

Standard Template Library (STL)

CONTAINERS

Effective STL: Items 1, 2, 3, 4, 5, 9, 11

Мина Димова

Видове (според предназначението)

- The standard STL sequence containers – *vector*, *string*, *deque*, *list*
- The standard STL associative containers – *set*, *multiset*, *map*, *multimap*
- The nonstandard sequence containers – *slist*, *rope*
- The nonstandard associative containers – *hash_set*, *hash_multiset*, *hash_map*, *hash_multimap*
- Standard non-STL containers – *array*, *bitset*, *stack*, *queue*, *priority_queue*

Видове (според разположението в паметта)


- ❑ Contiguous-memory containers (array-based containers) – *vector*, *string*, *deque* – store their elements in one or more(dynamically allocated) chunks of memory, each chunk holding more than one container element.
- ❑ Node-based containers – *list*, *map*, *set*, *multimap*, *multiset* – store only a single element per chunk of (dynamically allocated) memory.

Избор на контейнер

- ◆ Възможност за добавяне на елемент на произволно място – no associative
- ◆ От значение е подредбата на елементите – no hash
- ◆ Какъв итератор ще е необходим?
 - random access iterators – vector, deque, string
 - bidirectional iterators – no slist
- ◆ Изисква се да не се преместват присъстващите елементи при вмъкване или триене – no contiguous

The Illusion of container-independent code

- Sequence containers support `push_front` and/or `push_back`, while associative containers don't.
- Associative containers offer logarithmic-time `lower_bound`, `upper_bound`, and `equal_range` member functions, but sequence containers don't.
- When you insert an object into a sequence container, it stays where you put it, but if you insert an object into an associative container, the container moves the object to where it belongs in the container's sort order.

- 
- `vector<bool>` is not an STL container and it doesn't hold bools.
 - The form of `erase` taking an iterator returns a new iterator when invoked on a sequence container, but it returns nothing when invoked on an associative container.
 - The presence of `list` also means you give up `operator[]`, and you limit yourself to the capabilities of bidirectional iterators so you must stay away from algorithms that demand random access iterators, including `sort`, `stable_sort`, `partial_sort`, and `nth_element`

The Illusion of container-independent code

You're left with a "generalized sequence container" where you can't call `reserve`, `capacity`, `operator[]`, `push_front`, `pop_front`, `splice`, or any algorithm requiring random access iterators: a container where every call to insert and erase takes linear time and invalidates all iterators, pointers, and references: and a container incompatible with C where bools can't be stored. *Is that really the kind of container you want to use in your applications?*

Use of typedefs for container and iterator types

Instead of writing this

```
class Widget {...};  
vector<Widget> vw;  
Widget bestWidget = ...;  
vector<Widget>::iterator i = find(vw.begin(), vw.end(),  
bestWidget);
```

write this:

```
class Widget {...};  
typedef vector<Widget> WidgetContainer;  
typedef WidgetContainer::iterator WCIterator;  
WidgetContainer vw;  
Widget bestWidget = ...;  
WCIterator i = find(vw.begin(), vw.end(), bestWidget);
```


This makes it a lot easier to change container types, something that's especially convenient if the change in question is simply to add a custom allocator.

```
class Widget {... };  
template<typename T>  
SpecialAllocator{...}  
typedef vector<Widget, SpecialAllocator<Widget> >  
    WidgetContainer;  
typedef WidgetContainer::iterator WCIterator;  
WidgetContainer vw;  
Widget bestWidget;  
...  
WCIterator i = find(vw.begin(), vw.end(), bestWidget);
```

Cheap and correct copying

The slicing problem:

```
vector<Widget> vw;
```

```
class SpecialWidget : public Widget {...};
```

```
SpecialWidget sw;
```

```
vw.push_back(sw); //sw is copied as a base class
```

The solution:

Create containers of pointers instead of containers of objects when necessary

STL is generally designed to avoid copying objects unnecessarily.

```
Widget w[maxNumWidgets];
```

```
//create an array of maxNumWidgets
```

```
//Widgets, default-constructing each one
```

VS.

```
vector<Widget> vw;
```

```
//create a vector with zero Widget
```

```
//objects that will expand as needed
```

```
vw.reserve(maxNumWidgets);
```


Containers create only as many objects as you ask them for and use a default constructor only when you say they should.



Call `empty()` instead of checking
`size()` against zero

Empty is a constant-time operation for all standard containers while for some list implementations, size takes linear time.

But what makes list so troublesome?



```
list<int> list1;
```

```
list<int> list2;
```

```
...
```


```
list1.splice(list1.end(),list2,
```

```
    find(list2.begin(), list2.end(), 5),
```

```
    find(list2.rbegin(), list2.rend(), 10).base());
```

// it will work only if list2 contains a 10 somewhere
beyond a 5

//and how many elements are in list1 after the splice?!

- 
- If you want to find out how many elements are in a list, you'd like to make size a constant-time operation. Also you know that of all the standard containers only list offers the ability to splice elements from one place to another without copying any data.
 - And you know that splicing a range from one list to another can be accomplished in constant time.
 - If is to be a constant-time operation, each list member function must update the sizes of the lists on which it operates; which includes splice;
 - The only way for splice to update the sizes of the lists it modifies is for it to count the number of elements being spliced, and doing that would prevent splice from achieving the constant-time performance.

Prefer range member functions to their single-element counterparts

- It's generally less work to write the code using range member functions.
- Range member functions tend to lead to code that is clearer and more straightforward.

```
v1.assign(v2.begin() + v2.size() / 2, v2.end());
```

vs.

```
v1.clear();
```

```
for ( vector<Widget>::const_iterator ci = v2.begin() +  
      v2.size() / 2; ci != v2.end(); ++ci)
```

```
v1.push_back(*ci);
```



```
int data[numValues];  
vector<int> v;  
...  
v.insert(v.begin(), data, data + numValues);  
/* insert the ints in data into v at the front */  
  
vector<int>::iterator insertLoc(v.begin());  
for (int i = 0; i < numValues; ++i)  
{  
    insertLoc = v.insert(insertLoc, data[i]);  
}
```

- **Range construction** – all standard containers offer a constructor of this form:

container::container(InputIterator begin, InputIterator end);

- **Range insertion** – all standard sequence containers offer this form of insert:

void container::insert(iterator position, InputIterator begin, InputIterator end);

- **Range erasure**

- sequence containers:

*iterator container::erase(iterator begin,
iterator end);*

- associative container:

*void container::erase(iterator begin,
iterator end);*

- **Range assignment**

*void container::assign(InputIterator begin,
InputIterator end);*

Choose carefully among erasing options

- Container `<int> c;`
- `c.erase(remove(c.begin(), c.end(), 1963), c.end());`

/ the erase-remove idiom е най-добрият начин да премахнем елементи с определена стойност; c е vector, string или deque*/*

- `c.remove(1963);` */* c е list*/*
- `c.erase(1963);` *// standard associative container (logarithmic time)*

- `bool badValue(int x);`
- `c.erase(remove_if(c.begin(), c.end(),
badValue);` // *vector, string or deque*
- `c.remove_if(badValue);` // *c - list*

If the container is a standard associative container, use `remove_copy_if` and `swap`, or write a loop to walk the container elements, being sure to postincrement your iterator when you pass it to `erase`.

```
AssocContainer<int> c;
```

```
....
```

```
for(AssocContainer<int>::iterator i=c.begin();
```

```
    i!=c.end());
```

```
/*nothing*/
```

```
{    if(badValue(*i)) c.erase(i++);
```

```
    else ++i;
```

```
}
```

If the container is a standard sequence container, write a loop to walk the container elements, being sure to update your iterator with erase's return value each time you call it.

```
for (SeqContainer<int>::iterator i = c.begin();  
i != c.end();)  
{ if (badValue(*i))  
    {   logFile << "Erasing " << *i << '\n';  
        i = c.erase(i); // keep i valid by assigning  
    }   //erase's return value to it  
    else ++i;  
}
```


Allocators

Custom allocators are well suited when:

- you know that objects in certain containers are typically used together so you'd like to place them near one another in a special heap to maximize locality of reference;
- you'd like to set up a unique heap that corresponds to shared memory, then put one or more containers so they can be shared by other processes;

```
void* mallocShared(size_t bytesNeeded);
```

```
void freeShared(void *ptr);
```

// you'd like to put the contents of STL containers in
that shared memory

```
template<typename T>
class SharedMemoryANocator {
public:
    ...
    pointer allocate(size_type numObiects, const void *localityHint = 0)
    { return static_cast<pointer>(mallocShared(numObiects*
        sizeof(T)));
    }
    void deallocate(pointer ptrToMemory, size_type numObjects)
    { freeShared(ptrToMemory);
    }
    ...
};
```