OOP

Gavrilut Dragos

Course 8

rev 1

Summary

- Associative containers
- Smart Pointers

Associative containers

STL (associative containers - pair)

- “pair” is a template that contains two values (two different types)
- For use “**#include <utility>**”
- Two objects of type **pair** can be compared because the = operator is overloaded
- It’s internally used by associative containers
- The declaration of **pair** template:

App.cpp

```
template <class T1, class T2>
struct pair
{
     T1 first;
     T2 second;
     ...
}
```

STL (associative containers - map)

- Map is a container that store pairs of the form: key/value; the access to the **value** field can be done using the **key**
- For use "`#include <map>`"
- The key field is constant: after a key/value pair is added into a map, the key cannot be modified - only the value can be modified. Other pairs can be added and existing pairs can be deleted.
- The declaration of **map** template:

App.cpp

```
template < class Key, class Value, class Compare = less<Key> > class map
{
...
}
```

STL (associative containers - map)

- An example of using map:

App.cpp

```
void main(void)
{
	map<const char*, int> Grades;
	Grades[“Popescu"] = 10;
	Grades[“Ionescu"] = 9;
	Grades[“Marin"] = 7;
	printf(“Ionescu’s grade = %d\n", Grades[“Ionescu"]);
	printf(“Number of pairs = %d\n", Grades.size());
	map<const char*, int>::iterator it;
	for (it = Grades.begin(); it != Grades.end(); it++)
		printf("Grades[%s]=%d\n", it->first, it->second);
	it = Grades.find(“Ionescu");
	Grades.erase(it);
	int x = Grades [“Ionescu"];
	printf(“Ionescu’s grade = %d (x=%d)\n", Grades[“Ionescu"],x);
	printf("Number of pairs = %d\n", Grades.size());
}
```

Output

```
Ionescu’s grade  = 9
Number of pairs = 3
Grades[Popescu]=10
Grades[Ionescu]=9
Grades[Marin]=7
Ionescu’s grade  = 0 (x=0)
Number of pairs = 3
```

STL (associative containers - map)

- An example of using map:

App.cpp

```
void main(void)
{
	map<const char*, int> Grades;
	Grades[“Popescu"] = 10;
	Grades[“Ionescu"] = 9;
	Grades[“Marin"] = 7;
	printf(“Ionescu’s grade = %d\n", Grades[“Ionescu"]);
	printf(“Number of pairs = %d\n", Grades.size());
	map<const char*, int>::iterator it;
	for (it = Grades.begin(); it != Grades.end(); it++)
		printf("Grades[%s]=%d\n", it->first, it->second);
	it = Grades.find(“Ionescu");
	Grades.erase(it);
	int x = Grades [“Ionescu"];
	printf(“Ionescu’s grade = %d (x=%d)\n", Grades[“Ionescu"],x);
	printf("Number of pairs = %d\n", Grades.size());
}
```

The “Ionescu” key does not exist at this point. Because it not exists, a new one is created and the value is instantiated using the default constructor (which for **int** makes the value to be 0)

STL (associative containers - map)

- The methods supported by the map container:

Method/operator
Assignment (**operator=**)
Accessing and inserting values (**operator[]** and **at, insert** methods)
Deletion (**erase**, **clear** methods)
Search into a map (**find** method)
Iterators (**begin**, **end**, **rbegin**, **rend**, **cbegin**, **cend**, **crbegin**, **crend** (the last 4 from C++11))
Informations (**size**, **empty**, **max_size**)

STL (associative containers - map)

- Accessing elements ([] operator and **at**, **find** methods):

App.cpp

```
void main(void)
{
	map<const char*, int> Grades;
	Grades["Popescu"] = 10;
	printf("Grade = %d\n", Grades["Popescu"]);
}
```

App.cpp

```
void main(void)
{
	map<const char*, int> Grades;
	Grades["Popescu"] = 10;
	printf("Grade= %d\n", Grades.at("Popescu"));
}
```

App.cpp

```
void main(void)
{
	map<const char*, int> Grades;
	Grades["Popescu"] = 10;
	printf("Grade= %d\n", Grades.find("Popescu")->second);
}
```

STL (associative containers - map)

- Accessing elements ([] operator and **at**, **find** methods):

App.cpp

```
void main(void)
{
	map<const char*, int> Grades;
	Grades["Popescu"] = 10;
	printf("Grade = %d\n", Grades["Popescu"]);
}
```

App.cpp ("Ionescu" key does not exist)

```
void main(void)
{
	map<const char*, int> Grades;
	Grades["Popescu"] = 10;
	printf("Grade= %d\n", Grades.at("Ionescu"));
}
```

App.cpp ("Ionescu" key does not exist)

```
void main(void)
{
	map<const char*, int> Grades;
	Grades["Popescu"] = 10;
	printf("Grade= %d\n", Grades.find("Ionescu")->second);
}
```

STL (associative containers - map)

- Checking whether an element exists in a map can be done using **find** and **count** methods:

App.cpp

```
void main(void)
{
	map<const char*, int> Grades;
	Grades["Popescu"] = 10;
	if (Grades.find("Ionescu")==Grades.cend())
		printf("Ionescu does not exist in the grades list!");
}
```

App.cpp

```
void main(void)
{
	map<const char*, int> Grades;
	Grades["Popescu"] = 10;
	if (Grades.count("Ionescu")==0)
		printf("Ionescu does not exist in the grades list!");
}
```

STL (associative containers - map)

- The elements from a map containers are stored into a red-black tree
- This represents a compromise between the access/insertion time and the amount of allocated memory for the container
- Depending on what we are interested in, a map container is not always the best solution (e.g. if the number of insertions is much bigger than the number of reads, there are containers that are more efficient)
- We do the following experiment – the same algorithm is written using a map and a simple vector and we evaluate the insertion time
- The experiment is repeated ten times for each algorithm and the execution times are measured in milliseconds

STL (associative containers - map)

- The two algorithms:

Map-1.cpp

```
void main(void)
{
	map<int, int> Test;
	for (int tr = 0; tr < 1000000; tr++)
		Test[tr] = tr;
}
```

Map-2.cpp

```
void main(void)
{
	int *Test = new int[1000000];
	for (int tr = 0; tr < 1000000; tr++)
		Test[tr] = tr;
}
```

- Even if it is obvious that App-2 is more efficient, the hash collisions must be considered. For the above case, there are no hash collisions because the keys are **int**egers.

STL (associative containers - map)

- Results:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
Map-2	0	16	0	0	16	0	15	0	16	0	**6.3**

- The specifications of the system:
  - OS: Windows 8.1 Pro
  - Compiler: cl.exe [18.00.21005.1 for x86]
  - Hardware: Dell Latitude 7440 –i7 –4600U, 2.70 GHz, 8 GB RAM

STL (associative containers - multimap)

- “multimap” is a container similar to map. The difference is that a key can contain more values.
- For use “**#include <map>**”
- Accessing elements: the [] operator and the **at** method can not be used anymore.
- The declaration of **multimap** template:

App.cpp

```
template < class Key, class Value, class Compare = less<Key> > class multimap
{
...
}
```

STL (associative containers - multimap)

- An example of using multimap:

App.cpp

```
void main(void)
{
	multimap<const char*, int> Grades;
	Grades.insert(pair<const char*, int>("Ionescu", 10));
	Grades.insert(pair<const char*, int>("Ionescu", 8));
	Grades.insert(pair<const char*, int>("Ionescu", 7));
	Grades.insert(pair<const char*, int>("Popescu", 9));

	multimap<const char*, int>::iterator it;
	for (it = Grades.begin(); it != Grades.end(); it++)
	{
		printf("%s [%d]\n", it->first,it->second);

	}
}
```

Output

```
Ionescu [10]
Ionescu [8]
Ionescu [7]
Popescu [9]
```

STL (associative containers - multimap)

- An example of using multimap:

App.cpp

```
void main(void)
{
	multimap<const char*, int> Grades;
	Grades.insert(pair<const char*, int>("Ionescu", 10));
	Grades.insert(pair<const char*, int>("Ionescu", 8));
	Grades.insert(pair<const char*, int>("Ionescu", 7));
	Grades.insert(pair<const char*, int>("Popescu", 9));
	Grades.insert(pair<const char*, int>("Popescu", 6));
	Grades.insert(pair<const char*, int>("Georgescu", 8));

	multimap<const char*, int>::iterator it;
	it = Grades.begin();
	printf("%s->%d\n", it->first, it->second);

	it = Grades.upper_bound(it->first);
	printf("%s->%d\n", it->first, it->second);

	it = Grades.upper_bound(it->first);
	printf("%s->%d\n", it->first, it->second);
}
```

Output

```
Ionescu->10
Popescu->9
Georgescu->8
```

STL (associative containers - multimap)

- An example of using multimap:

App.cpp

```
void main(void)
{
	multimap<const char*, int> Grades;
	Grades.insert(pair<const char*, int>("Ionescu", 10));
	Grades.insert(pair<const char*, int>("Ionescu", 8));
	Grades.insert(pair<const char*, int>("Ionescu", 7));
	Grades.insert(pair<const char*, int>("Popescu", 9));
	Grades.insert(pair<const char*, int>("Popescu", 6));
	Grades.insert(pair<const char*, int>("Georgescu", 8));

	multimap<const char*, int>::iterator it;
	for (it = Grades.begin(); it != Grades.end(); it = Grades.upper_bound(it->first))
	{
		printf("Unique key: %s\n", it->first);

	}
}
```

Output

```
Unique key: Ionescu
Unique key: Popescu
Unique key: Georgescu
```

STL (associative containers - multimap)

- An example of using multimap:

App.cpp

```
void main(void)
{
	multimap<const char*, int> Grades;
	Grades.insert(pair<const char*, int>("Ionescu", 10));
	Grades.insert(pair<const char*, int>("Ionescu", 8));
	Grades.insert(pair<const char*, int>("Ionescu", 7));
	Grades.insert(pair<const char*, int>("Popescu", 9));
	Grades.insert(pair<const char*, int>("Popescu", 6));
	Grades.insert(pair<const char*, int>("Georgescu", 8));

	multimap<const char*, int>::iterator it;
	it = Grades.begin();
	while (it != Grades.end())
	{
		pair <multimap<const char*, int>::iterator, multimap<const char*, int>::iterator> range;
		range = Grades.equal_range(it->first);
		printf("%s's grades: ", it->first);
		for (it = range.first; it != range.second; it++)
			printf("%d,", it->second);
		printf("\n");
	}
}
```

Output

```
Ionescu's grades: 10,8,7
Popescu's grades:9,6
Georgescu's grades:8
```

STL (associative containers - multimap)

- An example of using multimap:

App.cpp

```
void main(void)
{
	multimap<const char*, int> Grades;
	Grades.insert(pair<const char*, int>("Ionescu", 10));
	Grades.insert(pair<const char*, int>("Ionescu", 8));
	Grades.insert(pair<const char*, int>("Ionescu", 7));
	Grades.insert(pair<const char*, int>("Popescu", 9));
	Grades.insert(pair<const char*, int>("Popescu", 6));
	Grades.insert(pair<const char*, int>("Georgescu", 8));

	multimap<const char*, int>::iterator it,it2;
	it = Grades.find("Popescu");
	it2 = Grades.upper_bound(it->first);

	printf(“Popescu’s grades: ");
	while (it != it2)
	{
		printf("%d,", it->second);
		it++;
	}
}
```

Output

```
Popescu’s grades: 9,6
```

STL (associative containers - multimap)

- An example of using multimap:

App.cpp

```
void main(void)
{
	multimap<const char*, int> Grades;
	Grades.insert(pair<const char*, int>("Ionescu", 10));
	Grades.insert(pair<const char*, int>("Ionescu", 8));
	Grades.insert(pair<const char*, int>("Ionescu", 7));
	Grades.insert(pair<const char*, int>("Popescu", 9));
	Grades.insert(pair<const char*, int>("Popescu", 6));
	Grades.insert(pair<const char*, int>("Georgescu", 8));

	multimap<const char*, int>::iterator it,it2;
	for (it = Grades.begin(); it != Grades.end(); it = Grades.upper_bound(it->first))
	{
		printf("%s has %d grades\n", it->first, Grades.count(it->first));
	}
}
```

Output

```
Ionescu has 3 grades
Popescu has 2 grades
Georgescu has 1 grades
```

STL (associative containers - multimap)

- An example of using multimap:

App.cpp

```
void main(void)
{
	multimap<const char*, int> Grades;
	Grades.insert(pair<const char*, int>("Ionescu", 10));
	Grades.insert(pair<const char*, int>("Ionescu", 8));
	Grades.insert(pair<const char*, int>("Ionescu", 7));
	Grades.insert(pair<const char*, int>("Popescu", 9));
	Grades.insert(pair<const char*, int>("Popescu", 6));
	Grades.insert(pair<const char*, int>("Georgescu", 8));

	multimap<const char*, int>::iterator it,it2;
	it = Grades.find("Ionescu");
	it++;
	Grades.erase(it); // the “8” grade from Ionescu is deleted
	for (it = Grades.begin(); it != Grades.end(); it++)
	{
		printf("%s [%d]\n", it->first, it->second);
	}
}
```

Output

```
Ionescu [10]
Ionescu [7]
Popescu [9]
Popescu [6]
Georgescu [8]
```

STL (associative containers - multimap)

- An example of using multimap:

App.cpp

```
void main(void)
{
	multimap<const char*, int> Grades;
	Grades.insert(pair<const char*, int>("Ionescu", 10));
	Grades.insert(pair<const char*, int>("Ionescu", 8));
	Grades.insert(pair<const char*, int>("Ionescu", 7));
	Grades.insert(pair<const char*, int>("Popescu", 9));
	Grades.insert(pair<const char*, int>("Popescu", 6));
	Grades.insert(pair<const char*, int>("Georgescu", 8));

	if (Grades.find("Ionescu") != Grades.cend())
		printf("Ionescu exists!\n");

	if (Grades.find("Marin") == Grades.cend())
		printf("Marin DOES NOT exist!\n");
}
```

Output

```
Ionescu exists!
Marin DOES NOT exist!
```

STL (associative containers - multimap)

- The methods supported by the multimap container:

Method/operator
Assignment (**operator=**)
Insertion (**insert**)
Deletion (**erase**, **clear** methods)
Accessing elements (**find** method)
Iterators (**begin**, **end**, **rbegin**, **rend**, **cbegin**, **cend**, **crbegin**, **crend** (the last 4 from C++11))
Informations (**size**, **empty**, **max_size** methods)
Special methods (**upper_bound** and **lower_bound** – to access the intervals in witch there are elements with the same key; **equal_range** - to obtain an interval for all the elements stored for a key)

STL (associative containers - set)

- “set” is a container that store unique elements
- For use “`#include <set>`”
- The declaration of **set** template:

App.cpp

```
template < class Key, class Compare = less<Key> > class set
{
...
}
```

STL (associative containers - set)

- An example of using set:

App.cpp

```
void main(void)
{
	set<int> s;
	s.insert(10);
	s.insert(20);
	s.insert(5);
	s.insert(10);
	set<int>::iterator it;

	for (it = s.begin(); it != s.end(); it++)
		printf("%d ", *it);
}
```

Output

```
5 10 20
```

STL (associative containers - set)

- An example of using set:

App.cpp

```
struct Comparator {
	bool operator() (const int& leftValue, const int& rightValue) const
	{
		return (leftValue / 20) < (rightValue / 20);
	}
};

void main(void)
{
	set<int, Comparator> s;
	s.insert(10);
	s.insert(20);
	s.insert(5);
	s.insert(10);
	set<int, Comparator>::iterator it;

	for (it = s.begin(); it != s.end(); it++)
		printf("%d ", *it);
}
```

Output

```
10 20
```

STL (associative containers - set)

- The elements from a “set” follows a specific order.
- This brings a performance penalty.

Set-1.cpp

```
void main(void)
{
	set<int> s;
	for (int tr = 0; tr < 1000000; tr++)
		s.insert(tr);
}
```

- Let’s remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
Map-2	0	16	0	0	16	0	15	0	16	0	**6.3**
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**

STL (associative containers - set)

- The methods supported by the set container:

Method/operator
Assignment (**operator=**)
Insertion (**insert** method)
Deletion (**erase**, **clear** methods)
Accessing elements (**find** method)
Iterators (**begin**, **end**, **rbegin**, **rend**, **cbegin**, **cend**, **crbegin**, **crend** (the last 4 from C++11))
Informations (**size**, **empty**, **max_size** methods)

STL (associative containers - multiset)

- “**multiset**” is similar with set, but duplicate elements are allowed
- For use “`#include <set>`”
- The declaration of **multiset** template:

App.cpp

```
template < class Key, class Compare = less<Key> > class multiset
{
...
}
```

STL (associative containers - multiset)

- An example of using multiset:

App.cpp

```
void main(void)
{
	multiset<int> s;
	s.insert(10);
	s.insert(20);
	s.insert(5);
	s.insert(10);
	multiset<int>::iterator it;

	for (it = s.begin(); it != s.end(); it++)
		printf("%d ", *it);
}
```

Output

```
5 10 10 20
```

STL (associative containers - multiset)

- The methods supported by the **multiset** container:

Method/operator
Assignment (**operator=**)
Insertion (**insert**)
Deletion (**erase**, **clear** methods)
Accessing elements (**find** method)
Iterators (**begin**, **end**, **rbegin**, **rend**, **cbegin**, **cend**, **crbegin**, **crend** (the last 4 from C++11))
Informations (**size**, **empty**, **max_size** methods)
Special methods (**upper_bound** and **lower_bound** – to access the intervals in witch there are elements with the same key; **equal_range** - to obtain an interval that includes all the elements which have a specific key)

STL (associative containers - unordered_map)

- The elements from an **unordered_map** container are stored into a hash-table
- Introduced in C++11
- For use "**#include <unordered_map>**"
- Supports the same methods as **map** plus methods for the control of buckets.
- The declaration of **unordered_map** template:

App.cpp

```
template < class Key, class Value, class Hash, class Equal > class unordered_map
{
...
}
```

STL (associative containers - unordered_map)

- How hash tables works:

Popescu

Ionescu

Georgescu

Hash function(transforms a string into a index)

Index	Value

STL (associative containers - unordered_map)

- How hash tables works:

Popescu

Ionescu

Georgescu

We consider the following hashing function:

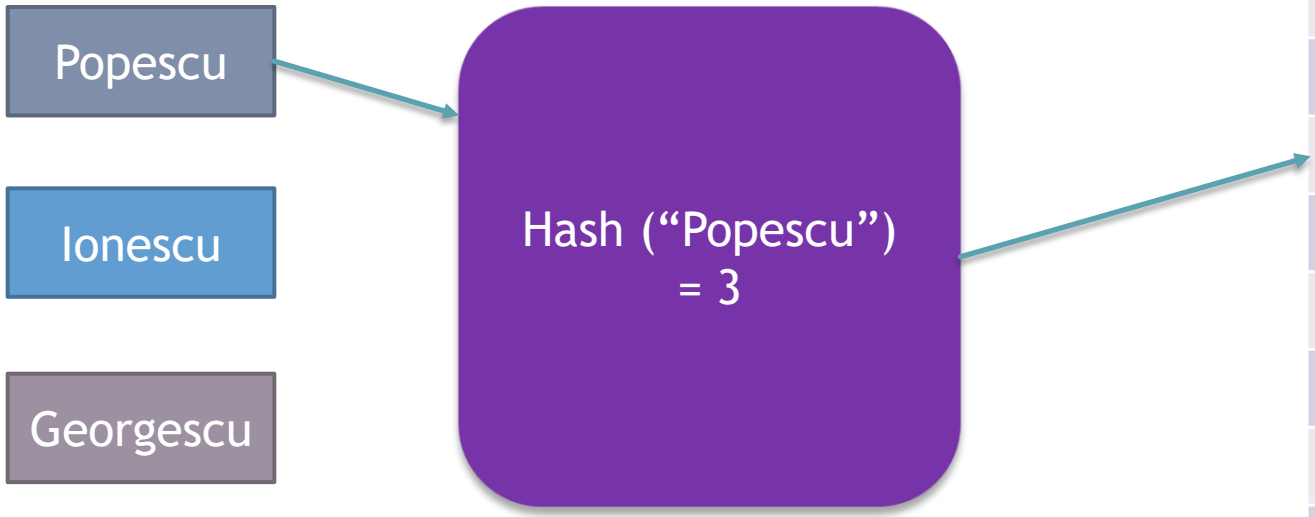
HashFunction

```
int HashFunction(const char* s)
{
	int sum = 0;
	while ((*s) != 0)
	{
		sum += (*s);
		s++;
	}
	return sum % 12;
}
```

Index	Value
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL
10	NULL
11	NULL

STL (associative containers - unordered_map)

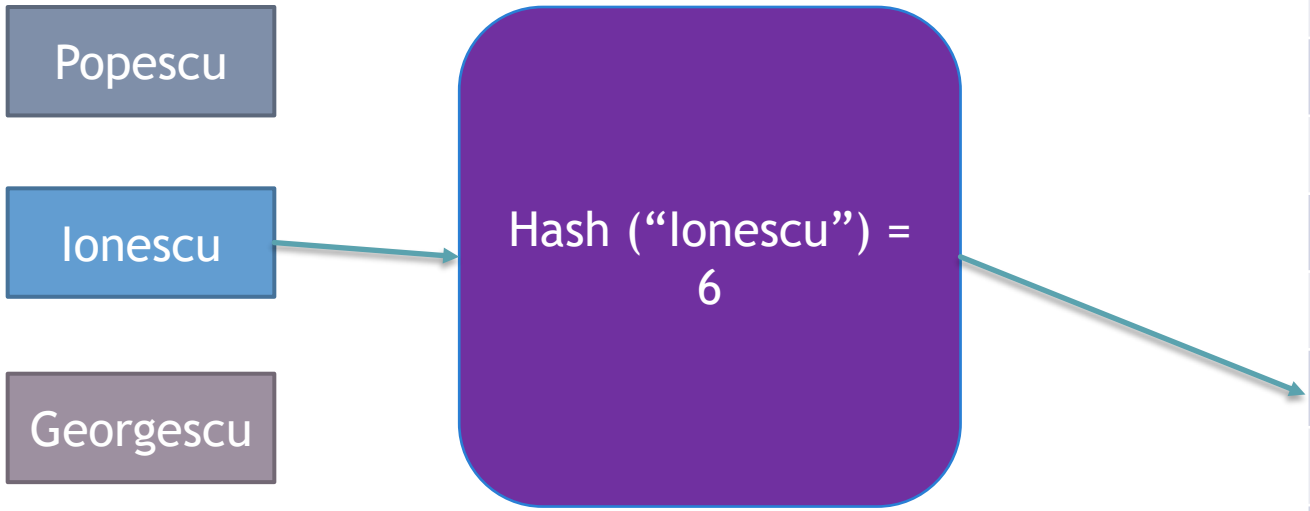
- How hash tables works:



Index	Value
0	NULL
1	NULL
2	NULL
3	Popescu
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL
10	NULL
11	NULL

STL (associative containers - unordered_map)

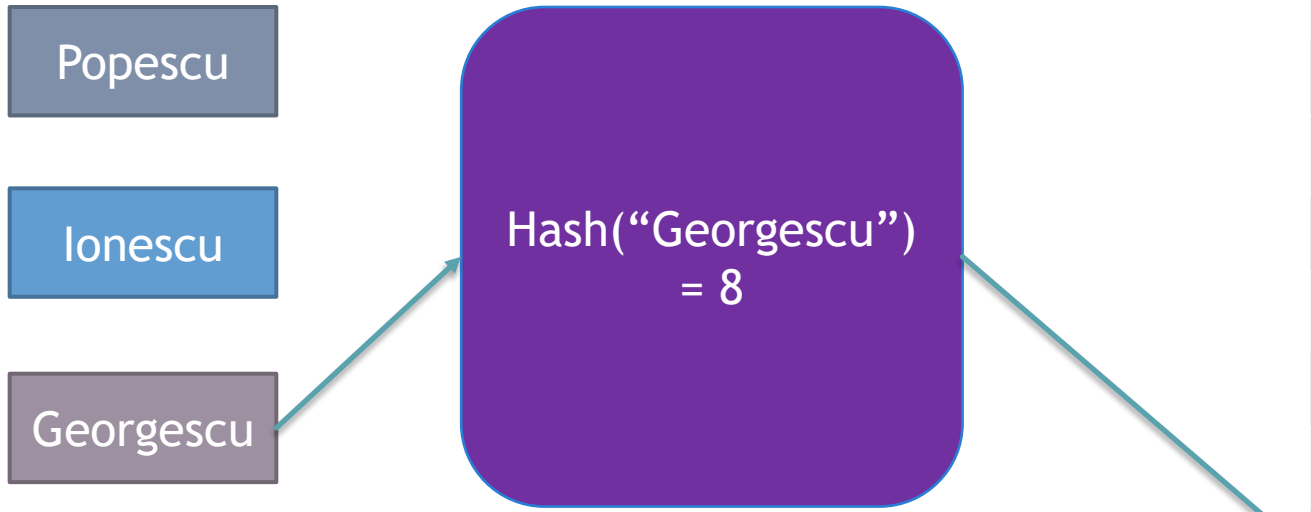
- How hash tables works:



Index	Value
0	NULL
1	NULL
2	NULL
3	Popescu
4	NULL
5	NULL
6	Ionescu
7	NULL
8	NULL
9	NULL
10	NULL
11	NULL

STL (associative containers - unordered_map)

- How hash tables works:



Index	Value
0	NULL
1	NULL
2	NULL
3	Popescu
4	NULL
5	NULL
6	Ionescu
7	NULL
8	Georgescu
9	NULL
10	NULL
11	NULL

STL (associative containers - unordered_map)

- Consider the following code:

Umap-1.cpp

```
void main(void)
{
	unordered_map<int,int> Test;
	for (int tr = 0; tr < 1000000; tr++)
		Test[tr]=tr;
}
```

- Let's remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
Map-2	0	16	0	0	16	0	15	0	16	0	**6.3**
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	**15310**

STL (associative containers - unordered_map)

- Consider the following code:

Umap-2.cpp

```
void main(void)
{
	unordered_map<int,int> Test;
	Test.reserve(1000000);
	for (int tr = 0; tr < 1000000; tr++)
		Test[tr]=tr;
}
```

- Let's remake the previous experiment :

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
Map-2	0	16	0	0	16	0	15	0	16	0	**6.3**
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	**15310**
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	**10006**

STL (associative containers - unordered_set)

- **“unordered_set”** is similar with the **set** container, but the elements are not sorted
- Introduced in C++11
- For use “**#include <unordered_set>**”
- Supports the same methods as **set** plus methods for the control of buckets
- The declaration of **unordered_set** template:

App.cpp

```
template < class Key, class Hash, class Equal > class unordered_set
{
...
}
```

STL (associative containers - unordered_set)

- Consider the following code:

Uset-1.cpp

```
void main(void)
{
	unordered_set<int> s;
	for (int tr = 0; tr < 1000000; tr++)
		s.insert(tr);
}
```

- Let's remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
Map-2	0	16	0	0	16	0	15	0	16	0	**6.3**
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	**15310**
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	**10006**
Uset-1	12140	11625	12047	11984	12109	12078	11609	11578	11782	11672	**11862**

STL (associative containers - unordered_set)

- And the variant with **reserve**:

Uset-2.cpp

```
void main(void)
{
	unordered_set<int> s;
	s.reserve(1000000);
	for (int tr = 0; tr < 1000000; tr++)
		s.insert(tr);
}
```

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Medie
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
Map-2	0	16	0	0	16	0	15	0	16	0	**6.3**
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	**15310**
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	**10006**
Uset-1	12140	11625	12047	11984	12109	12078	11609	11578	11782	11672	**11862**
Uset-2	6516	6328	6875	6844	6812	6453	6453	6531	6500	6515	**6582**

STL (associative containers)

- Besides the presented containers, two other associative containers exists:
  - unordered_multimap
  - unordered_multiset
- Similar with multimap/multiset (the difference is that hash tables are used, not sorted trees)
- When choosing between **map** and **unordered_map (set** and **unordered_set**, **…),** the amount of available memory and the execution times (insertion, deletion, access of elements, …) must be considered.

Smart Pointers

Smart Pointers

- A smart pointer in C++ is an object that behaves like a pointer but manages the lifetime of the allocated memory it points to. Unlike raw pointers smart pointers automatically free the memory when it's no longer needed

- C++ has the following smart pointers:
  - unique_ptr
  - shared_ptr
  - weak_ptr

- Up until C++11, there was also auto_ptr (that became deprecated with C++11) and was removed from C++17.

unique_ptr

- std::unique_ptr is a C++ smart pointer that owns a dynamically allocated object exclusively and automatically deletes it when the **unique_ptr** goes out of scope. Unique_ptr is defined in <memory> (*use #include "memory"*)
- To create a **unique_ptr** you can use:

1. Its own constructor (from C++11)

App.cpp

```
std::unique_ptr<int> a = std::unique_ptr<int>(new int);
*a = 10;
```

2. Std::make_unique (from C++14)

App.cpp

```
std::unique_ptr<int> a = std::make_unique<int>(10)
```

unique_ptr

- Let’s see an example:

App.cpp (with ctor)

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {
    Test* t = new Test(10);
    unique_ptr<Test> a = unique_ptr<Test>(t);
    cout << a->x;
    return 0;
}
```

App.cpp (with make_unique)

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main()
{
    unique_ptr<Test> a = make_unique<Test>(10);
    cout << a->x;
    return 0;
}
```

unique_ptr

- What is the difference:
- Unique_ptr works like **emplace** (meaning that it creates the pointer in place – that implies that there is only one **OWNER** over that memory location).

App.cpp

```
unique_ptr<_Ty> make_unique(_Types&&... _Args)
```

- In contrast using the **ctor** you can keep a copy of that pointer and still use it.

App.cpp

```
explicit unique_ptr(pointer _Ptr) noexcept
```

unique_ptr

- Let’s analyze the following example:

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {
    Test* t = new Test(10);
    unique_ptr<Test> a = unique_ptr<Test>(t);
    cout << a->x << "," << t->x;
    delete t;
    cout << a->x;
    return 0;
}
```

unique_ptr

- Let’s analyze the following example:

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {
    Test* t = new Test(10);
    unique_ptr<Test> a = unique_ptr<Test>(t);
    cout << a->x << "," << t->x;
    delete t;
    cout << a->x;
    return 0;
}
```

This code will print 10,10

unique_ptr

- Let's analyze the following example:

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {
    Test* t = new Test(10);
    unique_ptr<Test> a = unique_ptr<Test>(t);
    cout << a->x << "," << t->x;
    delete t;
    cout << a->x;
    return 0;
}
```

This type of behavior led to the creation of **make_unique** template that avoids creating the pointer separately and also being able to access / delete the pointer outside the unique_ptr object.

This code will **crash** (once "t" is deleted – "a" remains a unique_ptr object that points to a deallocated memory zone)

unique_ptr

- However, because of the **get()** method from unique_ptr the same behavior can be obtained even if using **make_unique()**, as get() method returns a pointer towards the inner data that can further be used to modify it.

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {
    unique_ptr<Test> a = make_unique<Test>(10);
    Test* t = a.get();
    cout << a->x << "," << t->x;
    delete t;
    cout << a->x;
    return 0;
}
```

This code will **crash** (once “t” is deleted – “a” remains a unique_ptr object that points to a deallocated memory zone)

unique_ptr

- Unique_ptr however has a method **release()** that returns the inner pointer and clears it (similar to removing the ownership over that pointer from the object level). This is different from the get() method as the unique_ptr object will not have access to the inner object anymore.

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {
    unique_ptr<Test> a = make_unique<Test>(10);
    Test* t = a.release();
    delete t;
    return 0;
}
```

This code will compile and run as expected.

unique_ptr

- In contrast, using get() method will crash the execution when if the newly obtained pointer (“t”) is being deleted manually as the destructor of unique_ptr will attempt to delete it one more time when “a” variable goes out of scope.

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {
    unique_ptr<Test> a = make_unique<Test>(10);
    Test* t = a.get();
    delete t;
    return 0;
}
```

```
return 0;
mov         dword ptr [rbp+144h],0
lea         rcx,[a]
call        std::unique_ptr<Test>::destructor
mov         eax,dword ptr [rbp+144h]
```

This code will crash when the destructor of the “a” variable is being called.

unique_ptr

- unique_ptr has some methods (ctors) deleted. This offers a way to NOT ALLOW COPYING its content:

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {

    unique_ptr<Test> a = make_unique<Test>(10);

    unique_ptr<Test> b = a;

    return 0;
}
```

This is OK (the move ctor) will be used in this case as make_unique<…> will return an xvalue

unique_ptr

- unique_ptr has some methods (ctors) deleted. This offers a way to NOT ALLOW COPYING its content:

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {

    unique_ptr<Test> a = make_unique<Test>(10);

    unique_ptr<Test> b = a;

    return 0;
}
```

```
error C2280:
unique_ptr<Test>::unique_ptr(const unique_ptr<Test> &):
attempting to reference a deleted function
```

This code will not compile as it implies using the **copy-constructor** that will duplice the inner pointer.

unique_ptr

- Similarly – the copy-assignament operator is also deleted for the same reason:

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
};

int main() {

    unique_ptr<Test> a = make_unique<Test>(10);

    unique_ptr<Test> b;
    b = a;

    return 0;
}
```

```
error C2280:
unique_ptr<Test>::operator= (const unique_ptr<Test> &):
attempting to reference a deleted function
```

This code will not compile as it implies using the **copy-asignament operator** that will duplice the inner pointer.

unique_ptr

- You can also use the reset() method to delete the inner pointer and replace it with a new one:

App.cpp

```
#include <iostream>
#include <memory>

using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {

    unique_ptr<Test> a = make_unique<Test>(10);
    cout << a->x << "\n";
    a.reset(new Test(20));
    cout << a->x << "\n";

    return 0;
}
```

```
Output:
10
DTOR(x=10)
20
DTOR(x=20)
```

unique_ptr

- The bool operator is also overwritten to allow one to check if the inner pointer is valid (not null) or not.

App.cpp

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v): x(v) {}
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    unique_ptr<Test> a = make_unique<Test>(10);
    cout << a->x << "\n";
    a.reset(nullptr);
    if (a) {
        cout << a->x << "\n";
    } else {
        cout << "a is empty";
    }
    return 0;
}
```

```
Output:
10
DTOR(x=10)
a is empty
```

In this case we use .reset(nullptr) to delete the inner pointer manually.

shared_ptr

- std::shared_ptr is a C++ smart pointer that holds an inner counter that reflects how many objects refer to the same memory location). This is often called a ***reference count*** (and it prevents deallocation if there are still some objects / variable that keep a pointer to that memory location).
- To create a **shared_ptr** you can use:
  1. Its own constructor (from C++11)

App.cpp

```
std::shared_ptr<int> a = std::shared_ptr<int>(new int);
*a = 10;
```

  2. Std::make_shared (from C++14)

App.cpp

```
std::shared_ptr<int> a = std::make_shared<int>(10)
```

shared_ptr

- shared_ptr takes more memory than a regular pointer:

App.cpp (run on x64)

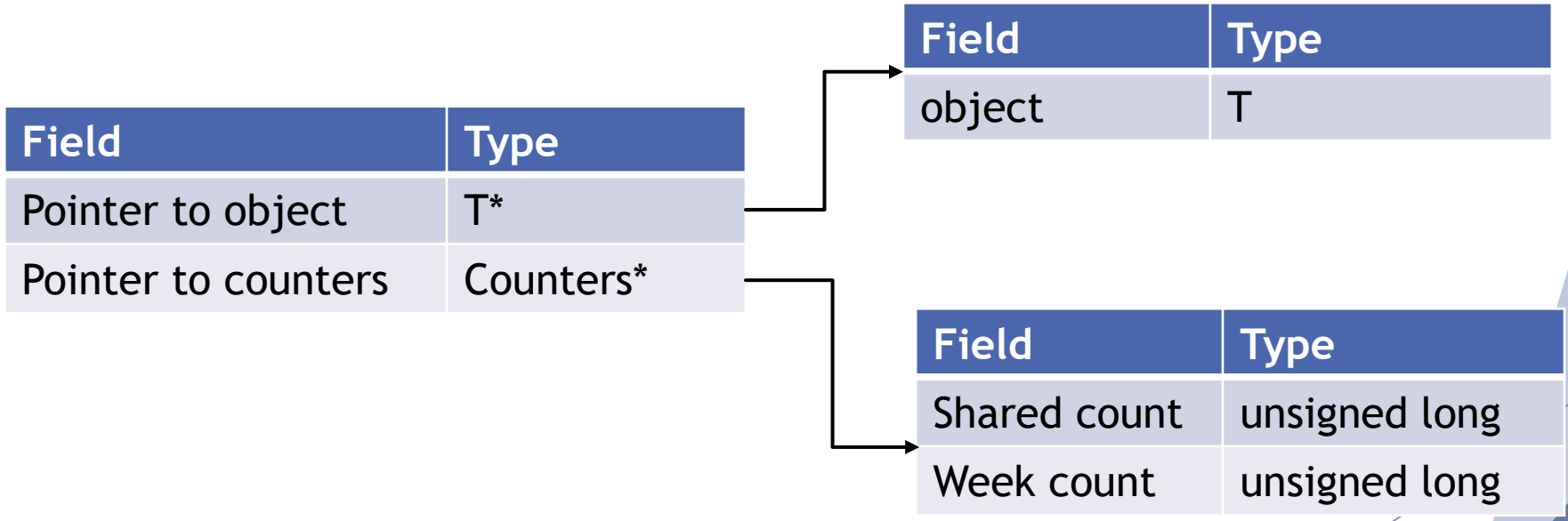
```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    cout << "Shared: " << sizeof(shared_ptr<int>) << endl;
    cout << "Unique: " << sizeof(unique_ptr<int>) << endl;
    return 0;
}
```

```
Output:
Shared: 16
Unique: 8
```

- This is because a shared_ptr needs to hold the reference count as well (as part of the object).

shared_ptr

- A simplified version on how a shared_ptr looks in memory:



- **Remarks:** Keep in mind that implementation may be different from version to version as well as between compilers (gcc , cl, clang).

shared_ptr

How it works:

- When the shared_ptr is created for the first time, the memory is allocated, and the counters are initialized
- Whenever a new copy of the shared_ptr is being created (e.g.: via operator=) the counter is incremented
- When the copy is destroyed the counter is decrements
- When the counter reaches 0, the entire object is destroyed.

Remarks: counter operations are **ATOMIC** (meaning that they are not affected by a multi-thread execution). Access to the inner type is however **NOT** thread-safe !

shared_ptr

- Let’s analyze the following example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    cout << t1.use_count() << endl;
    {
        shared_ptr<Test> t2 = t1;
        cout << t1.use_count() << "," << t2.use_count() << endl;
        {
            shared_ptr<Test> t3 = t2;
            cout << t1.use_count() << "," << t2.use_count() << "," << t3.use_count() << endl;
        }
        cout << t1.use_count() << "," << t2.use_count() << endl;
    }
    cout << t1.use_count() << endl; return 0;
}
```

Output:

```
CTOR(x=10)
1
2,2
3,3,3
2,2
1
DTOR(x=10)
```

shared_ptr

- Let's analyze the following example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    cout << t1.use_count() << endl;
    {
        shared_ptr<Test> t2 = t1;
        cout << t1.use_count() << "," << t2.u
        {
            shared_ptr<Test> t3 = t2;
            cout << t1.use_count() << "," << t2.use_count() << "," << t3.use_count() << endl;
        }
        cout << t1.use_count() << "," << t2.use_count() << endl;
    }
    cout << t1.use_count() << endl; return 0;
}
```

```
Output:
CTOR(x=10)
1
2,2
3,3,3
2,2
1
DTOR(x=10)
```

An object of type Test is being created (and allocated in memory).

shared_ptr

- Let's analyze the following example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    cout << t1.use_count() << endl;
    {
        shared_ptr<Test> t2 = t1;
        cout << t1.use_count() << '
        {
            shared_ptr<Test> t3 = t
            cout << t1.use_count()
        }
        cout << t1.use_count() << "," << t2.use_count() << endl;
    }
    cout << t1.use_count() << endl; return 0;
}
```

```
Output:
CTOR(x=10)
1
2,2
3,3,3
2,2
1
DTOR(x=10)
```

The reference count is set to 1

shared_ptr

- Let's analyze the following example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    cout << t1.use_count() << endl;
    {
        shared_ptr<Test> t2 = t1;
        cout << t1.use_count() << "," << t2.use_count() << endl;
        {
            shared_ptr<Test> t3 = t2;
            cout << t1.use_count() << "," << t2.use_cou    << "," << t3.use_count() << endl;

    }
    c
}
```

```
Output:
CTOR(x=10)
1
2,2
3,3,3
2,2
1
DTOR(x=10)
```

When we create "t2" from "t1" we will just increase the reference count (both t1 and t2 points to the same reference count block)

shared_ptr

- Let's analyze the following example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(i
    ~Test(
};
int main()
    share
    cout <
    {
        shared_ptr<Test> t2 = t1;
        cout << t1.use_count() << "," <  .use_count() << endl;
        {
            shared_ptr<Test> t3 = t2;
            cout << t1.use_count() << "," << t2.use_count() << "," << t3.use_count() << endl;
        }
        cout << t1.use_count() << "," << t2.use_count() << endl;
    }
    cout << t1.use_count() << endl; return 0;
}
```

The same logic applies for "t3" - the reference count is increased (t1, t2 and t3 have the same reference count - 3)

```
Output:
CTOR(x=10)
1
2,2
3,3,3
2,2
1
DTOR(x=10)
```

shared_ptr

- Let's analyze the following example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int
    ~Test()
};
int main()
    shared_
    cout <<
    {
        shared_ptr<Test> t2 = t1;
        cout << t1.use_count() << "," << t2.use_count() << endl;
        {
            shared_ptr<Test> t3 = t2;
            cout << t1.use_count() << "," << t2.use_count() << "," << t3.use_count() << endl;
        }
        cout << t1.use_count() << "," << t2.use_count() << endl;
    }
    cout << t1.use_count() << endl; return 0;
}
```

When the **scope** of "t3" ends the reference count is decreased to 2.

"t3" scope

```
Output:
CTOR(x=10)
1
2,2
3,3,3
2,2
1
DTOR(x=10)
```

shared_ptr

- Let's analyze the following example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using na
struct T
    int
    Test
    ~Tes
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    cout << t1.use_count() << endl;
    {
        shared_ptr<Test> t2 = t1;
        cout << t1.use_count() << "," << t2.use_count() << endl;
        {
            shared_ptr<Test> t3 = t2;
            cout << t1.use_count() << "," << t2.use_count() << "," << t3.use_count() << endl;
        }
        cout << t1.use_count() << "," << t2.use_count() << endl;
    }
    cout << t1.use_count() << endl; return 0;
}
```

When the **scope** of "t2" ends, the reference count is decreased to 1.

"t2" scope

```
Output:
CTOR(x=10)
1
2,2
3,3,3
2,2
1
DTOR(x=10)
```

shared_ptr

- Let's analyze the following example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    cout << t1.use_count() << endl;
    {
        shared_ptr<Test> t2 = t1;
        cout << t1.use_count() << endl;
        shared_ptr<Test> t3 = t1;
        cout << t1.use_count() << endl;
    }
    cout << t1.use_count() << endl; return 0;
}
```

Output:

```
CTOR(x=10)
1
2,2
3,3,3
2,2
1
DTOR(x=10)
```

Finally, when "t1" goes out of scope, the reference count decreases to 0 and the destructor for the object is being called.

shared_ptr

- The counter increment is done when you create a new object (via copy ctor – see creation of **t2**) or via assignment (operator=) – see assignment of **t2 to t3**.

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    shared_ptr<Test> t3;
    cout << "Iniailly       : " << t1.use_count() << endl;
    shared_ptr<Test> t2 = t1;
    cout << "After t2 CTOR  : " << t1.use_count() << endl;
    t3 = t1;
    cout << "After t2 Assign: " << t1.use_count() << endl;
    return 0;
}
```

Output:

```
CTOR(x=10)
Iniailly       : 1
After t2 CTOR  : 2
After t2 Assign: 3
DTOR(x=10)
```

shared_ptr

- There is also a method **.get()** that returns the raw pointer to an object, that can create similar problems in terms of releasing the memory of a pointer:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    Test* t2 = t1.get();
    std::cout << "Count = " << t1.use_count() << endl;
    delete t2;
    std::cout << "Count = " << t1.use_count() << endl;
    return 0;
}
```

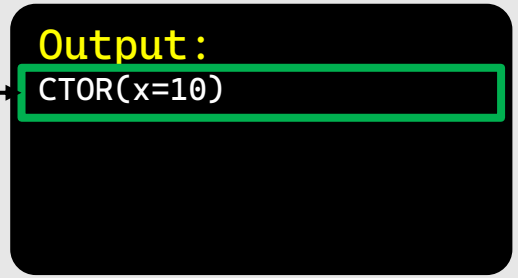
Output:

shared_ptr

- There is also a method **.get()** that returns the raw pointer to an object, that can create similar problems in terms of releasing the memory of a pointer:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    Test* t2 = t1.get();
    std::cout << "Count = " << t1.use_count() << endl;
    delete t2;
    std::cout << "Count = " << t1.use_count() << endl;
    return 0;
}
```



shared_ptr

- There is also a method **.get()** that returns the raw pointer to an object, that can create similar problems in terms of releasing the memory of a pointer:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    i
    T                                    )\n"; }
    ~
};
int m
    shared_ptr<Test> t1 = make_shared<Test>(10);
    Test* t2 = t1.get();
    std::cout << "Count = " << t1.use_count() << endl;
    delete t2;
    std::cout << "Count = " << t1.use_count() << endl;
    return 0;
}
```

Reference count is 1, however, there are two pointers to the memory allocated: **t1** and **t2**

Output:

```
CTOR(x=10)
Count = 1
```

shared_ptr

- There is also a method **.get()** that returns the raw pointer to an object, that can create similar problems in terms of releasing the memory of a pointer:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~T
};
int ma
    sh
    Te
    std::cout << "Count = " << t1.use_count() << endl;
    delete t2;
    std::cout << "Count = " << t1.use_count() << endl;
    return 0;
}
```

```
Output:
CTOR(x=10)
Count = 1
DTOR(x=10)
```

Deleting **t2** will free the memory allocated by the object (hence the call to the Test::DTOR)

shared_ptr

- There is also a method **.get()** that returns the raw pointer to an object, that can create similar problems in terms of releasing the memory of a pointer:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
};
int
    std::cout << "Count = " << t1.use_count() << endl;
    delete t2;
    std::cout << "Count = " << t1.use_count() << endl;
    return 0;
}
```

Even if "t2" was deleted (the reference count block was NOT) and as such calling t1.use_count() will return 1

```
Output:
CTOR(x=10)
Count = 1
DTOR(x=10)
Count = 1
```

shared_ptr

- There is also a method **.get()** that returns the raw pointer to an object, that can create similar problems in terms of releasing the memory of a pointer:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    Test* t2 = t1.get();
    std::cout << "Count = " << t1.use_count() << endl;
    delete t2;
    std::cout << "Count = " << t1.use_count() << endl;
    return 0;
}
```

```
Output:
CTOR(x=10)
Count = 1
DTOR(x=10)
Count = 1
```

At this point, the code will crush as “**t1**” will attempt to call de destructor of Test again once it decrements the reference count and reaches 0

shared_ptr

- There is also a reset() method that decrements the reference count and replaces the actual pointer with another one.
- If the call is “.reset(nullptr)” the result is that the reference count is decremented and if it reaches 0 it will delete de entire object
- If the call is “.reset(<new_raw_pointer>)” the original reference count for the initial pointer is decremented. If it reaches 0, the original pointer is deleted. Then, a new pointer is set into the shared_pointer and the reference count is set to 1.

shared_ptr

- Let's see an example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    shared_ptr<Test> t2 = t1;
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x, t1.use_count(), t1.get());
    printf("t2 = {x=%d, Count=%d, Ptr=%p}\n", t2->x, t2.use_count(), t2.get());
    t1.reset(new Test(20));
    printf("------ After Reset -------\n");
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x, t1.use_count(), t1.get());
    printf("t2 = {x=%d, Count=%d, Ptr=%p}\n", t2->x, t2.use_count(), t2.get());
    return 0;
}
```

```
Output:
CTOR(x=10)
t1 = {x=10, Count=2, Ptr=0000024FAED14330}
t2 = {x=10, Count=2, Ptr=0000024FAED14330}
CTOR(x=20)
------ After Reset -------
t1 = {x=20, Count=1, Ptr=0000024FAED07B80}
t2 = {x=10, Count=1, Ptr=0000024FAED14330}
DTOR(x=10)
DTOR(x=20)
```

shared_ptr

- Let's see an example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    shared_ptr<Test> t2 = t1;
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x, t1.use_count(), t1.get());
    printf("t2 = {x=%d, Count=%d, Ptr=%p}\n", t2->x, t2.use_count(), t2.get());
    t1.reset(new Test(20));
    printf("------ After Reset -------\n");
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x,
    printf("t2 = {x=%d, Count=%d, Ptr=%p}\n", t2->x,
    return 0;
}
```

```
Output:
CTOR(x=10)
t1 = {x=10, Count=2, Ptr=0000024FAED14330}
t2 = {x=10, Count=2, Ptr=0000024FAED14330}
CTOR(x=20)
------ After Reset -------
t1 = {x=20, Count=1, Ptr=0000024FAED07B80}
t2 = {x=10, Count=1, Ptr=0000024FAED14330}
DTOR(x=10)
DTOR(x=20)
```

Notice that both **t1** and **t2** refer to the same memory location (see Ptr) and as such have the same count (2)

shared_ptr

- Let's see an example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    shared_ptr<Test> t2 = t1;
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x, t1.use_count(), t1.get());
    printf("t2 = {x=%d, Count=%d, Ptr=%p}\n", t2->x, t2.use_count(), t2.get());
    t1.reset(new Test(20));
    printf("------ After Reset -------\n");
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x, t1.use_count(), t1.get());
    printf("t2 = {x=%d, Count
    return 0;
}
```

```
Output:
CTOR(x=10)
t1 = {x=10, Count=2, Ptr=0000024FAED14330}
t2 = {x=10, Count=2, Ptr=0000024FAED14330}
CTOR(x=20)
------ After Reset -------
t1 = {x=20, Count=1, Ptr=0000024FAED07B80}
t2 = {x=10, Count=1, Ptr=0000024FAED14330}
DTOR(x=10)
DTOR(x=20)
```

The CTOR for the second object is being called (with the value 20)

shared_ptr

- Let's see an example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    shared_ptr<Test> t2 = t1;
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x, t1.use_co
    printf("t2 = {x=%d, Count=%d, Ptr=%p}\n", t2->x, t2.use_co
    t1.reset(new Test(20));
    printf("------ After Reset -------\n");
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x, t1.use_co
    printf("t2 = {x=%d, Count=%d, Ptr=%p}\n", t2->x, t2.use_count(), t2.get());
    return 0;
}
```

```
Output:
CTOR(x=10)
t1 = {x=10, Count=2, Ptr=0000024FAED14330}
t2 = {x=10, Count=2, Ptr=0000024FAED14330}
CTOR(x=20)
------ After Reset -------
t1 = {x=20, Count=1, Ptr=0000024FAED07B80}
t2 = {x=10, Count=1, Ptr=0000024FAED14330}
DTOR(x=10)
DTOR(x=20)
```

After t1.reset(), t1 and t2 point to a different memory location. Also t2 reference count was decreased from 2 to 1

shared_ptr

- Let's see an example:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
                                                    t1.get());
                                                    et());
    printf("t1 = {x=%d, Count=%d, Ptr=%p}\n", t1->x, t1.use_count(), t1.get());
    printf("t2 = {x=%d, Count=%d, Ptr=%p}\n", t2->x, t2.use_count(), t2.get());
    return 0;
}
```

```
Output:
CTOR(x=10)
t1 = {x=10, Count=2, Ptr=0000024FAED14330}
t2 = {x=10, Count=2, Ptr=0000024FAED14330}
CTOR(x=20)
------ After Reset -------
t1 = {x=20, Count=1, Ptr=0000024FAED07B80}
t2 = {x=10, Count=1, Ptr=0000024FAED14330}
DTOR(x=10)
DTOR(x=20)
```

When the scope of t1 and t2 ends, they will decrement the count. Both of them will reach count 0 and will call the destructor.

shared_ptr

- You can also use unique() method to validate that the reference count is 1. This method was removed (considered deprecated) in C++17 but it is available in C++14.

App.cpp (run on x64)

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
    int x;
    Test(int v) : x(v) { cout << "CTOR(x=" << x << ")\n"; }
    ~Test() { cout << "DTOR(x=" << x << ")\n"; }
};
int main() {
    shared_ptr<Test> t1 = make_shared<Test>(10);
    cout << "T1 is unique: " << t1.unique() << endl;
    {
        cout << "  | Start scope of T2" << endl;
        shared_ptr<Test> t2 = t1;
        cout << "  | T1 is unique: " << t1.unique() << endl;
        cout << "  | End scope of T2" << endl;
    }
    cout << "T1 is unique: " << t1.unique() << endl;
    return 0;
}
```

Output:

```
CTOR(x=10)
T1 is unique: 1
  | Start scope of T2
  | T1 is unique: 0
  | End scope of T2
T1 is unique: 1
DTOR(x=10)
```

weak_ptr

- std::weak_ptr is a C++ pointer that encapsulates a regular (raw) pointer. When a weak_ptr is destroyed it will not free the memory it holds (it does not own the memory).

- To create a **weak_ptr** you have to use a shared_ptr with a method .lock() . You can not create a weak_ptr directly.

- weak_ptr are important to avoid circular references.

weak_ptr

- weak_ptr also takes more memory than a regular pointer:

App.cpp (run on x64)

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    cout << "Weak  : " << sizeof(weak_ptr<int>) << endl;
    cout << "Shared: " << sizeof(shared_ptr<int>) << endl;
    cout << "Unique: " << sizeof(unique_ptr<int>) << endl;
    return 0;
}
```

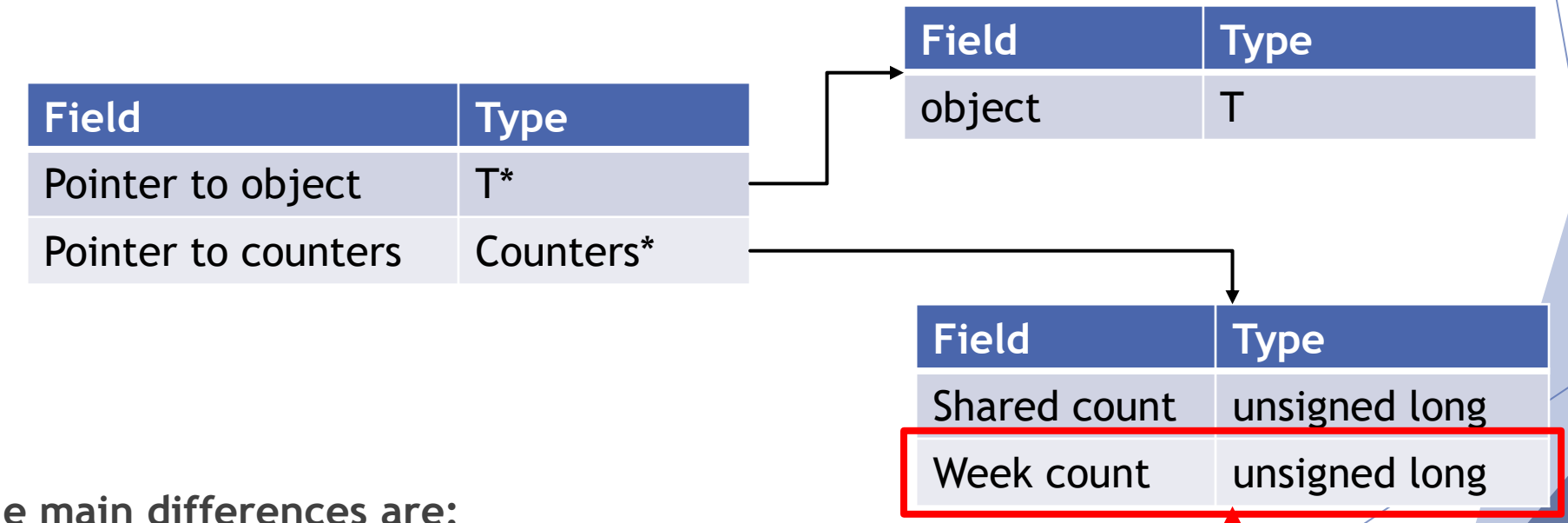
Output:

```
Weak  : 16
Shared: 16
Unique: 8
```

- This is because a weak_ptr needs to hold the reference count as well (as part of the object).

weak_ptr

- A simplified version on how a **weak_ptr** looks in memory is similar to the one from **shared_ptr**.



- **The main differences are:**
  1. **weak_ptr does not free the memory of the object T when deleted**
  2. **weak_ptr increments and decrement the week count**

weak_ptr

- Let's analyze the following example:

App.cpp

```
#include <iostream>
#include <memory>
using namespace std;
struct B;
struct A {
    shared_ptr<B> obj;
    A() { cout << "CTOR-A\n"; }
    ~A() { cout << "DTOR-A\n"; }
};
struct B {
    shared_ptr<A> obj;
    B() { cout << "CTOR-B\n"; }
    ~B() { cout << "DTOR-B\n"; }
};
```

```
int main() {
    shared_ptr<A> a = make_shared<A>();
    shared_ptr<B> b = make_shared<B>();
    a->obj = b;
    b->obj = a;
    return 0;
}
```

```
Output:
CTOR-A
CTOR-B
```

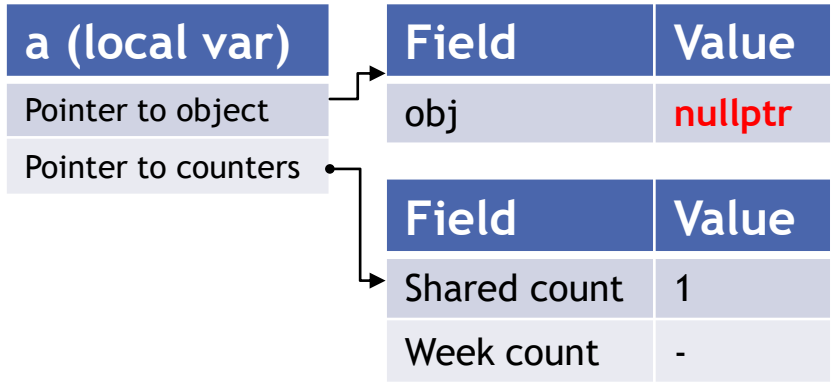
- Notice that the destructor was not called (meaning that we have a memory leak problem !!!)

weak_ptr

- Let’s analyze the execution in this case:

App.cpp

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

struct B;
struct A {…};
struct B {…};

int main() {
    shared_ptr<A> a = make_shared<A>();
    shared_ptr<B> b = make_shared<B>();
    a->obj = b;
    b->obj = a;
    return 0;
}
```

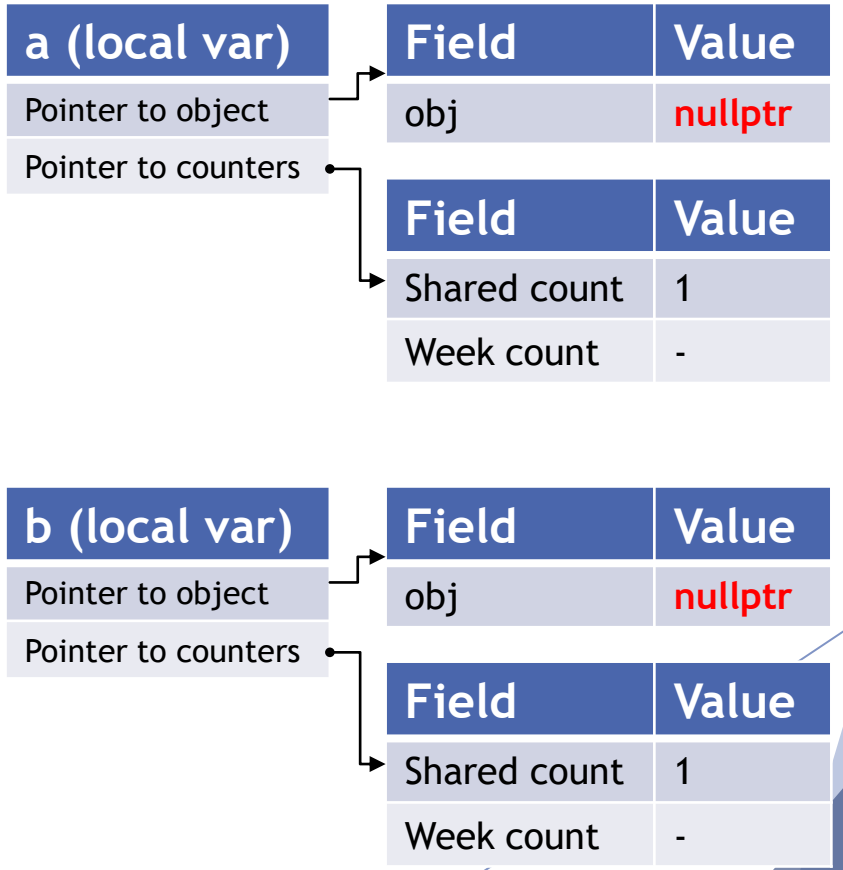


weak_ptr

- Let’s analyze the execution in this case:

App.cpp

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

struct B;
struct A {…};
struct B {…};

int main() {
    shared_ptr<A> a = make_shared<A>();
    shared_ptr<B> b = make_shared<B>();
    a->obj = b;
    b->obj = a;
    return 0;
}
```

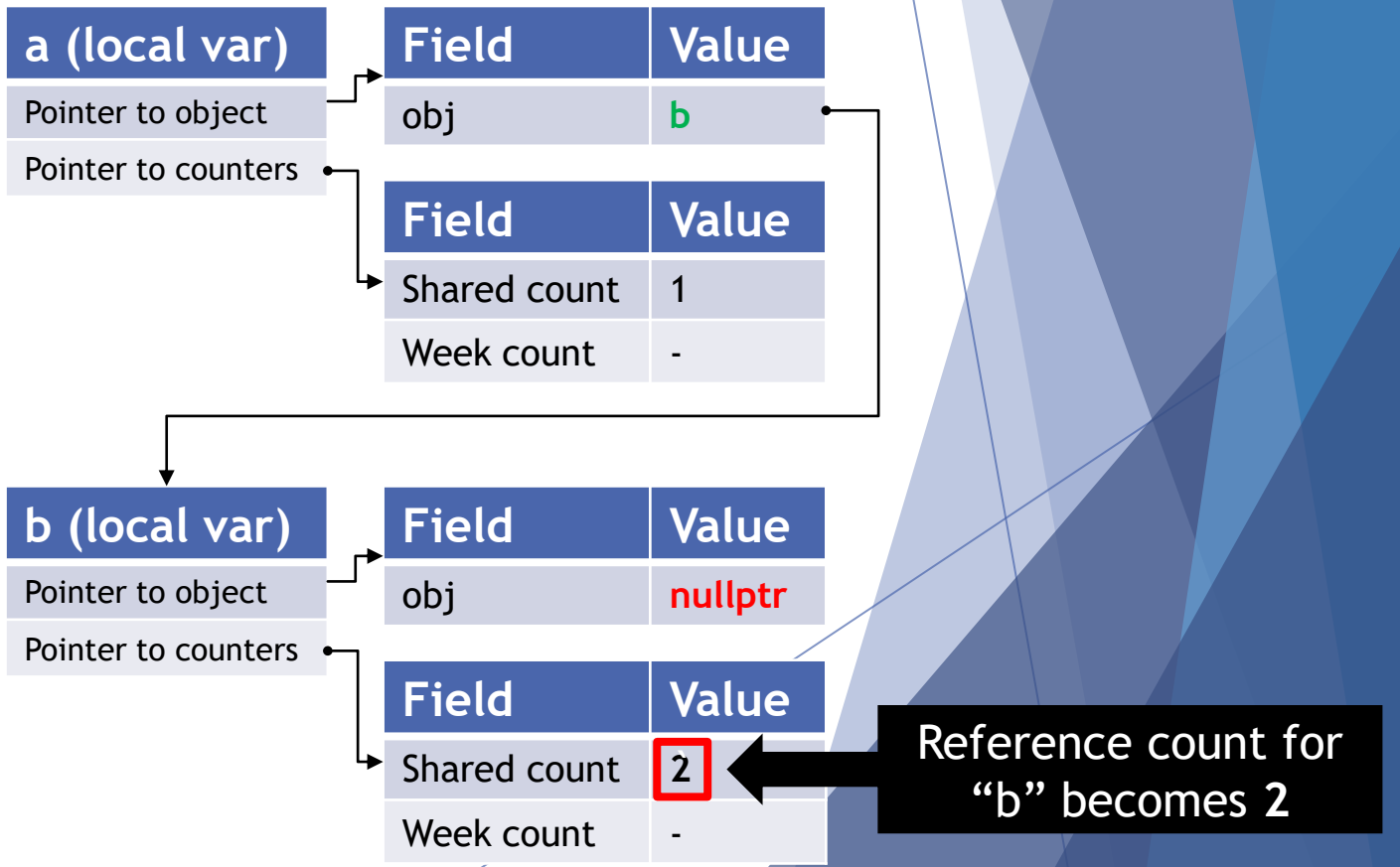


weak_ptr

- Let’s analyze the execution in this case:

App.cpp

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

struct B;
struct A {…};
struct B {…};

int main() {
    shared_ptr<A> a = make_shared<A>();
    shared_ptr<B> b = make_shared<B>();
    a->obj = b;
    b->obj = a;
    return 0;
}
```

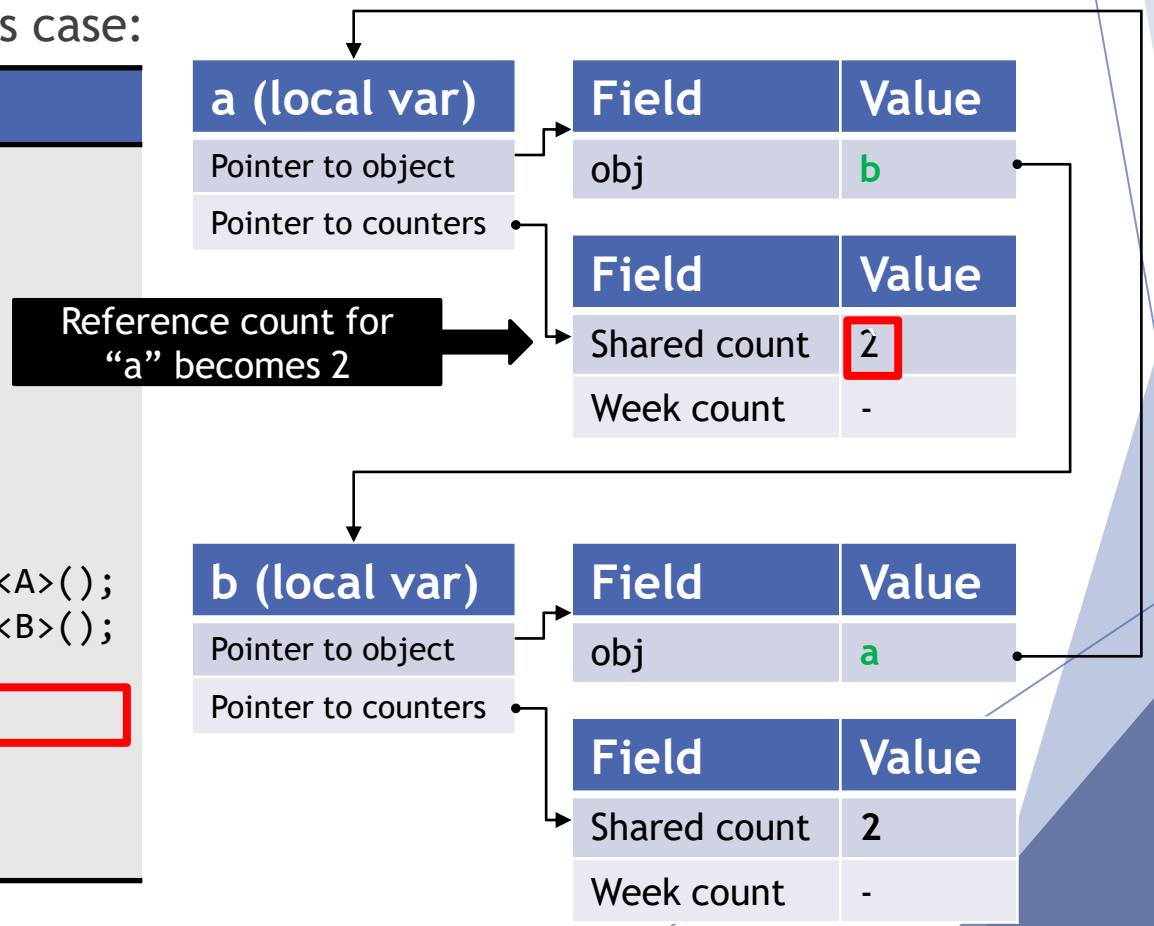


weak_ptr

- Let's analyze the execution in this case:

App.cpp

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

struct B;
struct A {…};
struct B {…};

int main() {
    shared_ptr<A> a = make_shared<A>();
    shared_ptr<B> b = make_shared<B>();
    a->obj = b;
    b->obj = a;
    return 0;
}
```



weak_ptr

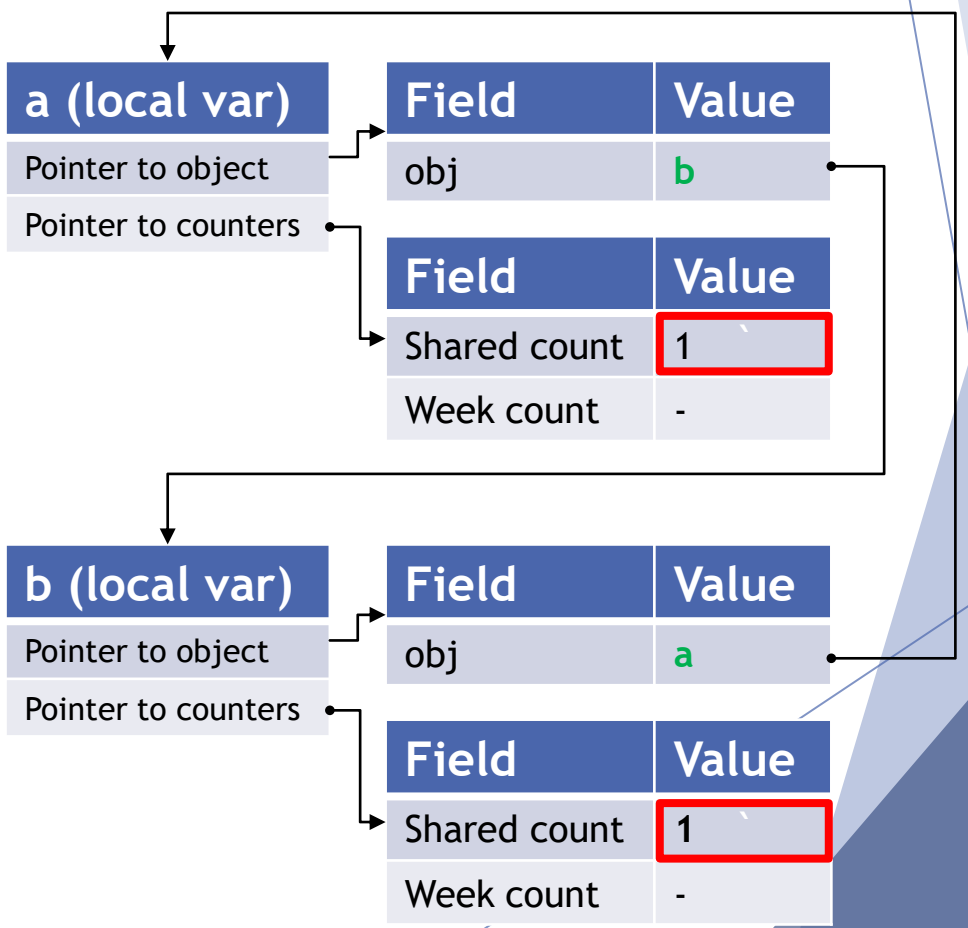
- Let's analyze the execution in this case:

App.cpp

```
#include <iostream>
#include <memory>

    b->obj = a;
    return 0;
}
```

Destructors are being called for "a" and "b":

1. In case of "a", reference count is 2, when decremented becomes 1 and as such the memory allocated by variable "a" will not be freed (as there is another shared_ptr (the one from "b") that still exists)
2. Similar for "b" - reference count becomes 1 after decrementation so the inner object will not be freed.



weak_ptr

- The solution is to use a weak_ptr on one of the classes (e.g., B):

App.cpp

```
#include <iostream>
#include <memory>
using namespace std;
struct B;
struct A {
    shared_ptr<B> obj;
    A() { cout << "CTOR-A\n"; }
    ~A() { cout << "DTOR-A\n"; }
};
struct B {
    weak_ptr<A> obj;
    B() { cout << "CTOR-B\n"; }
    ~B() { cout << "DTOR-B\n"; }
};
```

```
int main() {
    shared_ptr<A> a = make_shared<A>();
    shared_ptr<B> b = make_shared<B>();
    a->obj = b;
    b->obj = a;
    return 0;
}
```

```
Output:
CTOR-A
CTOR-B
DTOR-A
DTOR-B
```

- Notice that the destructor is called correctly for both “a” and “b”

weak_ptr

- The .lock() will not return a valid shared_ptr if the inner pointer has been freed.

App.cpp

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

int main() {
    weak_ptr<int> w;
    {
        shared_ptr<int> i1 = make_shared<int>(10);
        w = i1;
    }
    if (shared_ptr<int> i3 = w.lock()) {
        cout << "Value = " << *i3 << " Count=" << i3.use_count() << endl;
    }
    else {
        cout << "shared_ptr is no longer valid !" << endl;
    };
    return 0;
}
```

Output:

```
shared_ptr is no longer valid !"
```

"i1" scope

In this case, the scope of "i1" ends but not until it sets up the "w" to point to its inner reference block & structures.

weak_ptr

- The .lock() will not return a valid shared_ptr if the inner pointer has been freed.

App.cpp

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

int main() {
    weak_ptr<int> w;
    {
        shared_ptr<int> i1 = make_shared<int>(10);
        w = i1;
    }
    if (shared_ptr<int> i3 = w.lock()) {
        cout << "Value = " << *i3 << " Count="
    }
    else {
        cout << "shared_ptr is no longer valid !" << endl;
    };
    return 0;
}
```

Output:

```
shared_ptr is no longer valid !
```

This code will not return a valid shared_ptr as i1 reference count has already reach the value 0 and as such the inner pointer would be freed.

weak_ptr

- The .lock() will not return a valid shared_ptr if the inner pointer has been freed.

App.cpp

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

int main() {
    weak_ptr<int> w;
    //{
        shared_ptr<int> i1 = make_shared<int>(10);
        w = i1;
    //}
    if (shared_ptr<int> i3 = w.lock()) {
        cout << "Value = " << *i3 << " Count=" << i3.use_count() << endl;
    }
    else {
        cout << "shared_ptr is no longer valid !" << endl;
    };
    return 0;
}
```

Output:

```
Value = 10 Count=2
```

Notice that we have commented “{“ and “}”, thus making the scope of “i1” the entire main function. As a direct result, w.lock() will return a valid shared_ptr/

weak_ptr

- weak_ptr also has a method .use_count () that returns the reference count of the shared_ptr as well as a method .reset() that can be used to clear the inner pointer.

App.cpp (run on x64)

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

int main() {
    shared_ptr<int> i1 = make_shared<int>(10);
    weak_ptr<int> w = i1;
    cout << "Value = " << *i1 << " Count=" << i1.use_count() << "," << w.use_count() << endl;
    w.reset();
    cout << "Value = " << *i1 << " Count=" << i1.use_count() << "," << w.use_count() << endl;
    return 0;
}
```

```
Output:
Value = 10 Count=1,1
Value = 10 Count=1,0
```

Smart pointers

General observations:

- **Don’t use .get() method for shared_ptr or unique_ptr** (it is more likely that this will create either a dangling pointer or a crash due to a double free scenario)
- **When using shared_ptr with graphs / double linked-lists or circular lists** make sure that you always check to see if the destructor is called (and make sure that you use weak_ptr to avoid circular references)

Q & A