Ascend Educare

C++ Elective – Day 2

Content created by
Vishwajeetsinh Jadeja

D E Shaw & Co

Contents

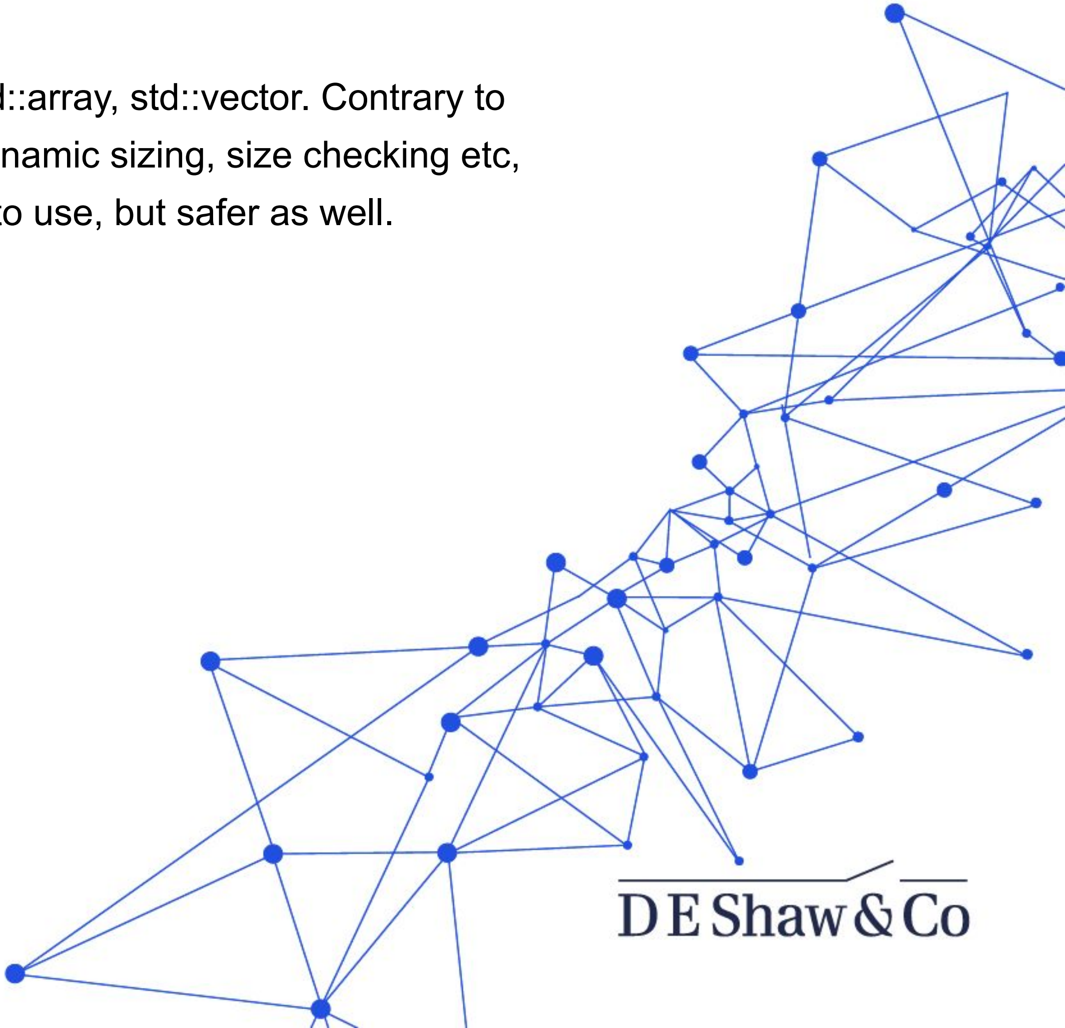
- STL
- Deleted, default functions, lambdas
- Threading library
- Concurrency concepts
- Sockets

STL

- Collection of classes that provide templated containers, algorithms, and iterators
- Programmer can take advantage of these classes without having to write and debug the classes themselves
- The standard library does a good job providing reasonably efficient versions of these classes.
- Standard library is complex and can be a little intimidating since everything is templated.

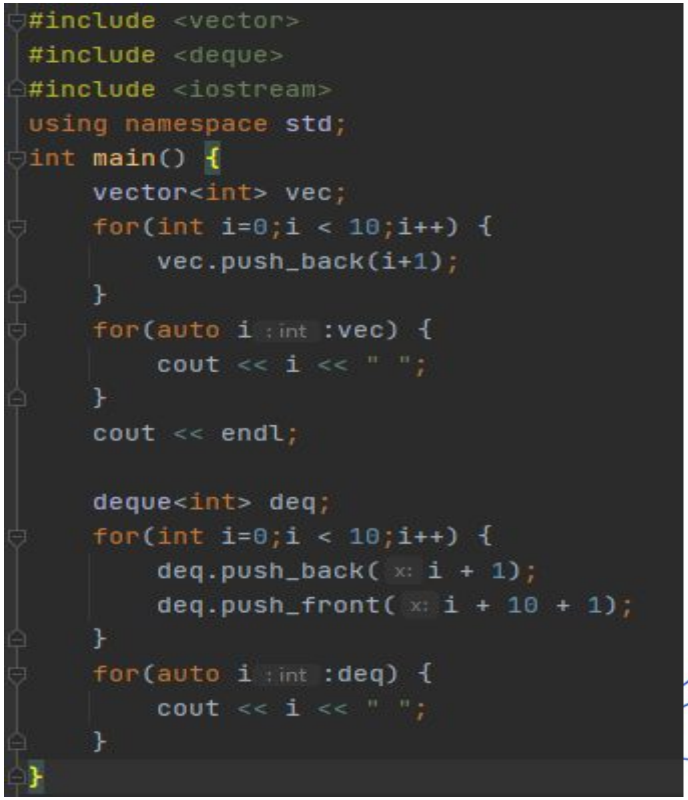
STL - Containers

- Container is a class designed to hold and organize multiple instances of another type.
- Commonly used containers are std::array, std::vector. Contrary to traditional arrays, these provide dynamic sizing, size checking etc, which makes them not just easier to use, but safer as well.

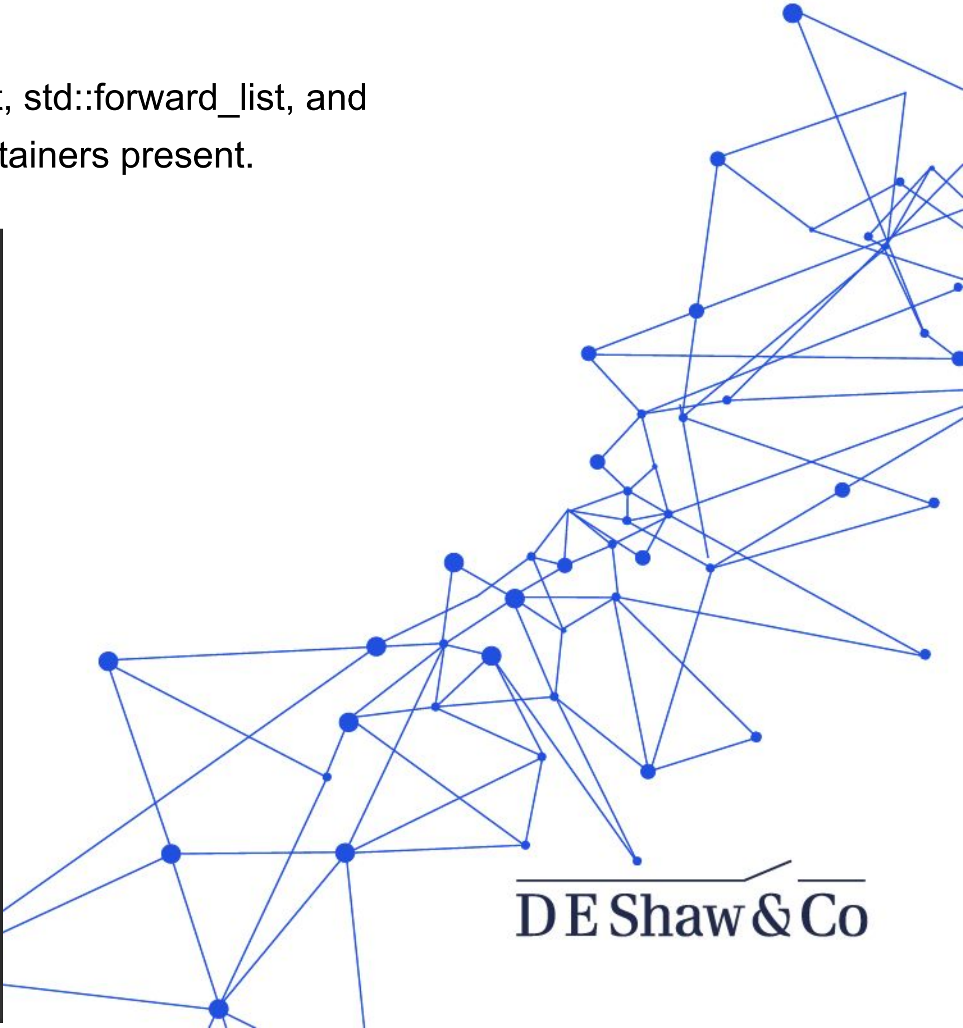


STL – Sequence Containers

- Container classes that store values in the order in which they are inserted into the container.
- std::vector, std::deque, std::array, std::list, std::forward_list, and std::basic_string are the 6 sequence containers present.

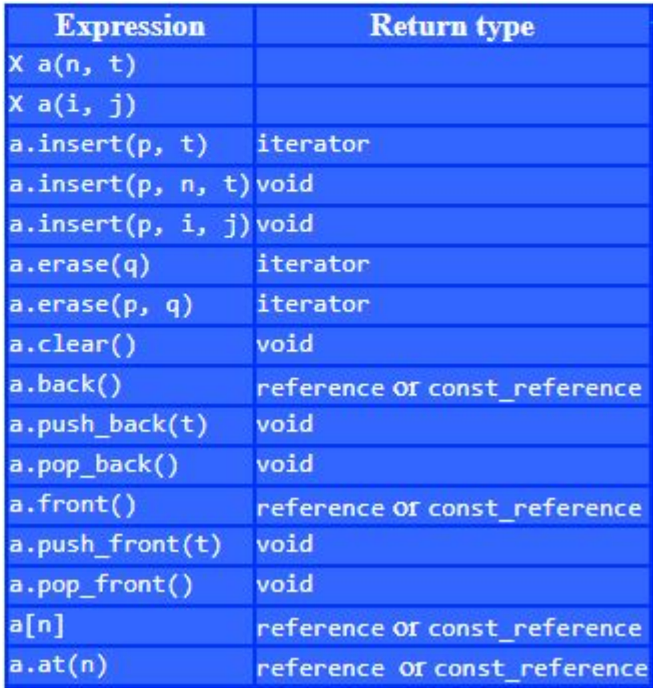


```
#include <vector>
#include <deque>
#include <iostream>
using namespace std;
int main() {
    vector<int> vec;
    for(int i=0;i < 10;i++) {
        vec.push_back(i+1);
    }
    for(auto i :int :vec) {
        cout << i << " ";
    }
    cout << endl;

    deque<int> deq;
    for(int i=0;i < 10;i++) {
        deq.push_back( x: i + 1);
        deq.push_front( x: i + 10 + 1);
    }
    for(auto i :int :deq) {
        cout << i << " ";
    }
}
```



STL – Sequence Containers

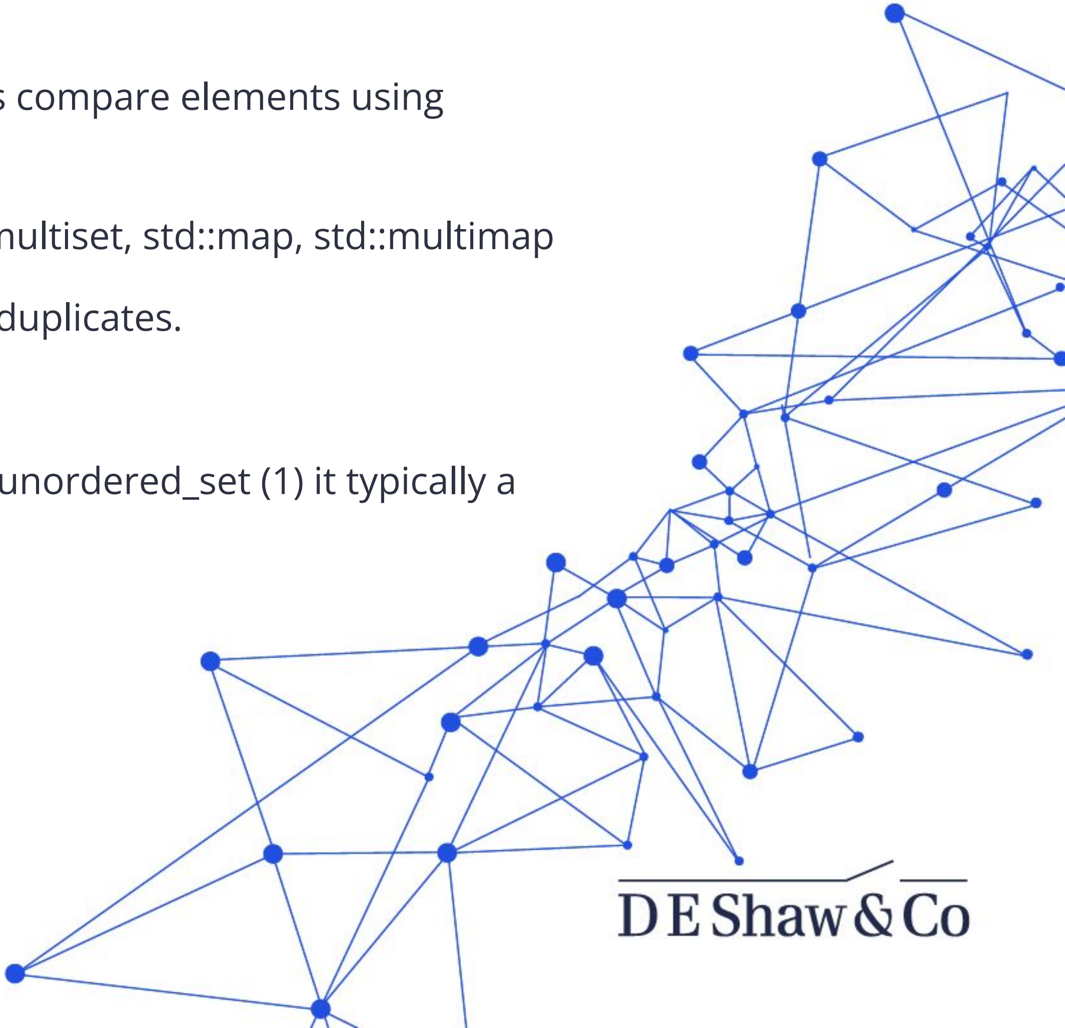
- General APIs for Sequence containers:



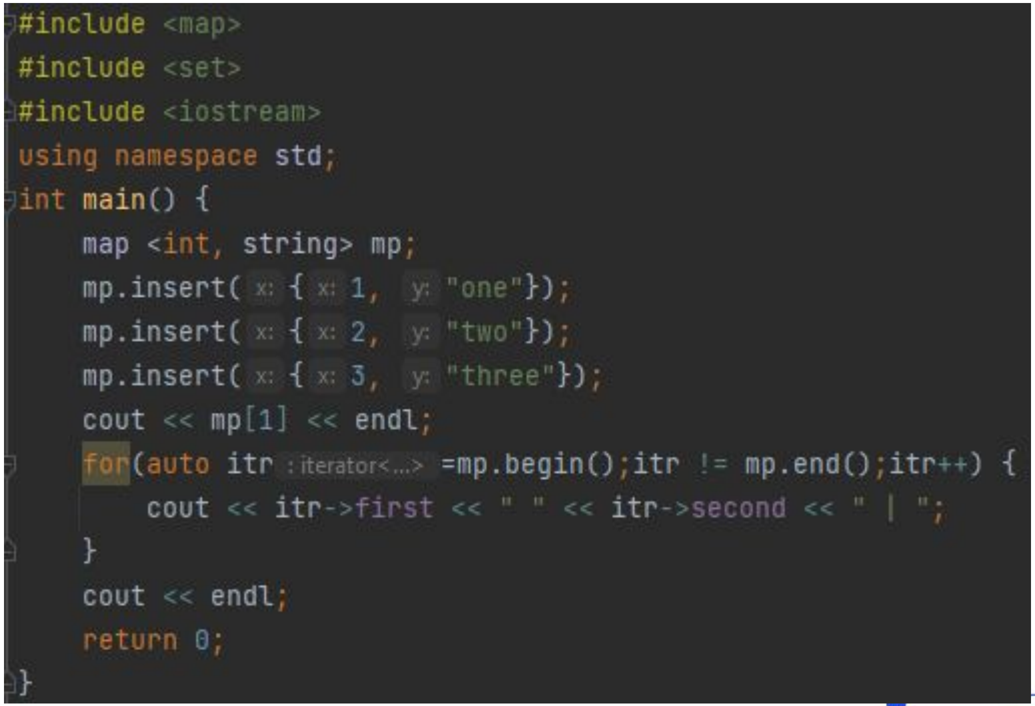
Expression	Return type
X a(n, t)	
X a(i, j)	
a.insert(p, t)	iterator
a.insert(p, n, t)	void
a.insert(p, i, j)	void
a.erase(q)	iterator
a.erase(p, q)	iterator
a.clear()	void
a.back()	reference or const_reference
a.push_back(t)	void
a.pop_back()	void
a.front()	reference or const_reference
a.push_front(t)	void
a.pop_front()	void
a[n]	reference or const_reference
a.at(n)	reference or const_reference

STL – Associative Containers

- Containers that automatically sort their inputs when those inputs are inserted into the container.
- By default, associative containers compare elements using operator <
- The containers are std::set, std::multiset, std::map, std::multimap
- multiset and multimap allow for duplicates.
- map is a <key, value> container.
- Set (log n) is a balanced BST and unordered_set (1) it typically a hashtable.

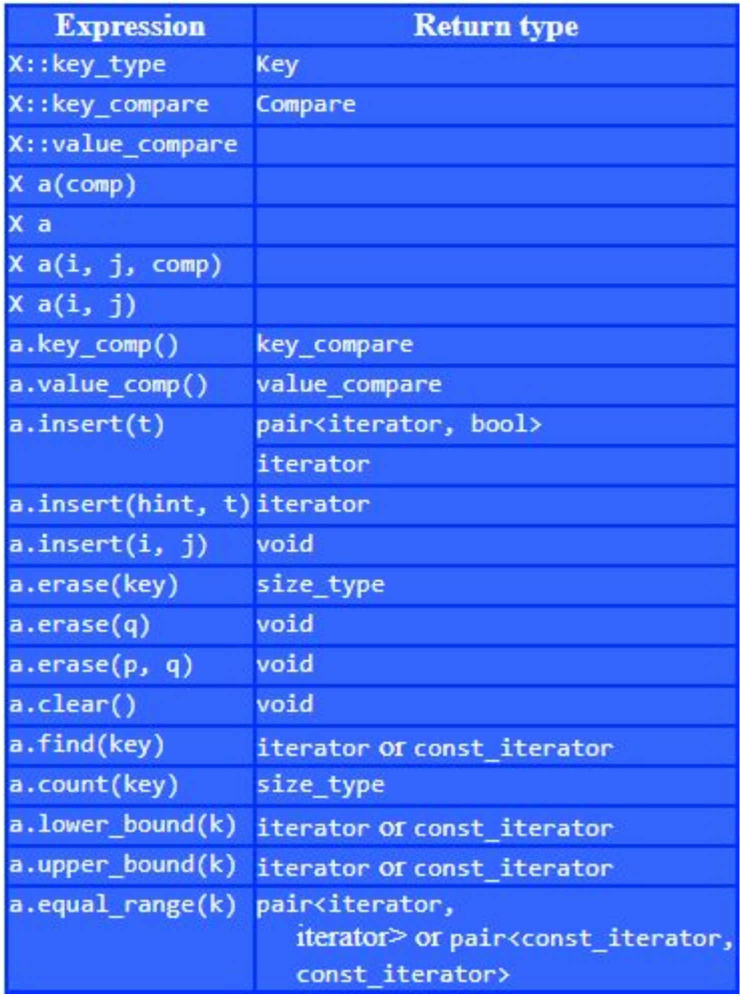


STL – Associative Containers



```
#include <map>
#include <set>
#include <iostream>
using namespace std;
int main() {
    map <int, string> mp;
    mp.insert( x: { x: 1,  y: "one"});
    mp.insert( x: { x: 2,  y: "two"});
    mp.insert( x: { x: 3,  y: "three"});
    cout << mp[1] << endl;
    for(auto itr :iterator<...> =mp.begin();itr != mp.end();itr++) {
        cout << itr->first << " " << itr->second << " | ";
    }
    cout << endl;
    return 0;
}
```

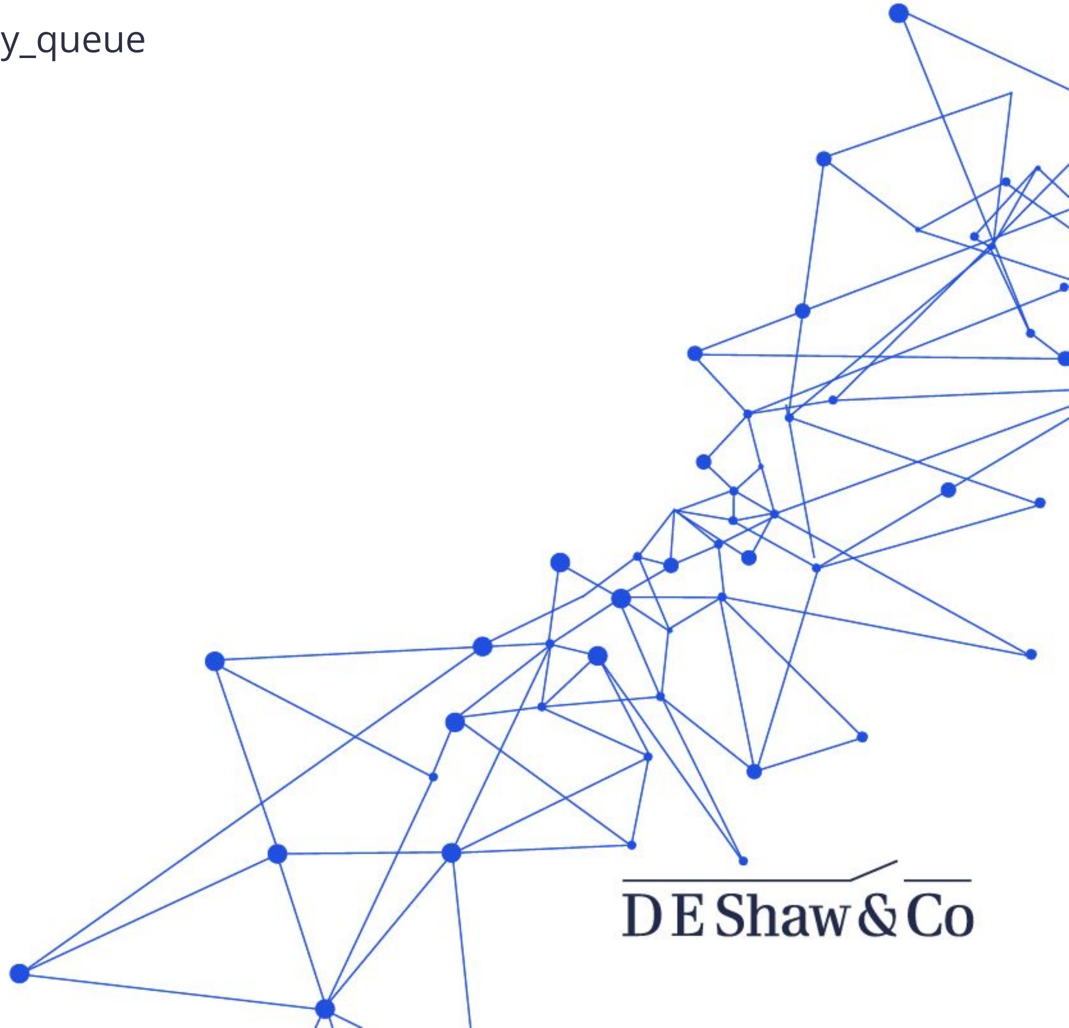
STL – Associative Containers



Expression	Return type
X::key_type	Key
X::key_compare	Compare
X::value_compare	
X a(comp)	
X a	
X a(i, j, comp)	
X a(i, j)	
a.key_comp()	key_compare
a.value_comp()	value_compare
a.insert(t)	pair<iterator, bool>
	iterator
a.insert(hint, t)	iterator
a.insert(i, j)	void
a.erase(key)	size_type
a.erase(q)	void
a.erase(p, q)	void
a.clear()	void
a.find(key)	iterator or const_iterator
a.count(key)	size_type
a.lower_bound(k)	iterator or const_iterator
a.upper_bound(k)	iterator or const_iterator
a.equal_range(k)	pair<iterator, iterator> or pair<const_iterator, const_iterator>

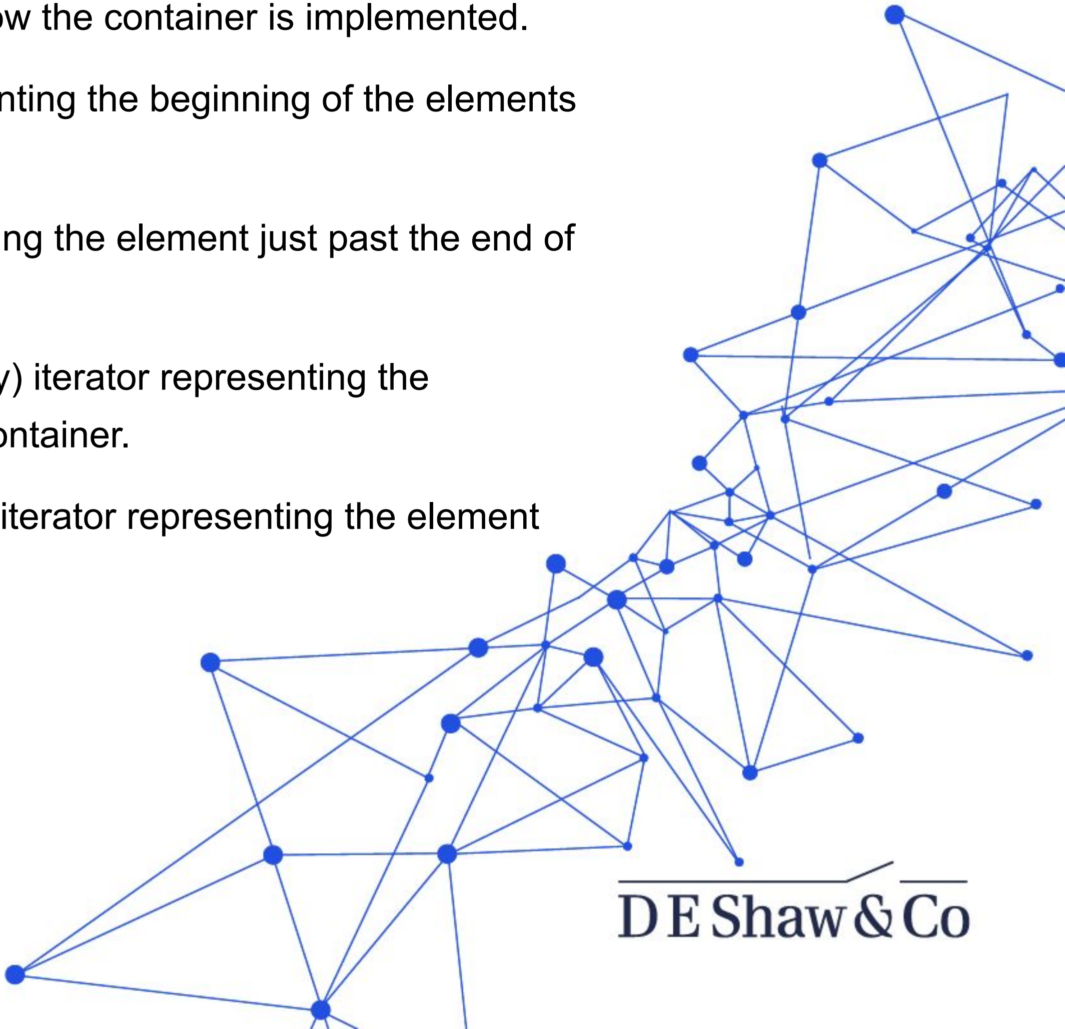
STL – Container Adapters

- Special predefined containers that are adapted to specific uses
- std::stack, std::queue, std::priority_queue



STL – Container Iterators

- Iterator is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented.
- **begin()** returns an iterator representing the beginning of the elements in the container.
- **end()** returns an iterator representing the element just past the end of the elements.
- **cbegin()** returns a const (read-only) iterator representing the beginning of the elements in the container.
- **cend()** returns a const (read-only) iterator representing the element just past the end of the elements.



STL – strings

- std::string. std::string (and std::wstring) is a string class that provides many operations to assign, compare, and modify strings.
- Creating a C-style string:
  char* str {new char[7]};
- Need to account extra character for null character.
- Copy the value:
  strcpy(str, “hello”);
- Buffer needs to be large enough. Need to deallocate the string as well:
  delete[] str;
- So much headache!

STL – strings

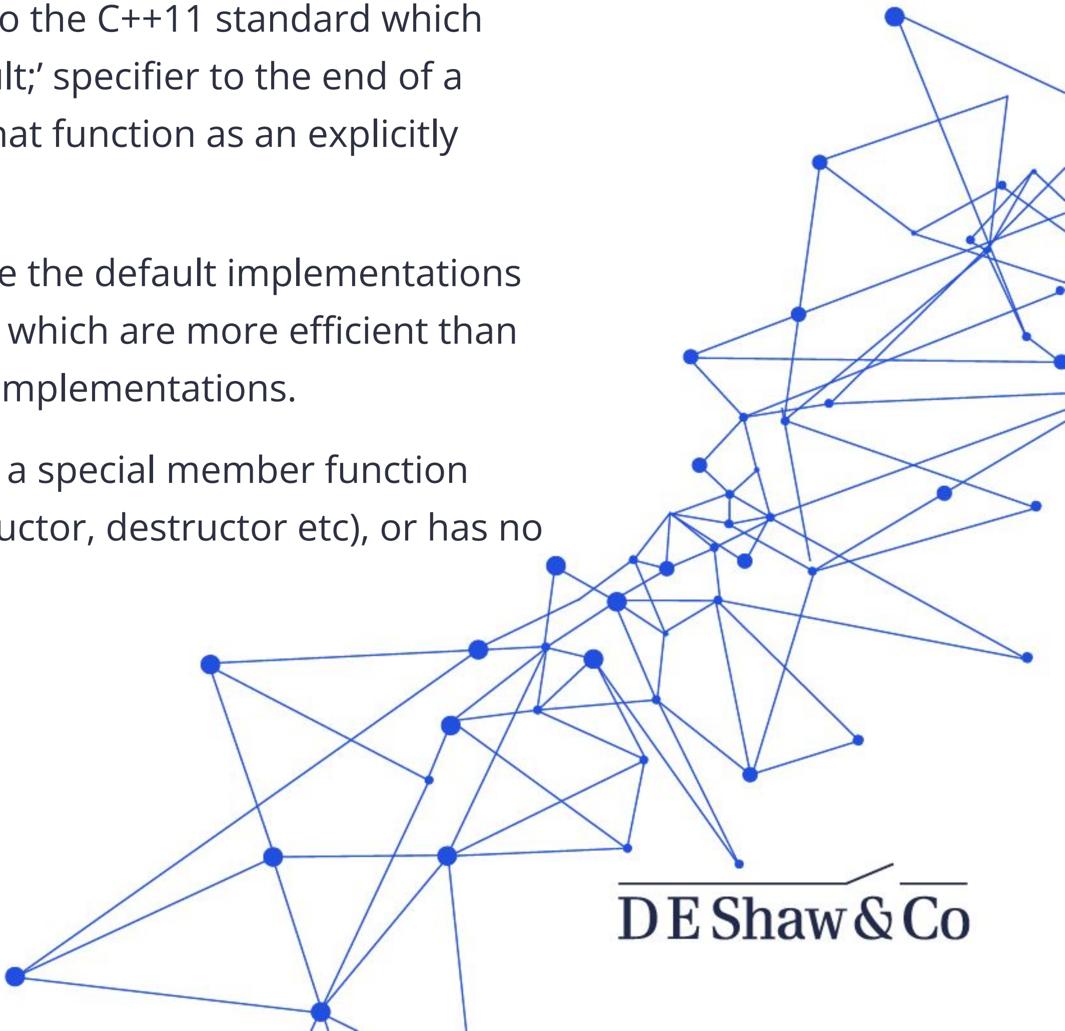
Ascend Educare

- Creation of std::string:
  std::string str{“hello”};
- Getting string length:
  str.length();
- Getting string capacity:
  str.capacity();
- Character access:
  str[1] // ’e’
- Convert to C-style string:
  str.c_str();

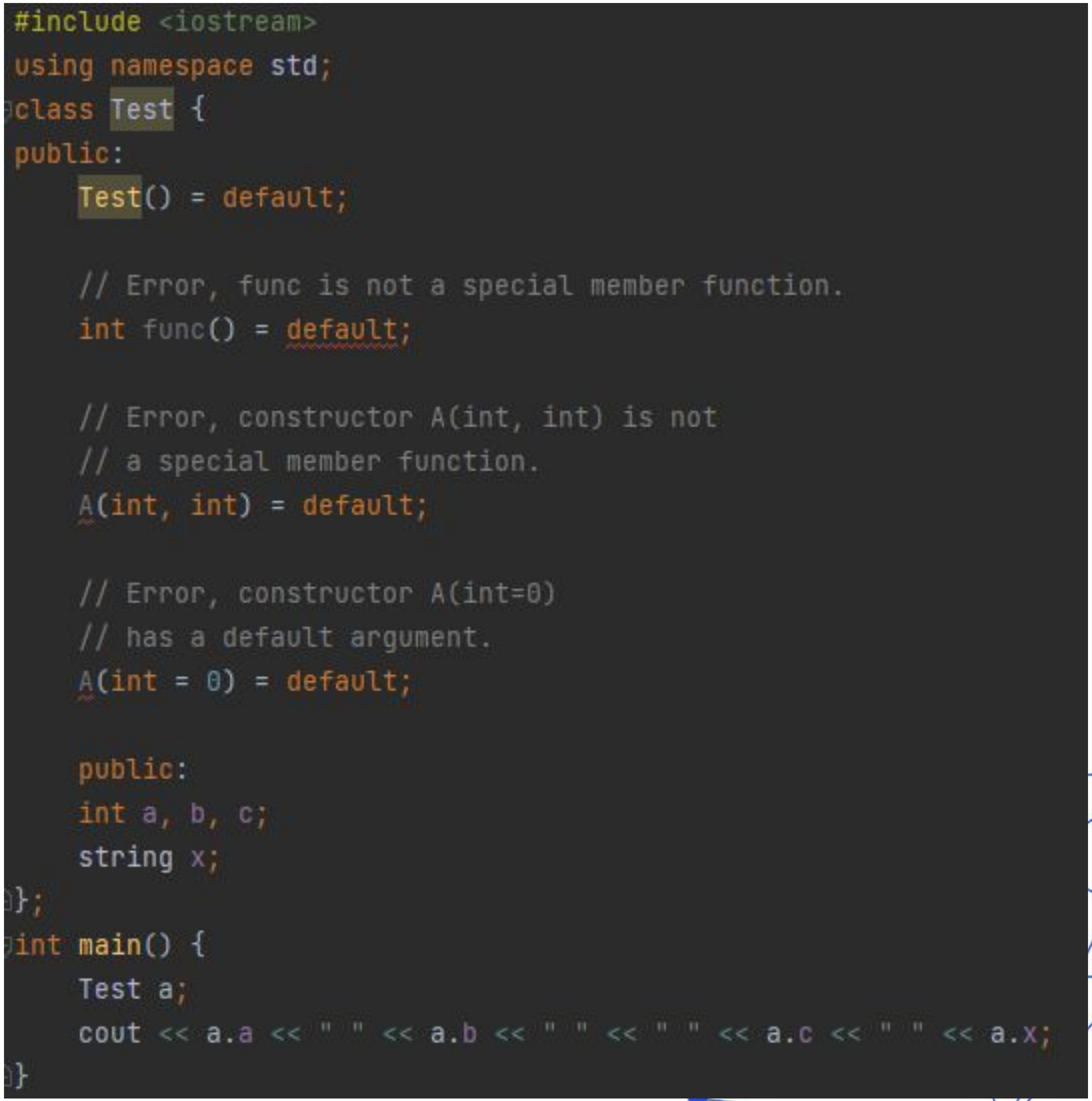
D E Shaw & Co

Default function

- Explicitly defaulted function declaration is a new form of function declaration that is introduced into the C++11 standard which allows you to append the ‘=default;’ specifier to the end of a function declaration to declare that function as an explicitly defaulted function.
- This makes the compiler generate the default implementations for explicitly defaulted functions, which are more efficient than manually programmed function implementations.
- A defaulted function needs to be a special member function (default constructor, copy constructor, destructor etc), or has no default arguments.



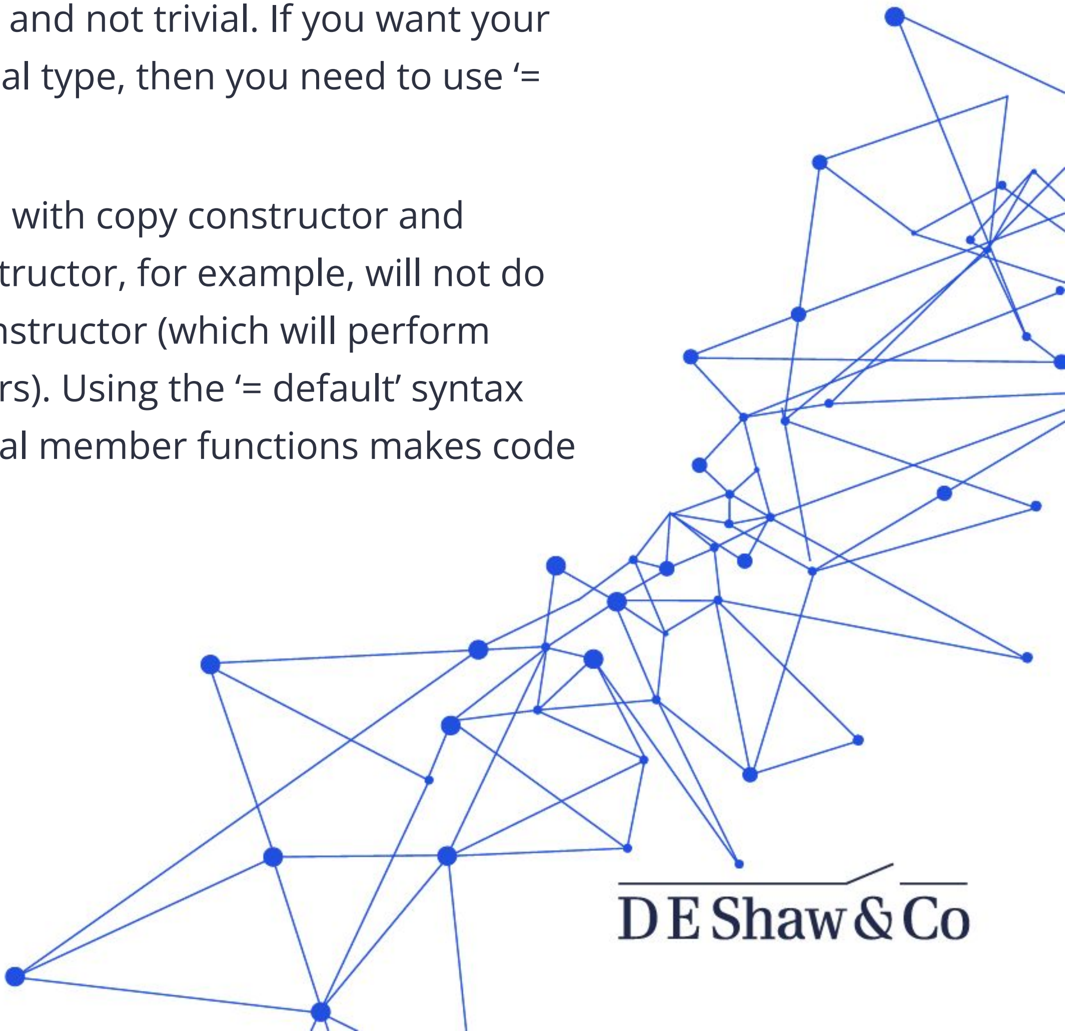
Default functions



```
#include <iostream>
using namespace std;
class Test {
public:
    Test() = default;

    // Error, func is not a special member function.
    int func() = default;

    // Error, constructor A(int, int) is not
    // a special member function.
    A(int, int) = default;

    // Error, constructor A(int=0)
    // has a default argument.
    A(int = 0) = default;

    public:
    int a, b, c;
    string x;
};
int main() {
    Test a;
    cout << a.a << " " << a.b << " " << " " << a.c << " " << a.x;
}
```

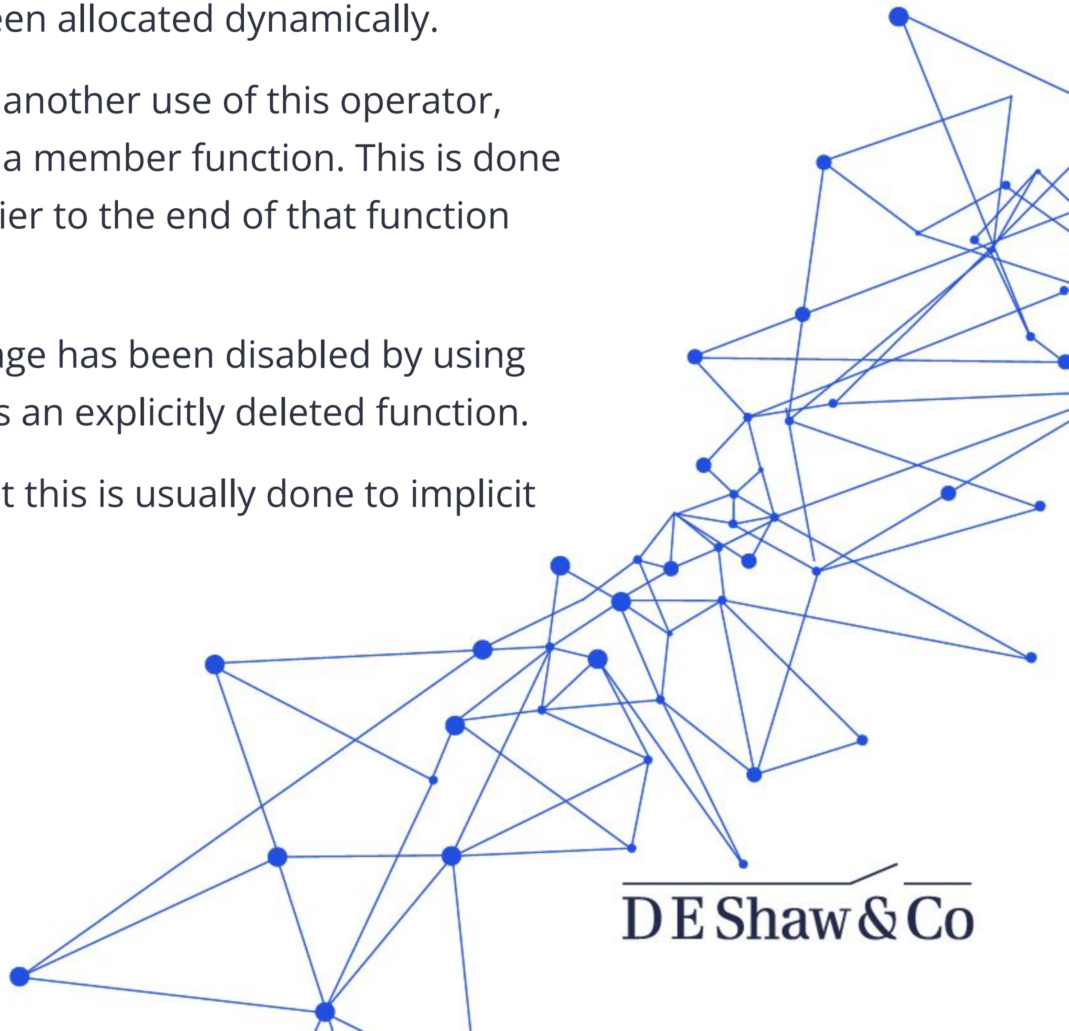
Default function

- Giving a user-defined constructor, even though it does nothing, makes the type not an aggregate and not trivial. If you want your class to be an aggregate or a trivial type, then you need to use '= default'.
- Using '= default' can also be used with copy constructor and destructors. An empty copy constructor, for example, will not do the same as a defaulted copy constructor (which will perform member-wise copy of its members). Using the '= default' syntax uniformly for each of these special member functions makes code easier to read.

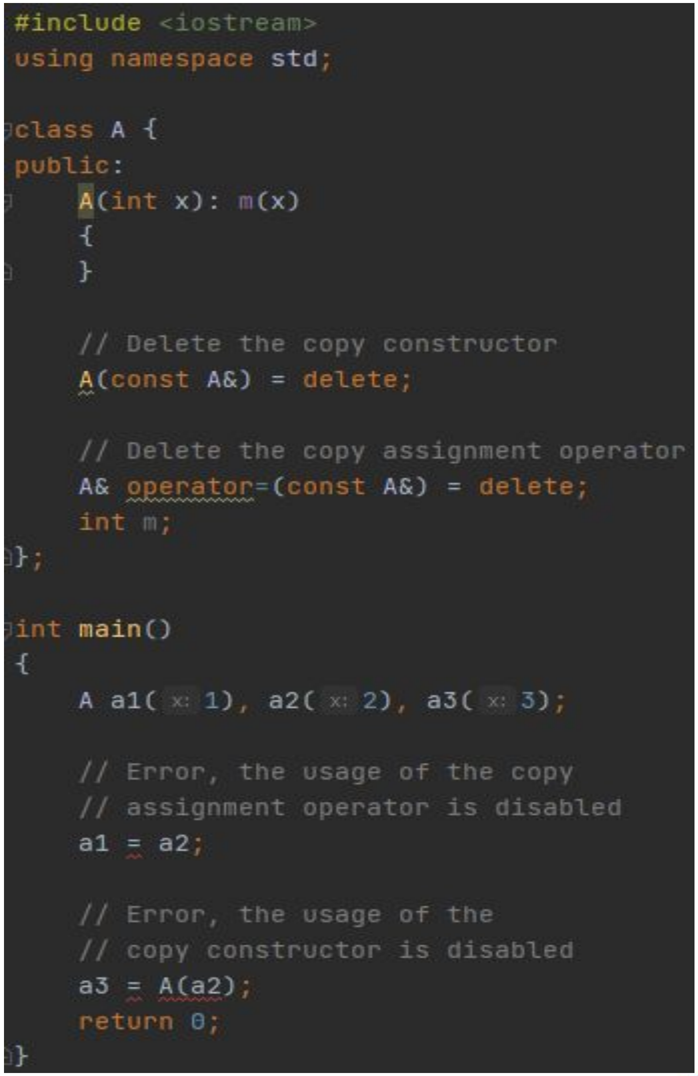


Deleted function

- Prior to C++ 11, the operator delete had only one purpose, to deallocate a memory that has been allocated dynamically.
- The C++ 11 standard introduced another use of this operator, which is: To disable the usage of a member function. This is done by appending the =delete; specifier to the end of that function declaration.
- Any member function whose usage has been disabled by using the ‘=delete’ specifier is known as an explicitly deleted function.
- Although not limited to them, but this is usually done to implicit functions



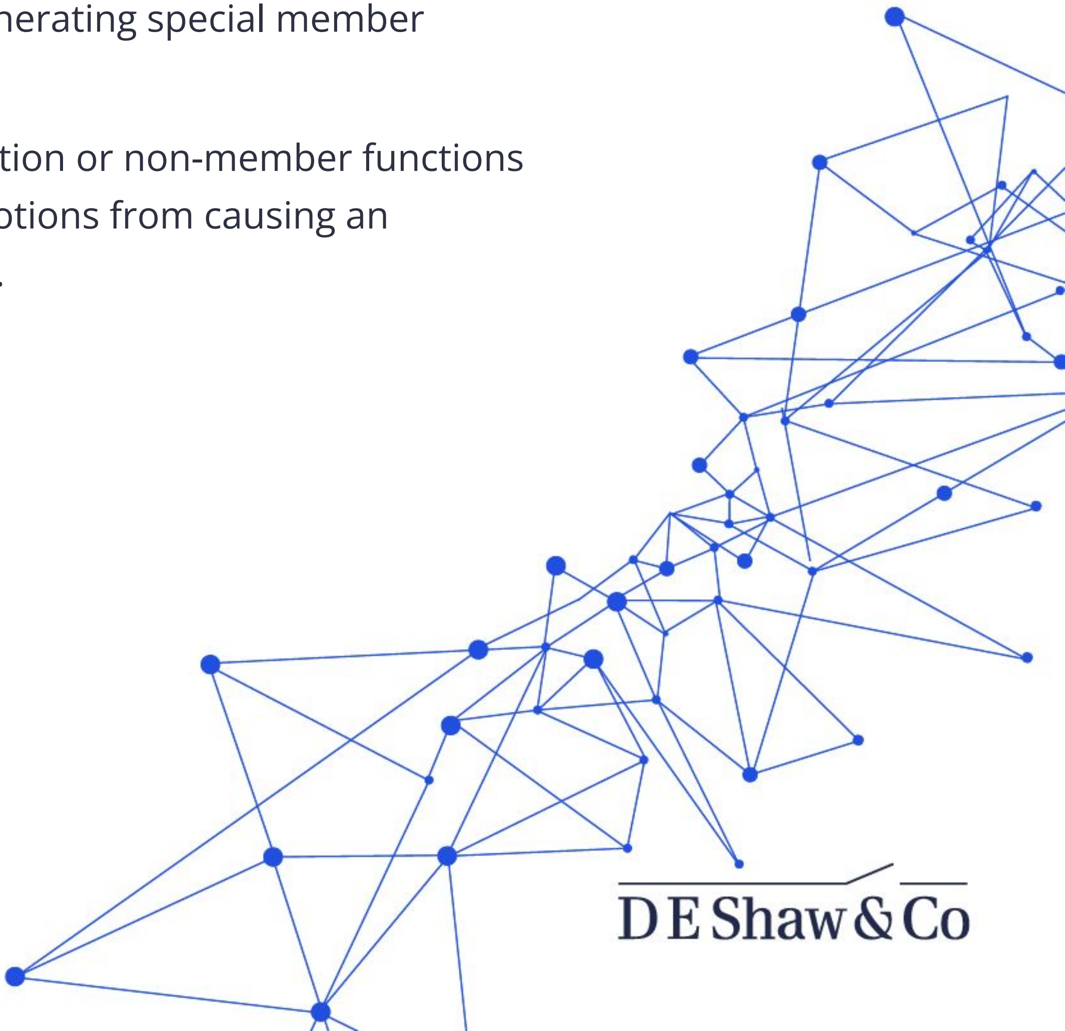
Deleted function



```
#include <iostream>
using namespace std;

class A {
public:
    A(int x): m(x)
    {
    }

    // Delete the copy constructor
    A(const A&) = delete;

    // Delete the copy assignment operator
    A& operator=(const A&) = delete;
    int m;
};

int main()
{
    A a1( x: 1), a2( x: 2), a3( x: 3);

    // Error, the usage of the copy
    // assignment operator is disabled
    a1 = a2;

    // Error, the usage of the
    // copy constructor is disabled
    a3 = A(a2);
    return 0;
}
```

Deleted function

- Deleting of special member functions provides a cleaner way of preventing the compiler from generating special member functions that we don’t want.
- Deleting of normal member function or non-member functions prevents problematic type promotions from causing an unintended function to be called.

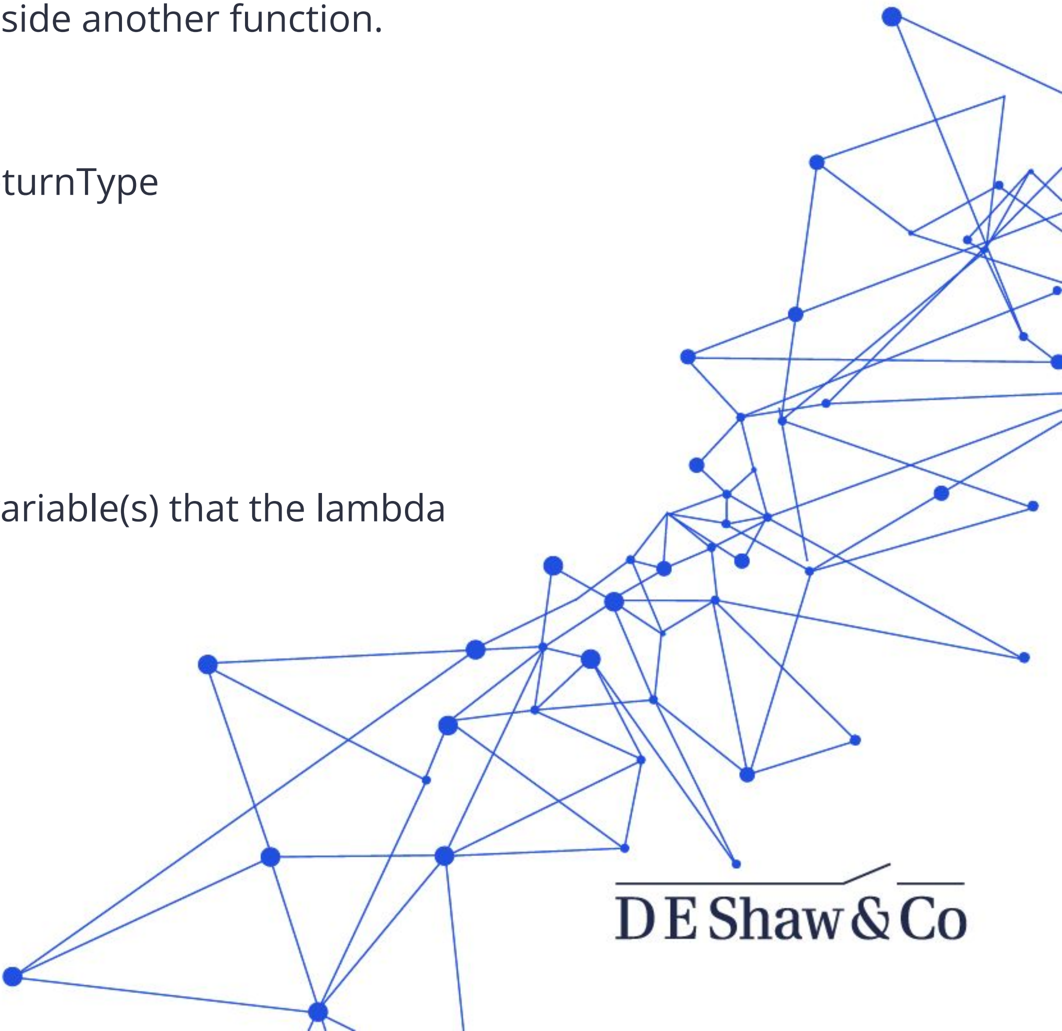


Lambda function

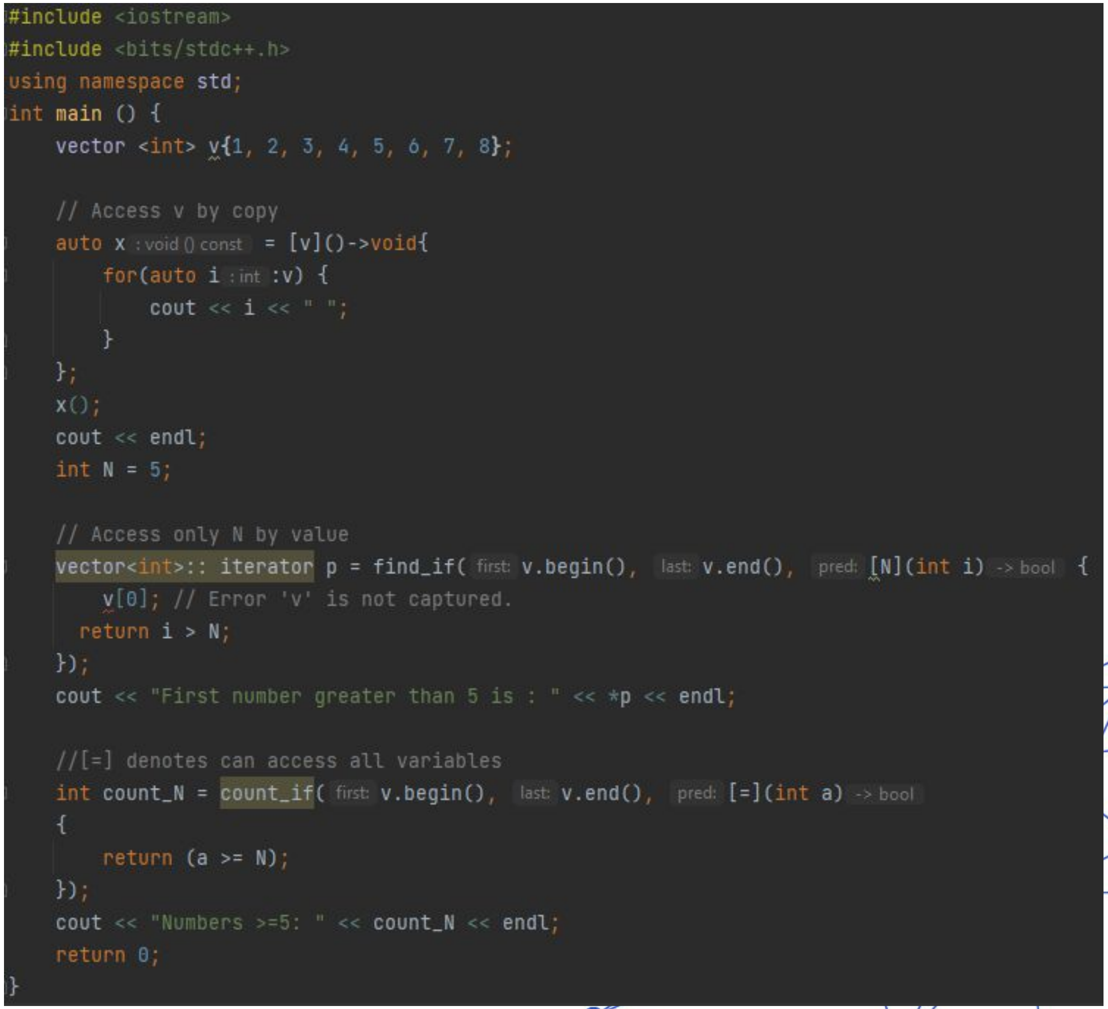
- A lambda expression (also called a lambda or closure) allows us to define an anonymous function inside another function.
- Syntax:

```
[ captureClause ] ( parameters ) -> returnType

{

   statements;

}
```

- The capture clause denotes the variable(s) that the lambda function can access.

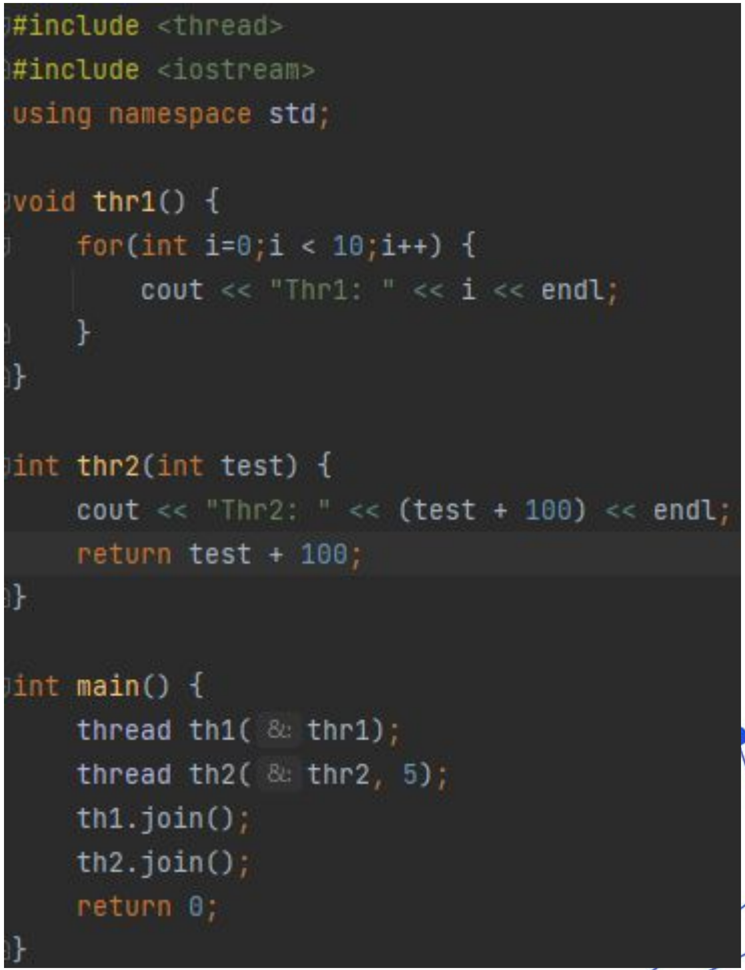


Lambda function



```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
int main () {
    vector <int> v{1, 2, 3, 4, 5, 6, 7, 8};

    // Access v by copy
    auto x : void () const  = [v]()->void{
        for(auto i : int :v) {
            cout << i << " ";
        }
    };
    x();
    cout << endl;
    int N = 5;

    // Access only N by value
    vector<int>:: iterator p = find_if( first: v.begin(),  last: v.end(),  pred: [N](int i) -> bool  {
        v[0]; // Error 'v' is not captured.
      return i > N;
    });
    cout << "First number greater than 5 is : " << *p << endl;

    //[=] denotes can access all variables
    int count_N = count_if( first: v.begin(),  last: v.end(),  pred: [=](int a) -> bool
    {
        return (a >= N);
    });
    cout << "Numbers >=5: " << count_N << endl;
    return 0;
}
```

Threading library



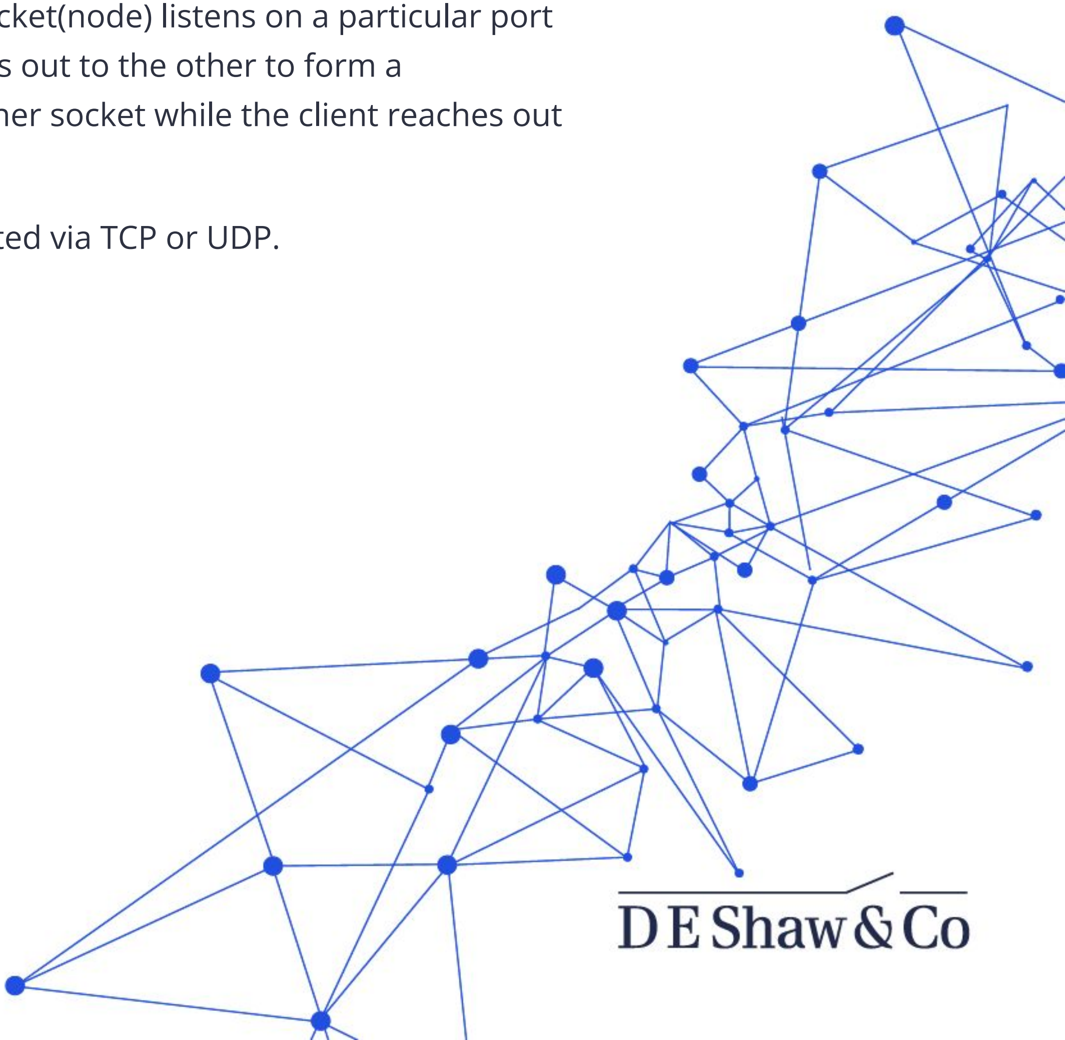
```
#include <thread>
#include <iostream>
using namespace std;

void thr1() {
    for(int i=0;i < 10;i++) {
        cout << "Thr1: " << i << endl;
    }
}

int thr2(int test) {
    cout << "Thr2: " << (test + 100) << endl;
    return test + 100;
}

int main() {
    thread th1( &: thr1);
    thread th2( &: thr2, 5);
    th1.join();
    th2.join();
    return 0;
}
```

Concurrency concepts

- Concurrency is when two tasks are overlapped. A simple concurrent application will use a single machine to store the program's instruction, but that process is executed by multiple, different threads.
- Data races are a common problem in this kind of a setup.
- Concurrency and parallelism often get mixed up, but it's important to understand the difference. In parallelism, we run multiple copies of the same program simultaneously, but they are executed on different data.
- For example, you could use parallelism to send requests to different websites but give each copy of the program a different set of URLs. These copies are not necessarily in communication with each other, but they are running at the same time in parallel.
- As we explained above, concurrent programming involves a shared memory location, and the different threads actually "read" the information provided by the previous threads.

Socket Programming

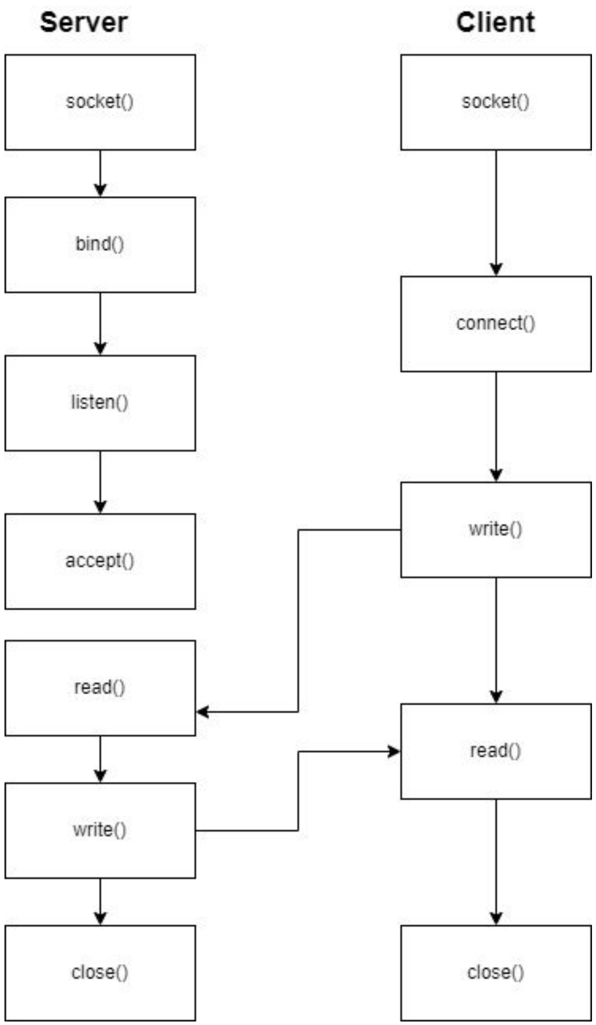
- Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.
- The clients and servers can be connected via TCP or UDP.



Socket Programming – TCP vs UDP

TCP	UDP
Secure	Unsecure
Connection Oriented	Connectionless
Slow	Fast
Guaranteed Transmission	No Guarantee
Used by critical applications	Used by real-time applications
Packet reoder mechanism	No reorder mechanism
Flow control	No flow control
Advanced error checking	Basic error checking (via checksums)
20 bytes header	8 bytes header
ACK mechanism	No ACK
Three-way handshake	No handshake
DNS, HTTP, HTTPS, FTP etc.	DNS, DHCP, TFTP, SNMP etc.

Socket Programming



Socket Programming

Code reference:
https://www.geeksforgeeks.org/socket-programming-cc/#:~:text=Socket%20programming%20is%20a%20way,reaches%20out%20to%20the%20server.