

# **The Standard C++ Library**

Version 1: Dr. Ofir Pele

Version 2: Dr. Erel Segal-Halevi

**Q: What types can we put  
in a template param?**

**A: Type which models the  
requested concept.**

# Concepts - Example

- Consider:

```
template<typename T> const T& min(const T& x,  
                                const T& y)  
{  
    return x < y? x : y;  
}
```

- The user must provide a type that has a less-than operator (<) to compare two values of type T, the result of which must be convertible to a Boolean value

# Concepts - Example

- Consider:

```
template<typename T> const T& min(const T& x,  
                                const T& y)  
{  
    return x < y? x : y;  
}
```

- The problem:  
cryptic error messages  
from the implementation of the function  
instead of a clear error message

# Concepts – What we would like it to be:

- **Not C++ syntax:**

```
template<LessThanComparable T> const T& min(const T& x,  
                                             const T& y)  
{  
    return x < y? x : y;  
}
```

- The user must provide a type that has a less-than operator (<) to compare two values of type T, the result of which must be convertible to a Boolean value

# Concepts

- Concepts are not a yet part of the C++ language,
- Currently there is no (standard) way to declare a concept in a program, or to declare that a particular type is a model of a concept
- “were not ready” for C++11,C++14,C++17
- C++-20 ?

# Concepts

- A concept is a list of requirements on a type.
- STL defines a hierarchy of concepts for containers, iterators, and element types.
- Concepts for element types include:

**Equality Comparable** -  
types with operator== ,...

**LessThan Comparable** -  
types with operator< ,...

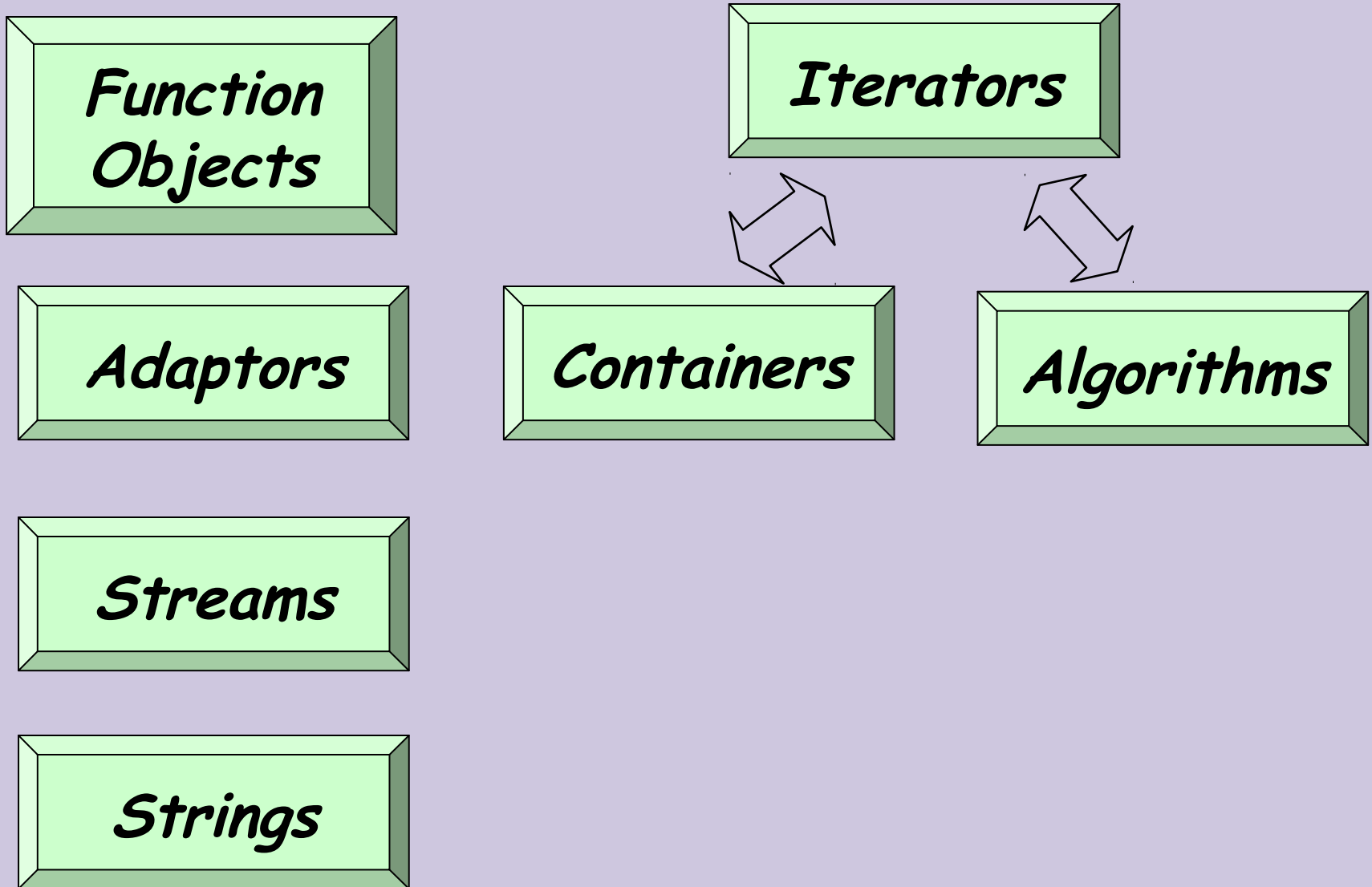
**Assignable** -  
types with operator=  
and copy Ctor

# Concepts

- Cleanly separate interface from implementation.
- Primitives can also conform to a concept.



# Main Components



# *Containers*

- Holds **copies** of elements.
- **Assumes** elements have:  
Copy Ctor & operator =
- The standard defines the **interface**.
- Two main classes
  - **Sequential containers:**  
list, vector, ....
  - **Associative containers:**  
map, set ...

**Assignable -**  
types with operator=  
and copy Ctor

## *Containers documentation*

see

<http://www.cplusplus.com/reference/stl/>

# Sequential Containers

-

# Sequential Containers

Maintain a linear sequence of objects.

**forward\_list** - a singly-linked list.

**list** - a doubly-linked list.

- Efficient insertion/deletion in front/end/middle

**vector** - an extendable sequence of objects

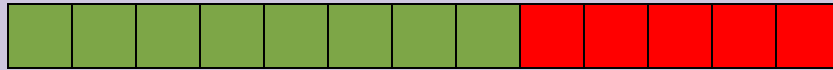
- Efficient insertion at end, and random access

**deque** – double-ended queue

- Efficient insertion/deletion at front/end
- Random access

**array** – fixed size, on the stack.

# vector<T>



- Contiguous array of elements of type T
- Random access
- Can grow on as needed basis

```
std::vector<int> v(2);  
v[0]= 45;  
v[1]= 32;  
v.emplace_back(60); //C++11
```

# emplace\_back / push\_back

## Average Time Complexity

If we inserted  $n$  elements we paid:

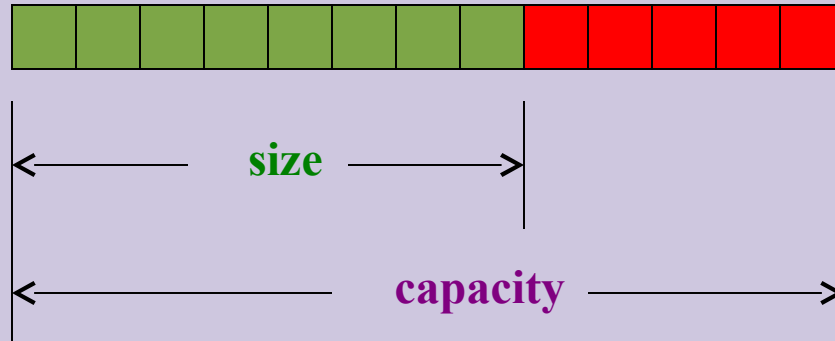
$$1+2+1+4+1+1+1+8+\dots+n =$$

$$O(n) + 1+2+4+\dots+n =$$

$$O(n)$$

On average an each insertion cost  $O(1)$

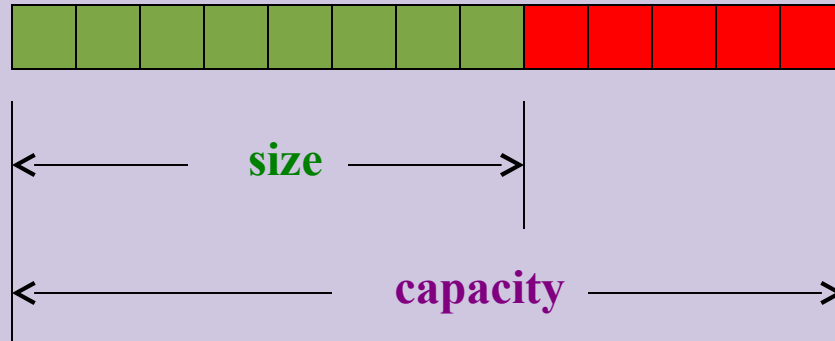
# size and capacity



- The first “size” elements are constructed (initialized)
- The last “capacity - size” elements are uninitialized



# size and capacity



- `size_type` **size**() `const`
- `size_type` **capacity**() `const`

# C++ vs. Java

- Look at cplusplus documentation of vector.
- Look at Java documentation of Vector.
- Differences:
  - Simple class vs. interface and vtable.
  - Simple elements vs. class element.
  - Two accessors (with and without range check) vs. a single accessor

# Creating vectors

- Empty vector:

```
std::vector<int> vec;
```

- vector with 10 ints each one of value `int()==0`:

```
std::vector<int> vec(10);
```

```
std::vector<int> vec(10,0); // better
```

- vector with 10 ints with the value of 42:

```
std::vector<int> vec(10, 42);
```

# Creating vectors

- Empty vector:

```
std::vector<int> vec;
```

- vector with 10 ints each one of value **int()==0**:

```
std::vector<int> vec(10);
```

```
std::vector<int> vec(10,0); // better
```

**Notice: int() is NOT a default constructor of int. Uninitialized ints (int k;) contain junk.**

# Creating vectors

- Empty vector:

```
std::vector<Fred> vec;
```

- vector with 10 default constructed Fred objects. i.e: 10 Fred() objects:

```
std::vector<Fred> vec(10);
```

- vector with 10 non-default constructed Fred objects:

```
std::vector<Fred> vec(10, Fred(5,7));
```

# Creating vectors: C++11

- vector with different ints inside it:

```
std::vector<int> vec{1, 5, 10, -2, 0, 3};
```

- vector with different Fred objects:

```
std::vector<Fred> vec{Fred(5,7), Fred(2,1) }
```

Or

```
std::vector<Fred> vec{ {5,7}, {2,1} }
```

# Associated types in vector

`vector<typename T>::`

- `value_type` - The type of object, T, stored
- `reference` - Reference to T
- `const_reference` - const Reference to T
- `iterator` - Iterator used to iterate through a vector (*how would you write it?*)
- ...

# Time complexity

- Random access to elements.
- Amortized constant time insertion and removal of elements at the end.
- Linear time insertion and removal of elements at the beginning or in the middle.
- vector is the simplest of the STL container classes, and in many cases the most efficient.



# Adding elements

- Inserts a new element at the end:

`void push_back(const T&)`

- `a.push_back(t)` } amortized constant time

# Adding elements-C++11

- Construct and insert a new element at the end:

```
template<typename... Args>
```

```
void emplace_back(Args&&... args)
```

- **a.emplace\_back(t)** } amortized constant time

# Accessing elements

## Without boundary checking:

- `reference operator[](size_type n)`
- `const_reference operator[](size_type n) const`

## With boundary checking:

- `reference at(size_type n)`
- `const_reference at(size_type n) const`

# What about checking boundaries only in DEBUG mode? - Linux

- g++ has a standard library in DEBUG mode, to activate it define `_GLIBCXX_DEBUG` (g++ -D\_GLIBCXX\_DEBUG ...)
- stlport is an implementation of the standard library which includes DEBUG mode (havn't checked it myself yet):

<http://www.stlport.org/>

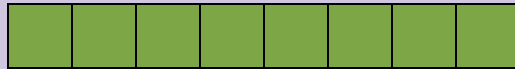
# What about checking boundaries only in DEBUG mode? - MS

- In MSVS 2012 Debug build is automatically safe and Release build mode is not
- Other versions also have these kind of “Safe STL” modes but might require defining some flags to turn off or on.

**vector<T> v**



**v.shrink\_to\_fit() // c++11**



# Associative Containers

- Supports efficient retrieval of elements (values) based on keys.
- (Typical) Implementation:  
red-black binary trees  
hash-table (added in c++11)

# Sorted Associative Containers

## **set**

- A set of unique keys ordered by <

## **map**

- Associate a value to key (associative array)
- Unique value of each key, ordered by <

## **multiset, multimap**

- Same, but allow multiple values

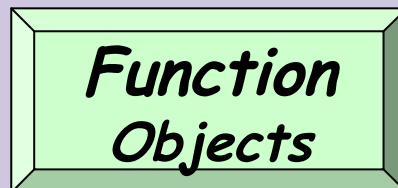
## **unordered\_set, unordered\_map**

- Same, but without order (faster).



# Sorted Associative Containers & Order

- Sorted Associative containers use operator< as default order
- We can control order by using our own comparison function
- To understand this issue, we need to use **function object**



# *Function Objects*

Anything that can be called as if a function.

For example:

- Pointer to function
- A class that implements `operator()`
- Lambda expressions (c++11)

## Example (folder 2)

```
class c_str_less {  
public:  
    bool operator() (const char* s1,  
                     const char* s2) {  
        return (strcmp(s1,s2) < 0);  
    }  
};
```

```
c_str_less cmp; // declare an object
```

```
if (cmp("aa","ab"))
```

```
...
```

```
if( c_str_less() ("a","b") )
```

Creates temporary objects, and then call operator()

# Template comparator example

```
template<typename T>
class less {
public:
    bool operator()(const T& lhs, const T& rhs)
    { return lhs < rhs; }
};
```

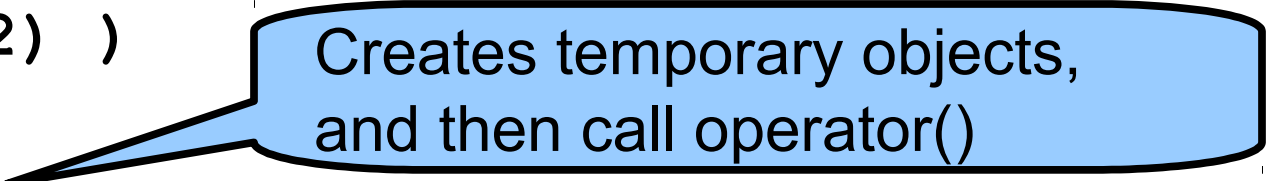
```
less<int> cmp;    // declare an object
```

```
if( cmp(1,2) )
```

```
...
```

```
if( less<int>()(1,2) )
```

```
...
```



Creates temporary objects,  
and then call operator()

# Using Comparators

```
// ascending order
// uses operator < for comparison
set<int> s1;
set<int, less<int>> s1; // same

// descending order
// uses operator > for comparison
set<int, greater<int>> s2;
```

# Using Comparators

```
set<int, MyComp> s3;
```

Creates a default constructed MyComp object.

```
MyComp cmp(42);
```

```
set<int, MyComp> s4(cmp);
```

Use given MyComp object.

# Why should we use classes as function objects?

- So we get the “power” of classes.
- Examples:
  - Inheritance.
  - To parameterize our functions in run time or in compile time.
  - To accumulate information.