



# CSCI251 Advanced Programming

Generic Programming V: The Standard Template Library (STL)  
Continue...



UNIVERSITY  
OF WOLLONGONG  
AUSTRALIA

# STL

- STL contains six major kinds of building blocks, implemented using templates:
  - Containers.
  - Iterators.
  - Generic algorithms.
  - Function objects.
  - Adaptors.
  - Allocators.



# Generic algorithms

- Perform common operation:
  - Finding, sorting, searching,...
- Operation is independent of the data type
- Algorithms are applied to containers through the iterators
  - they operate through the iterators and they are container independent.
  - Note that: they usually don't add NEW elements to the original sequence



# Generic algorithms

```
#include <algorithm>
```

- Most of the algorithms take one of the following four forms:

```
alg(beg, end, other args);  
alg(beg, end, dest, other args);  
alg(beg, end, beg2, other args);  
alg(beg, end, beg2, end2, other args);
```

- The arguments *dest*, *beg2*, and *end2*, are used to reference a destination, or a second input sequence.



# Algorithm categories

- Non-modifying sequence operations:
  - e.g. `find`.
- Modifying sequence operations:
  - E.g. `copy`.
- Partitioning operations
- Sorting operations
- Binary search operations (on sorted ranges)
- Set operations (on sorted ranges)
- Heap operations
- Minimum/maximum operations
- Numeric operations

# Complexity

- The details on the algorithms at <http://en.cppreference.com/w/cpp/algorithm> indicate their complexity.
- For example:
  - `next_permutation( see next )`
  - Complexity: At most  $N/2$  swaps, where  $N = \text{std::distance}(\text{first}, \text{last})$ .



# next\_permutation

```
#include <algorithm>
#include <string>
#include <iostream>

int main()
{
    std::string s = "abcd";
    std::sort(s.begin(), s.end());
    do {
        std::cout << s << '\n';
    } while(next_permutation(s.begin(), s.end()));
}
```



# Sort

- Sorts the elements in the range [first, last) in non-descending order
- Complexity:  $O(N \cdot \log(N))$
- *Example:*

```
#include <algorithm>
#include <iostream>
#include <array>
using namespace std;
int main()
{
    std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
    std::sort(s.begin(), s.end());
    for (auto a = s.begin(); a!=s.end(); a++)
        cout<<*a<<endl;
}
```





# Finding

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

int main()
{
    int myInts[] = { 10, 20, 30, 40 };

    vector<int> myVector (myInts,myInts+4); // vector of above ints
    vector<int>::iterator it; // vector iterator

    it = find(myVector.begin(), myVector.end(), 30);

    if (it != myVector.end())
        cout << "Element found in myVector: " << *it << '\n';
    else
        cout << "Element not found in myVector\n";

    return 0;
}
```

if (map.find(key) != map.end())



# Function objects

- Classes that have an overloaded `operator()`
- In the context of STL there are some specific templated function objects classes that are used.
- There is a list at:  
<http://en.cppreference.com/w/cpp/utility/functional>



# Function objects

- We can, for example, take a container (say a vector), and apply operation using its iterator and a function object ( ).
- The function object tells you how to do the individual operation for each iterator.



# Function objects

```
#include <vector>
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

void PrintFun(int n)
{
    cout << n << ' ';
}

class PrintObj
{
public:
    void operator()(int n)
    {
        cout << n << ' ';
    }
};
```

```
void Add3(int &n)
{
    n += 3;
}

class AddObj
{
public:
    AddObj(int number) : number_(number)
    {

    }

    void operator()(int &n)
    {
        n += number_;
    }

private:
    int number_;
};
```



# Function objects

```
class GreaterObj
{
public:
    GreaterObj(int number) : number_(number)
    {

    }
    bool operator()(int n)
    {
        return n > number_;
    }
private:
    int number_;
};
```



# Function objects

```
int main(void)
{
    int a[] = {1, 2, 3, 4, 5};
    vector<int> v(a, a + 5);

    //for_each(v.begin(), v.end(), PrintFun);

    for_each(v.begin(), v.end(), PrintObj());

    /*for_each(v.begin(), v.end(), Add3);
    for_each(v.begin(), v.end(), PrintFun);
    cout<<endl;*/

    for_each(v.begin(), v.end(), AddObj(5));
    for_each(v.begin(), v.end(), PrintFun);

    cout << count_if(a, a + 5, GreaterObj(3)) << endl;
    return 0;
}
```



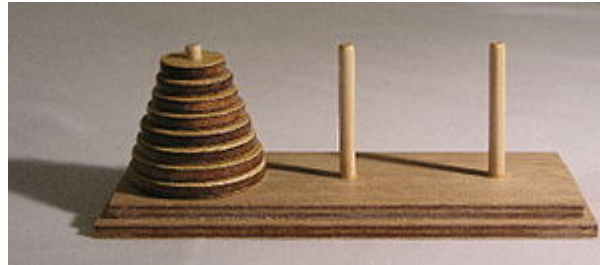
# Adaptors

- an adaptor is a mechanism for making one thing act like another
- There are three sequential container adaptors:
  - Stack: LIFO.
  - Queue: FIFO.
  - Priority\_queue: Prioritises based on, by default, <.



# Adaptors

- As an example; the stack adaptor turns a sequential container, other than `array` or `forward_list`, and makes it operate like a stack.
- Tower of Hanoi - Stack





# Allocators

- The library class `allocator` is used to allocate unconstructed memory.
- main purposes: decouple memory allocation from object construction.
  - we can allocate memory in large chunks and pay the overhead of constructing the objects only when we actually need to create them.



# Allocators vs new

- The use of `new` is constrained in that it combines allocating memory with constructing an object, or objects, in that memory.
- an allocator allows these two operations to be separated.
- You only use/destroy what you construct.



# Allocators

- **Vector:** the use of `pop` removes an element but doesn't free the associated memory.
- **We can use the following member functions:**
  - **Size:** Returns the number of elements in the vector.
    - `myints.size()`
  - **Capacity:** Returns the size of the storage space currently allocated for the vector



# Example (dynamic change)

```
#include <iostream>
#include <vector>

int main ()
{
    //std::vector<int>::size_type sz;

    std::vector<int> myvec;
    auto sz = myvec.capacity();
    std::cout << "making myvec grow:\n";
    for (int i=0; i<100; ++i) {
        myvec.push_back(i);
        std::cout<<myvec.size();
        if (sz!=myvec.capacity()) {
            sz = myvec.capacity();
            std::cout << "capacity changed: " << sz << '\n';
        }
    }
}
```



# Allocators

- Instances of the class allocator can be use to provide type-aware allocation of raw, unconstructed, memory.
- We defined an allocator object for objects of *type T* as follows:  

```
allocator<T> a;
```
- The operations for allocation, deallocation, creation and destruction are paired



# Allocators

```
auto const p =  
a.allocate(n);
```

Allocates raw, unconstructed memory to hold  $n$  objects of type  $T$ , and sets up  $p$  to point to it.

```
a.construct(p, args);
```

Calls the constructor for objects of type  $T$ , associated with the  $T^*$  pointer  $p$ .

```
a.destroy(p);
```

Calls the destructor on the object pointed to by the  $T^*$  pointer  $p$ .

```
a.deallocate(p, n);
```

Deallocates the memory of  $n$  type  $T$  objects pointed to by  $p$ .



# Allocators

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
int main()
{
    allocator<string> alloc;
    auto const p = alloc.allocate(5);
    auto q=p;
    alloc.construct(q++);
    alloc.construct(q++, 10, 'c');
    alloc.construct(q++, "hi");
    auto r=p;
    do
    cout << *r << endl;
    while (++r != q);
    Cout<<"done"<<endl;
    while (q != p)
    alloc.destroy(--q);
```

```
q=p; r=p;
alloc.construct(q++, 10, 'a');
alloc.construct(q++, "hi again");
do
cout << "here" << *r << endl;
while (++r != q);

alloc.deallocate(p, 5);
}
```

