            break;
        }
```

Dijkstra



```
        visited[hn.id] = true;
        for (auto arc : _nodes.at(hn.id)->arcs)
        {
            int vid = arc->V->id;
            if (visited[vid])
            {
                continue;
            }

            // For each unvisited neighbor, update estimate dist[].
            if (dist[vid] > dist[hn.id] + arc->weight)
            {
                // update estimate distance, parent link
                // Trick: Insert updated value into heap, instead of decreasing key.
                dist[vid] = dist[hn.id] + arc->weight;
                parent[vid] = hn.id;
                pq.insertKey(vid, dist[vid]);
            }
            if (vid == t)
            {
                success = true;
                break;
            }
        }
    }


    if (success)
    {
        cout << "Find path from #" << s << " to #" << t << endl;
        int curr = t;
        while (parent[curr] != -1)
        {
            cout << "#" << curr << "["<< dist[curr] << "]" << " <- ";
            curr = parent[curr];
        }
        cout << "#" << curr << "["<< dist[curr] << "]" << endl;
    }
    else
    {
        cout << "No path from #" << s << " to #" << t << endl;
    }

    return success;
}
```

排序算法

- MergeSort
- QuickSort
- CoutingSort
- RadixSort
- BucketSort

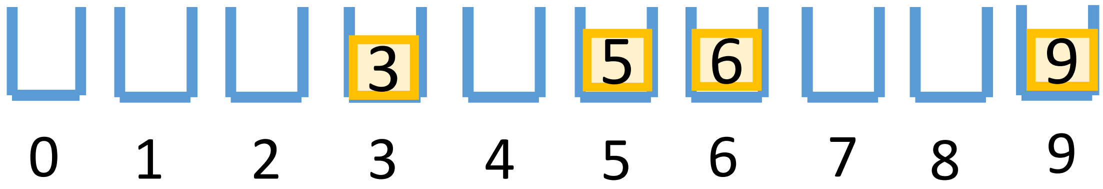
Couting sort

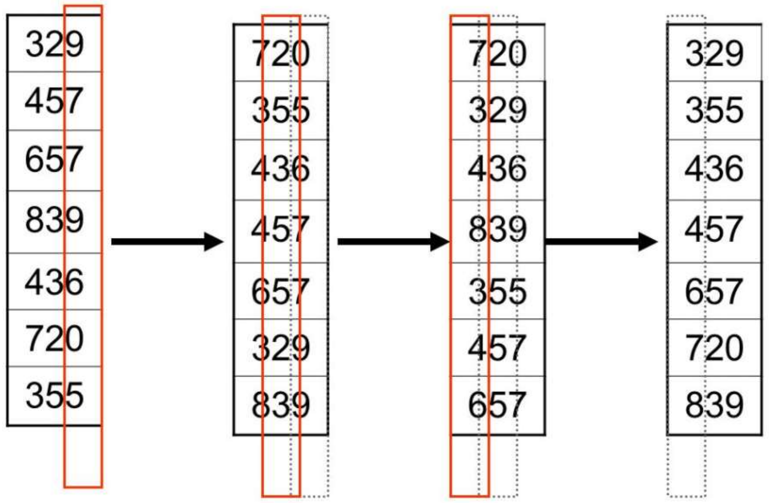
0	0	3	1	1	3	1	0

3	3	0	2

0	0	0	1	1	1	3	3

Bucket sort和□radix sort





329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

基本数据结构

- 数组和链表
- Stack
- Queue
- Recursion
- Heap
- HeapSort

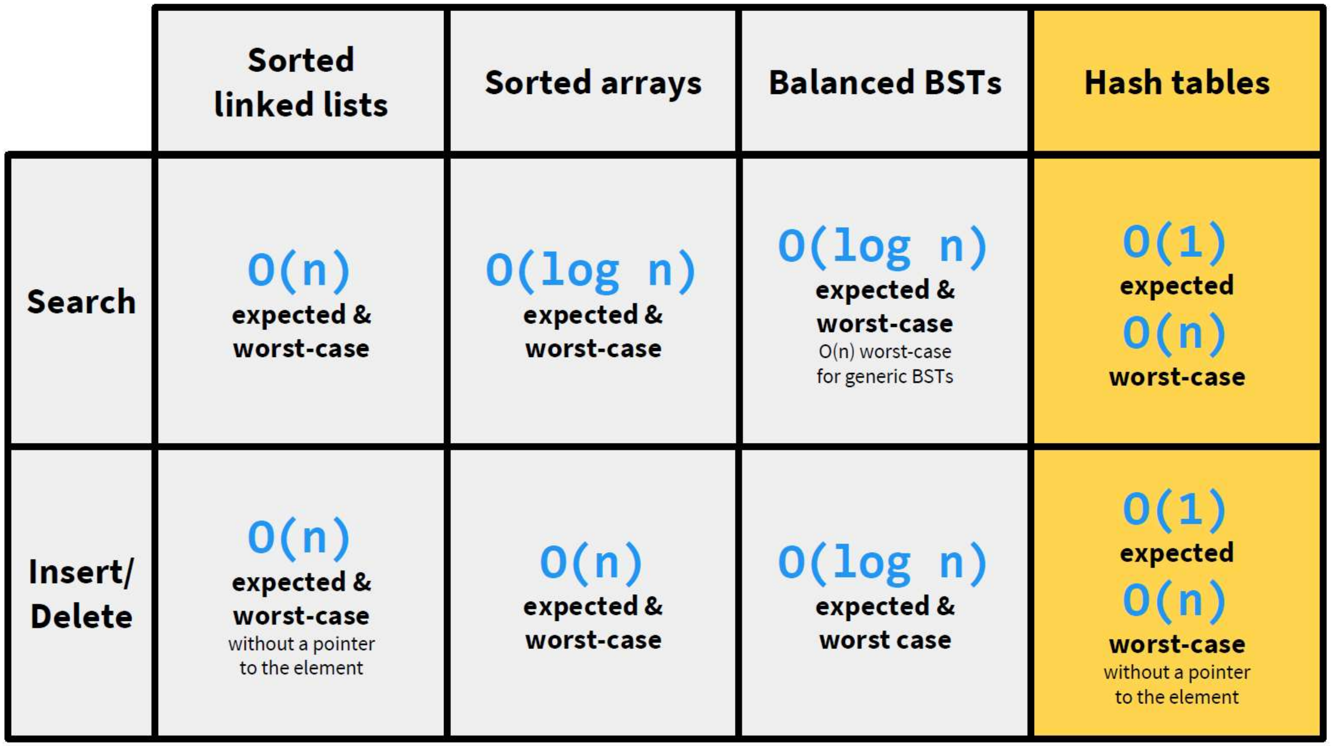
串匹配算法

- 暴力算法
- KR算法（hashing值）
- KMP（最长前后缀匹配长度，next数组）
- BM （好后缀和坏字符规则）
- 改进的Boyer–Moore–Horspool算法（只有坏字符规则）

树

- 数的遍历
  - 中序，前序，后序
  - 遍历迭代算法
- 二叉树（线索化）
- 二叉查找树
  - AVL
  - 红黑树
- B树和B+树
- Trie
- Hash
- （习题参考Lecture20.examples.pdf）

比较



	Sorted linked lists	Sorted arrays	Balanced BSTs	Hash tables

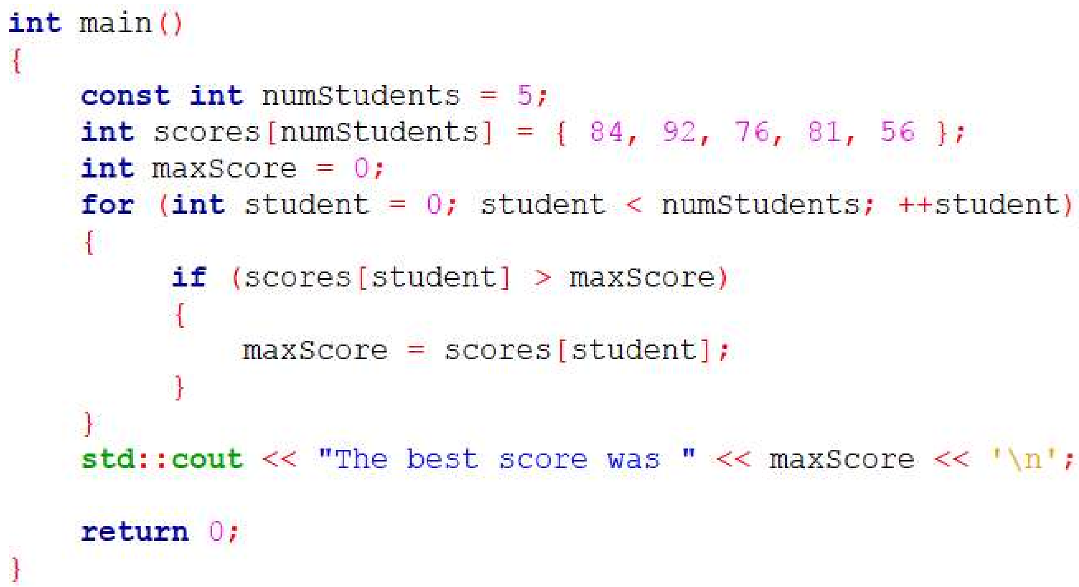
Search	O(n)
	expected & worst-case
		O(log n)
		expected & worst-case
			O(log n)
			expected & worst-case
			O(n) worst-case for generic BSTs
				O(1)
				expected
				O(n)
				worst-case

Insert/ Delete	O(n) expected & worst-case without a pointer to the element	O(n) expected & worst-case	O(log n) expected & worst case	O(1) expected O(n) worst-case without a pointer to the element

C++11新语句和STL

[C++11] for-each语句

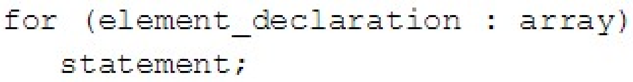
- for语句，用循环变量student作为数组下标操作



```
int main()
{
    const int numStudents = 5;
    int scores[numStudents] = { 84, 92, 76, 81, 56 };
    int maxScore = 0;
    for (int student = 0; student < numStudents; ++student)
    {
        if (scores[student] > maxScore)
        {
            maxScore = scores[student];
        }
    }
    std::cout << "The best score was " << maxScore << '\n';

    return 0;
}
```

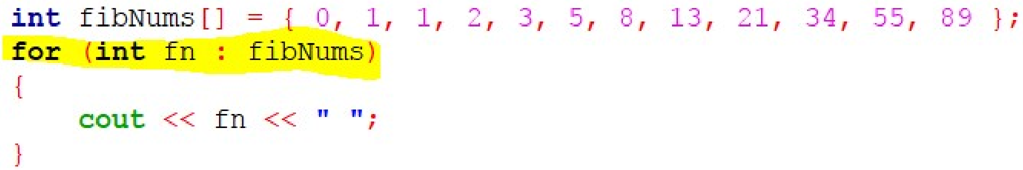
[C++11] for-each语句

- for-each语句



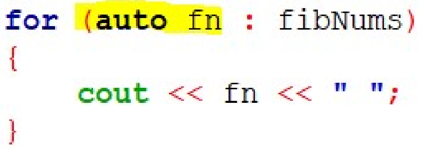
```
for (element_declaration : array)
    statement;
```

- 这里fn不是下标，fn是对fibNums[]数组元素的复制



```
int fibNums[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
for (int fn : fibNums)
{
    cout << fn << " ";
}
```

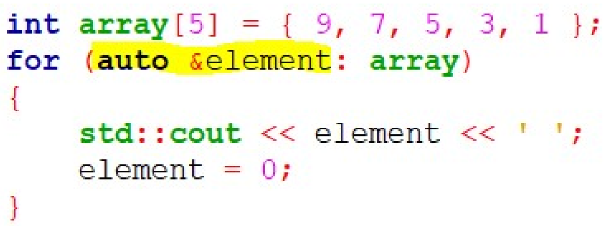
- 可以直接和auto搭配使用，更方便



```
for (auto fn : fibNums)
{
    cout << fn << " ";
}
```

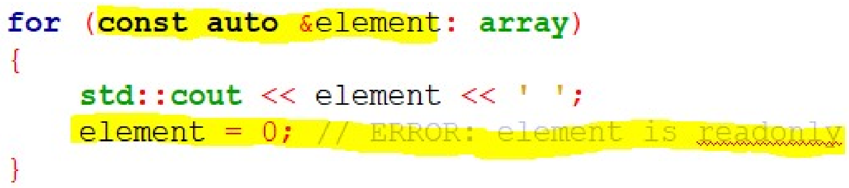
for-each和引用

- 如果加上引用（&），不再传值，可以修改原数组值



```
int array[5] = { 9, 7, 5, 3, 1 };
for (auto &element: array)
{
    std::cout << element << ' ';
    element = 0;
}
```

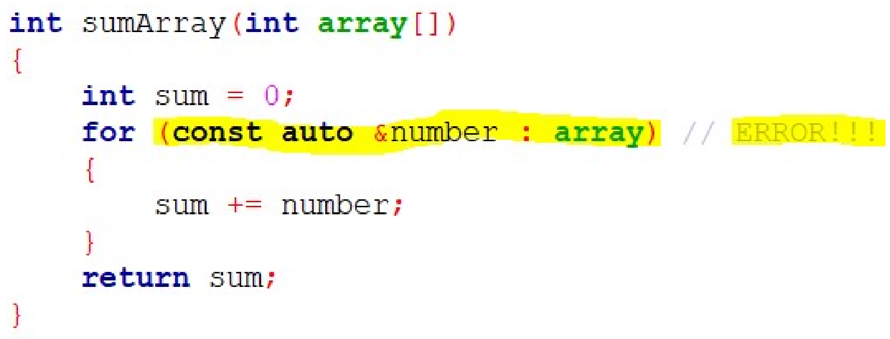
- 如果即要避免元素值复制，又不要修改原数据，用const reference



```
for (const auto &element: array)
{
    std::cout << element << ' ';
    element = 0; // ERROR: element is readonly
}
```

for-each语句

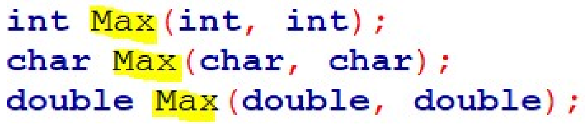
- 不能用在数组形参上，因为这个形参就是指针，不知道数组长度
- 所以参数形式最好改成指针，避免误解



```
int sumArray(int array[])
{
    int sum = 0;
    for (const auto &number : array) // ERROR!!!
    {
        sum += number;
    }
    return sum;
}
```

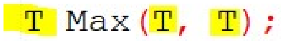
Template函数模板

- 如果一组重载函数仅仅是参数的类型不一样，程序的逻辑完全一样，那么这一组重载函数可以写成一个函数模板。



```
int Max(int, int);
char Max(char, char);
double Max(double, double);
```

- 所谓的函数模板就是实现类型的参数化（泛型化），即把函数中某些形式参数的类型定义成参数，称为模板参数



```
T Max(T, T);
```

- 在函数调用时，编译器根据实际参数的类型确定模板参数的值，生成不同的模板函数

函数模板的定义

- 一 般的定义形式

  template<模板形式参数表>

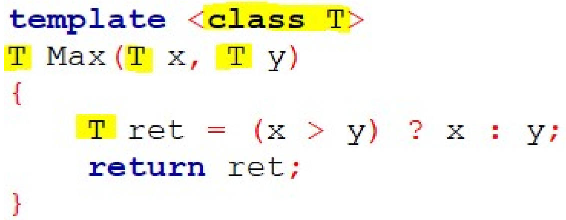
  返回类型 FunctionName(形式参数表)

  {

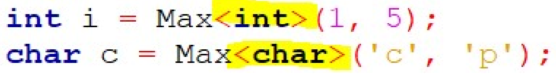
  //函数定义体

  }

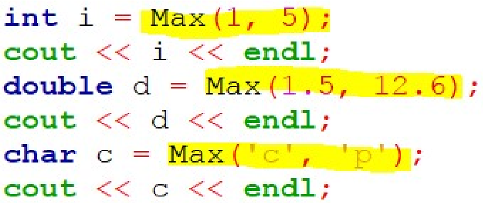
- 模板形式参数表可以包含基本数据类型，也可以包含类类型（需加前缀class或者typename）



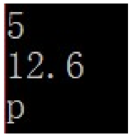
```
template <class T>
T Max(T x, T y)
{
    T ret = (x > y) ? x : y;
    return ret;
}
```



```
int i = Max<int>(1, 5);
char c = Max<char>('c', 'p');
```



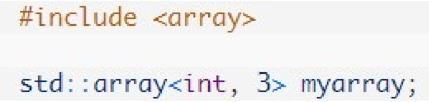
```
int i = Max(1, 5);
cout << i << endl;
double d = Max(1.5, 12.6);
cout << d << endl;
char c = Max('c', 'p');
cout << c << endl;
```



```
5
12.6
p
```

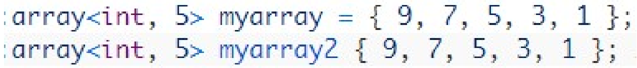
Array in C++11

- 固定长度（静态）数组。#include <array>

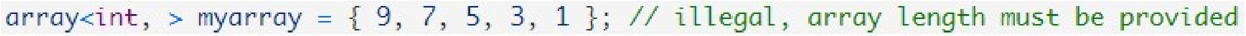


```
#include <array>

std::array<int, 3> myarray;
```

- 初始化

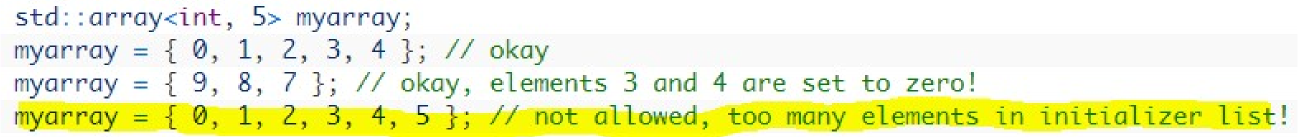


```
:array<int, 5> myarray = { 9, 7, 5, 3, 1 };
:array<int, 5> myarray2 { 9, 7, 5, 3, 1 };
```



```
array<int, > myarray = { 9, 7, 5, 3, 1 }; // illegal, array length must be provided
```

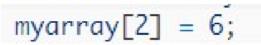
- 赋值



```
std::array<int, 5> myarray;
myarray = { 0, 1, 2, 3, 4 }; // okay
myarray = { 9, 8, 7 }; // okay, elements 3 and 4 are set to zero!
myarray = { 0, 1, 2, 3, 4, 5 }; // not allowed, too many elements in initializer list!
```

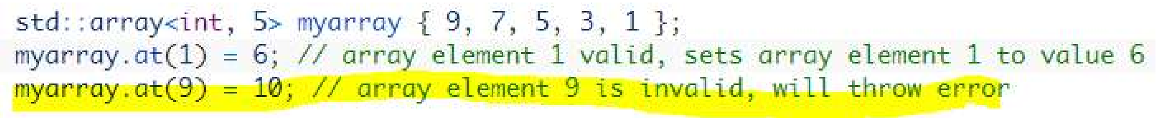
Array in C++11

- 用下标运算符[]取值，不检查越界



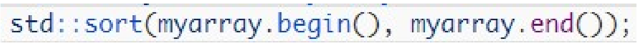
```
myarray[2] = 6;
```

- 用at()取值，检查越界



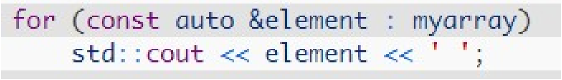
```
std::array<int, 5> myarray { 9, 7, 5, 3, 1 };
myarray.at(1) = 6; // array element 1 valid, sets array element 1 to value 6
myarray.at(9) = 10; // array element 9 is invalid, will throw error
```

- Size()取长度
- Sort()



```
std::sort(myarray.begin(), myarray.end());
```

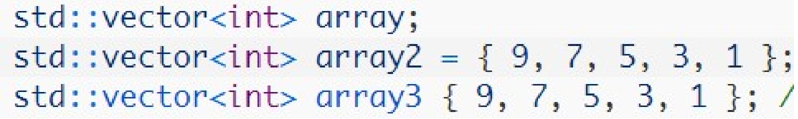
- 遍历



```
for (const auto &element : myarray)
    std::cout << element << ' ';
```

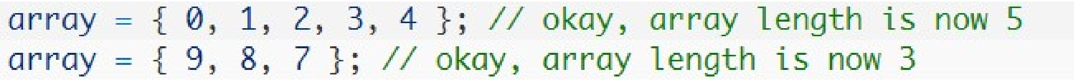
vector in C++03

- 动态数组。#include <vector>
- 初始化



```
std::vector<int> array;
std::vector<int> array2 = { 9, 7, 5, 3, 1 };
std::vector<int> array3 { 9, 7, 5, 3, 1 }; /
```

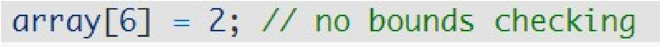
- C++11赋值。数组是动态的



```
array = { 0, 1, 2, 3, 4 }; // okay, array length is now 5
array = { 9, 8, 7 }; // okay, array length is now 3
```

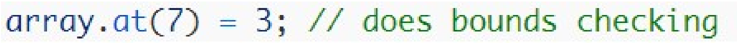
vector in C++03

- 用下标运算符[]取值，不检查越界



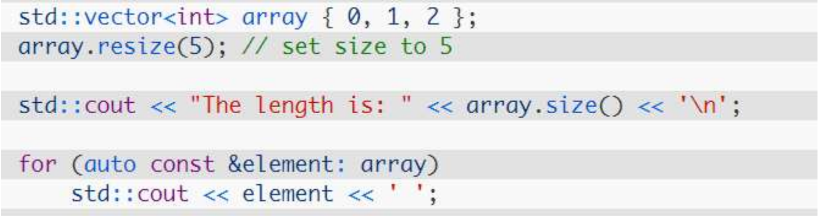
```
array[6] = 2; // no bounds checking
```

- 用at()取值，检查越界



```
array.at(7) = 3; // does bounds checking
```

- Size()取长度，Resize()改变长度
- 遍历



```
std::vector<int> array { 0, 1, 2 };
array.resize(5); // set size to 5

std::cout << "The length is: " << array.size() << '\n';

for (auto const &element: array)
    std::cout << element << ' ';
```

- 自动清理内存，不用担心泄露

小结

- 新的for-each语句可以使得数组遍历更容易，安全
- STL的静态数组（固定长度）：array<int, 10> stlArr
  - 可以替代C++静态数组：int arr[10]
  - 提供更多的功能，更安全
  - 注意作为函数的参数的话，不再是数组地址，而是传值，所以最好使用const reference，即传array<>的常量引用
- STL的动态数组：vector<int> stlVec
  - 可以替代C++动态数组：int *ptr = new int[length]
  - 提供更多的功能，更安全，不需要考虑内存释放的问题
  - 同样作为函数的参数的话，要注意传值的性能问题。

Q&A

Thanks!