POO

Curs-7

Gavrilut Dragos

- Containeri asociativi
- IOS
- Clasa String

STL (containere asociative - pair)

- “pair”este un template care contine doua valori (de doua tipuri diferite)
- Pentru utilzare “**#include <utility>**”
- Are definit operatorul= pentru a putea fi comparate doua obiecte de tipul pair
- Este utilizat intern pentru in containerele asociative
- Definirea template-ului **pair** se face in felul urmator:

App.cpp

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

STL (containere asociative - map)

- Map este un container care pastreaza perechi de forma (cheie/valoare) accesul la campul **valoare** putand sa se faca prin **cheie**
- Pentru utilzare "`#include <map>`"
- Campul cheie este constant in sensul ca o daca ce s-a adaugat intr-un map o pereche cheie/valoare, cheia nu se mai poate modifica ci doar valoarea. Se pot adauga alte perechi sau se poate sterge o pereche deja existent
- Definirea template-ului map se face in felul urmator:

App.cpp

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

STL (containere asociative - map)

- Un exemplu de utilizare map :

App.cpp

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

Output

```
Nota Ionescu  = 9
Numar intrari = 3
Note[Popescu]=10
Note[Ionescu]=9
Note[Marin]=7
Nota Ionescu  = 0 (x=0)
Numar intrari = 3
```

STL (containere asociative - map)

- Un exemplu de utilizare map :

App.cpp

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

Cheia “Ionescu” nu exista in acest punct. Pentru ca nu exista, se creaza una nou, iar valoarea este instantiata prin constructorul default (care pentru tipul **int** faca ca valoarea sa fie 0)

STL (containere asociative - map)

- Functiile suportate de containerul map sunt urmatoarele:

Functie/operator
Asignare (**operator=**)
Access la valori si adaugare (**operator[]** si functiile **at, insert**)
Stergere (functiile **erase** , **clear**)
Cautare in sir (functia **find**)
Iteratori (**begin**, **end**, **rbegin**, **rend**, **cbegin, cend, crbegin, crend** (ultiimii 4 din C++11))
Informatii (**size**, **empty**, **max_size**)

STL (containere asociative - map)

- Accesul la elemente se face prin (operatorul [] si functiile **at** si **find**)

App.cpp

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

App.cpp

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

App.cpp

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

STL (containere asociative - map)

- Accesul la elemente se face prin (operatorul [] si functiile **at** si **find**)

App.cpp

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

App.cpp (Ionescu nu exista)

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

App.cpp (Ionescu nu exista)

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

STL (containere asociative - map)

- Verificarea daca un element este in map se poate face cu functiile **find** si **count**

App.cpp

```
void main(void)
{
	map<const char*, int> Note;
	Note["Popescu"] = 10;
	if (Note.find("Ionescu")==Note.cend())
		printf("Ionescu nu este in lista de note !");
}
```

App.cpp

```
void main(void)
{
	map<const char*, int> Note;
	Note["Popescu"] = 10;
	if (Note.count("Ionescu")==0)
		printf("Ionescu nu este in lista de note !");
}
```

STL (containere asociative - map)

- Datele intr-un container map sunt stocate intr-un red-black tree
- Acest lucru reprezinta un compromise intre timpul de access si insertie a elementelor, si memoria alocata unui astfel de container
- In functie de ce operatie ne intereseaza, e posibil ca sa nu fie cea mai buna solutie (de exemplu daca discutam doar de multi citiri si de putine inserturi sunt implementari mult mai eficiente)
- Facem urmatorul experiment – acelasi algoritm e scris folosind map si un vector simplu si evaluam timpul de insertie.
- Experimentul se repeat de 10 ori pentru fiecare algoritm si masuram timpii de executie in milisecunde pentru fiacre caz.

STL (containere asociative - map)

- Cei doi algoritmi utilizati sunt urmatorii:

App-1.cpp

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

App-2.cpp

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

- Chiar daca este evident ca App-2 este mai efficient, trebuie tinut cont de coliziunile de hash care pot aparea. Pe cazul de mai sus, nu exista asa ceva pentru ca cheile sunt tot de tipul integer.

STL (containere asociative - map)

- Rezultate:

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

- Testeles-au realizatcu urmatoarelespecificatiisoftware sihardware:
  - 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 (containere asociative - multimap)

- “multimap” este un container similar cu “map”. Diferenta e ca o cheie poate sa contine mai multe valori
- Pentru utilzare “**#include <map>**”
- Accesul la elemente prin operatorul[] si functia at nu mai este posibil.
- Definirea template-ului multimap se face in felul urmator:

App.cpp

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

STL (containere asociative - multimap)

- Un exemplu de utilizare multimap :

App.cpp

```
void main(void)
{
	multimap<const char*, int> Note;
	Note.insert(pair<const char*, int>("Ionescu", 10));
	Note.insert(pair<const char*, int>("Ionescu", 8));
	Note.insert(pair<const char*, int>("Ionescu", 7));
	Note.insert(pair<const char*, int>("Popescu", 9));
	multimap<const char*, int>::iterator it;
	for (it = Note.begin(); it != Note.end(); it++)
	{
		printf("%s [%d]\n", it->first,it->second);

	}
}
```

Output

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

STL (containere asociative - multimap)

- Un exemplu de utilizare multimap :

App.cpp

```
void main(void)
{
	multimap<const char*, int> Note;
	Note.insert(pair<const char*, int>("Ionescu", 10));
	Note.insert(pair<const char*, int>("Ionescu", 8));
	Note.insert(pair<const char*, int>("Ionescu", 7));
	Note.insert(pair<const char*, int>("Popescu", 9));
	Note.insert(pair<const char*, int>("Popescu", 6));
	Note.insert(pair<const char*, int>("Georgescu", 8));
	multimap<const char*, int>::iterator it;
	it = Note.begin();
	printf("%s->%d\n", it->first, it->second);
	it = Note.upper_bound(it->first);
	printf("%s->%d\n", it->first, it->second);
	it = Note.upper_bound(it->first);
	printf("%s->%d\n", it->first, it->second);
}
```

Output

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

STL (containere asociative - multimap)

- Un exemplu de utilizare multimap :

App.cpp

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

	for (it = Note.begin(); it != Note.end(); it = Note.upper_bound(it->first))
	{
		printf("Cheie unica: %s\n", it->first);

	}
}
```

Output

```
Cheie unica: Ionescu
Cheie unica: Popescu
Cheie unica: Georgescu
```

STL (containere asociative - multimap)

- Un exemplu de utilizare multimap :

App.cpp

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

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

Output

```
Note Ionescu:10,8,7
Note Popescu:9,6
Note Georgescu:8
```

STL (containere asociative - multimap)

- Un exemplu de utilizare multimap :

App.cpp

```
void main(void)
{
	multimap<const char*, int> Note;
	Note.insert(pair<const char*, int>("Ionescu", 10));
	Note.insert(pair<const char*, int>("Ionescu", 8));
	Note.insert(pair<const char*, int>("Ionescu", 7));
	Note.insert(pair<const char*, int>("Popescu", 9));
	Note.insert(pair<const char*, int>("Popescu", 6));
	Note.insert(pair<const char*, int>("Georgescu", 8));
	multimap<const char*, int>::iterator it,it2;

	it = Note.find("Popescu");
	it2 = Note.upper_bound(it->first);
	printf("Notele lui Popescu: ");
	while (it != it2)
	{
		printf("%d,", it->second);
		it++;
	}
}
```

Output

```
Notele lui Popescu: 9,6
```

STL (containere asociative - multimap)

- Un exemplu de utilizare multimap :

App.cpp

```
void main(void)
{
	multimap<const char*, int> Note;
	Note.insert(pair<const char*, int>("Ionescu", 10));
	Note.insert(pair<const char*, int>("Ionescu", 8));
	Note.insert(pair<const char*, int>("Ionescu", 7));
	Note.insert(pair<const char*, int>("Popescu", 9));
	Note.insert(pair<const char*, int>("Popescu", 6));
	Note.insert(pair<const char*, int>("Georgescu", 8));
	multimap<const char*, int>::iterator it,it2;

	for (it = Note.begin(); it != Note.end(); it = Note.upper_bound(it->first))
	{
		printf("%s are %d note\n", it->first,Note.count(it->first));
	}
}
```

Output

```
Ionescu are 3 note
Popescu are 2 note
Georgescu are 1 note
```

STL (containere asociative - multimap)

- Un exemplu de utilizare multimap :

App.cpp

```
void main(void)
{
	multimap<const char*, int> Note;
	Note.insert(pair<const char*, int>("Ionescu", 10));
	Note.insert(pair<const char*, int>("Ionescu", 8));
	Note.insert(pair<const char*, int>("Ionescu", 7));
	Note.insert(pair<const char*, int>("Popescu", 9));
	Note.insert(pair<const char*, int>("Popescu", 6));
	Note.insert(pair<const char*, int>("Georgescu", 8));
	multimap<const char*, int>::iterator it,it2;

	it = Note.find("Ionescu");
	it++;
	Note.erase(it); // stergem nota 8 de la Ionescu
	for (it = Note.begin(); it != Note.end(); it++)
	{
		printf("%s [%d]\n", it->first, it->second);
	}
}
```

Output

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

STL (containere asociative - multimap)

- Un exemplu de utilizare multimap :

App.cpp

```
void main(void)
{
	multimap<const char*, int> Note;
	Note.insert(pair<const char*, int>("Ionescu", 10));
	Note.insert(pair<const char*, int>("Ionescu", 8));
	Note.insert(pair<const char*, int>("Ionescu", 7));
	Note.insert(pair<const char*, int>("Popescu", 9));
	Note.insert(pair<const char*, int>("Popescu", 6));
	Note.insert(pair<const char*, int>("Georgescu", 8));
	multimap<const char*, int>::iterator it,it2;

	if (Note.find("Ionescu") != Note.cend())
		printf("Ionescu este in catalog !\n");
	if (Note.find("Marin") == Note.cend())
		printf("Marin NU este in catalog !\n");
}
```

Output

```
Ionescu este in catalog !
Marin NU este in catalog !
```

STL (containere asociative - multimap)

- Functiile suportate de containerul multimap sunt urmatoarele:

Functie/operator
Asignare (**operator=**)
Adaugare (**insert**)
Stergere (functiile **erase** , **clear**)
Access la elemente (functia **find**)
Iteratori (**begin**, **end**, **rbegin**, **rend**, **cbegin, cend, crbegin, crend** (ultiimii 4 din C++11))
Informatii (**size**, **empty**, **max_size**)
Functii speciale (**upper_bound** si **lower_bound** – pentru a accesa intervalele in care se gasesc elemente cu aceasi cheie) sau **equal_range** pentru a obtine un range pentru toate elementele stocate pentru o anumita cheie

STL (containere asociative - set)

- “set” este un container ce permita pastrarea doar a unei singure valori intr-o lista
- Pentru utilzare “**#include <set>**”
- Definirea template-ului **set** se face in felul urmator:

App.cpp

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

STL (containere asociative - set)

- Un exemplu de utilizare 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 (containere asociative - set)

- Un exemplu de utilizare 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 (containere asociative - set)

- “set” functioneaza tinand toate datele sortate.
- Acest lucru aduce un penalty de performanta la utilizarea lui.

App-3.cpp

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

- Daca refacem experimental precedent o sa obtinem urmatoarele valori:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Medie
App-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
App-2	0	16	0	0	16	0	15	0	16	0	**6.3**
App-3	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**

STL (containere asociative - set)

- Functiile suportate de containerul **set** sunt urmatoarele:

Functie/operator
Asignare (**operator=**)
Adaugare (**insert**)
Stergere (functiile **erase** , **clear**)
Access la elemente (functia **find**)
Iteratori (**begin**, **end**, **rbegin**, **rend**, **cbegin, cend, crbegin, crend** (ultiimii 4 din C++11))
Informatii (**size**, **empty**, **max_size**)

STL (containere asociative - multiset)

- “multiset” este un container ce permita pastrarea a mai multor valori intr-o lista
- Pentru utilzare “**#include <set>**”
- Definirea template-ului **multiset** se face in felul urmator:

App.cpp

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

STL (containere asociative - multiset)

- Un exemplu de utilizare 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 (containere asociative - multiset)

- Functiile suportate de containerul multi**set** sunt urmatoarele:

Functie/operator
Asignare (**operator=**)
Adaugare (**insert**)
Stergere (functiile **erase** , **clear**)
Access la elemente (functia **find**)
Iteratori (**begin**, **end**, **rbegin**, **rend**, **cbegin, cend, crbegin, crend** (ultiimii 4 din C++11))
Informatii (**size**, **empty**, **max_size**)
Functii speciale (**upper_bound** si **lower_bound** – pentru a accesa intervalele in care se gasesc elemente cu aceasi cheie) sau **equal_range** pentru a obtine un range pentru toate elementele stocate pentru o anumita cheie

STL (containere asociative – unordered_map)

- Datele intr-un container unordered_map sunt stocate intr-un hash-table
- A fost introdus in C++11
- Pentru utilzare "**#include <unordered_map>**"
- Suporta aceleasi functii ca si **map** la care se mai adauga si functiile pentru controlul bucket-urilor
- Definirea template-ului **unordered_map** se face in felul urmator:

App.cpp

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

STL (containere asociative – unordered_map)

- Functionare tabele de dispersie

Popescu

Ionescu

Georgescu

Functie de hash (transforma un sir de caractere intr-un index)

Index	Value

STL (containere asociative – unordered_map)

- Functionare tabele de dispersie

Popescu

Ionescu

Georgescu

Consideram ca functia de hash-ing este urmatoarea:

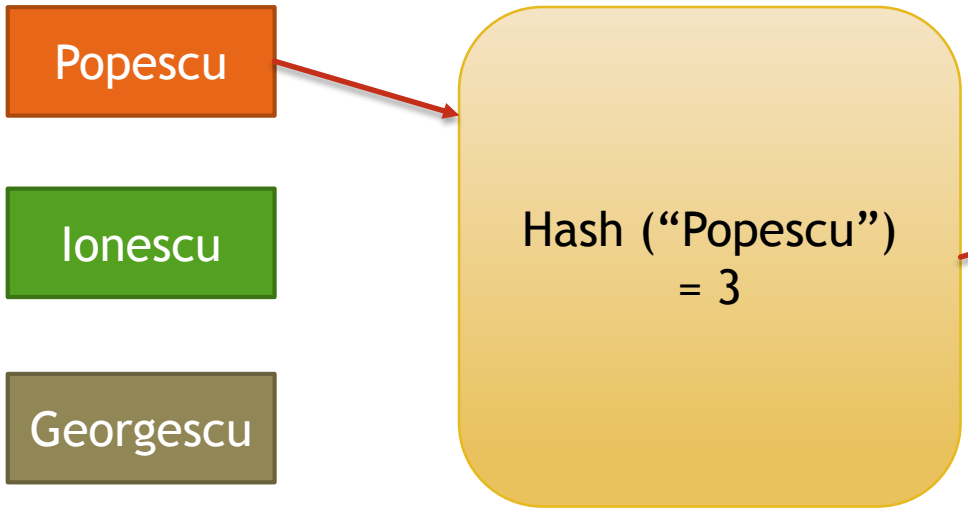
HashFunction

```
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 (containere asociative – unordered_map)

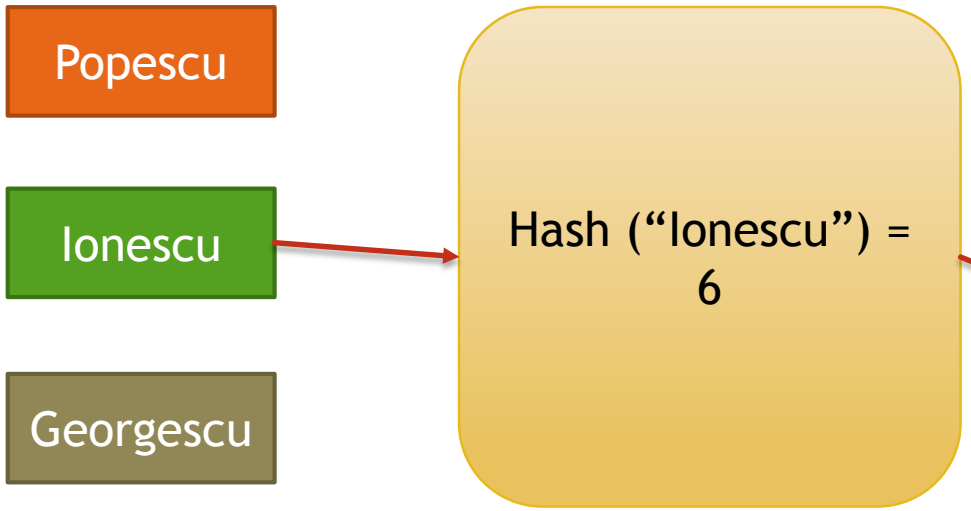
- Functionare tabele de dispersie



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 (containere asociative – unordered_map)

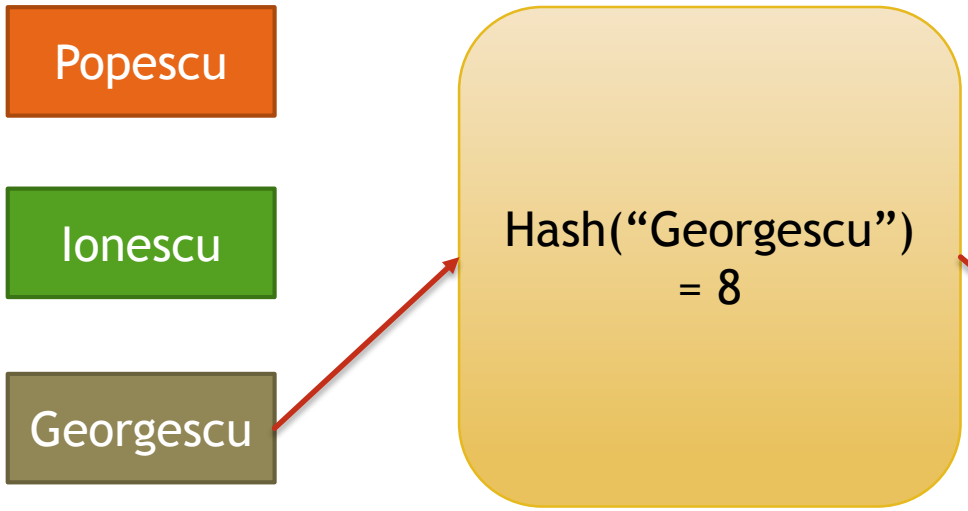
- Functionare tabele de dispersie



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 (containere asociative – unordered_map)

- Functionare tabele de dispersie



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 (containere asociative – unordered_map)

- Fie urmatorul cod:

App-4.cpp

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

- Daca refacem experimental precedent o sa obtinem urmatoarele valori:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Medie
App-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
App-2	0	16	0	0	16	0	15	0	16	0	**6.3**
App-3	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**
App-4	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	**15310**

STL (containere asociative – unordered_map)

- Fie urmatorul cod:

App-5.cpp

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

- Daca refacem experimental precedent o sa obtinem urmatoarele valori:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Medie
App-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
App-2	0	16	0	0	16	0	15	0	16	0	**6.3**
App-3	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**
App-4	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	**15310**
App-5	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	**10006**

STL (containere asociative – unordered_set)

- Containerul **unordered_set** este similar cu un container set doar ca datele nu sunt sortate in memorie
- A fost introdus in C++11
- Pentru utilzare “**#include <unordered_set>**”
- Suporta aceleasi functii ca si **set** + functiile pentru controlul bucket-urilor
- Definirea template-ului **unordered_set** se face in felul urmator:

App.cpp

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

STL (containere asociative – unordered_set)

- Fie urmatorul cod:

App-6.cpp

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

- Daca refacem experimental precedent o sa obtinem urmatoarele valori:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Medie
App-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
App-2	0	16	0	0	16	0	15	0	16	0	**6.3**
App-3	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**
App-4	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	**15310**
App-5	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	**10006**
App-6	12140	11625	12047	11984	12109	12078	11609	11578	11782	11672	**11862**

STL (containere asociative – unordered_set)

- Si variant cu reserve:

App-7.cpp

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

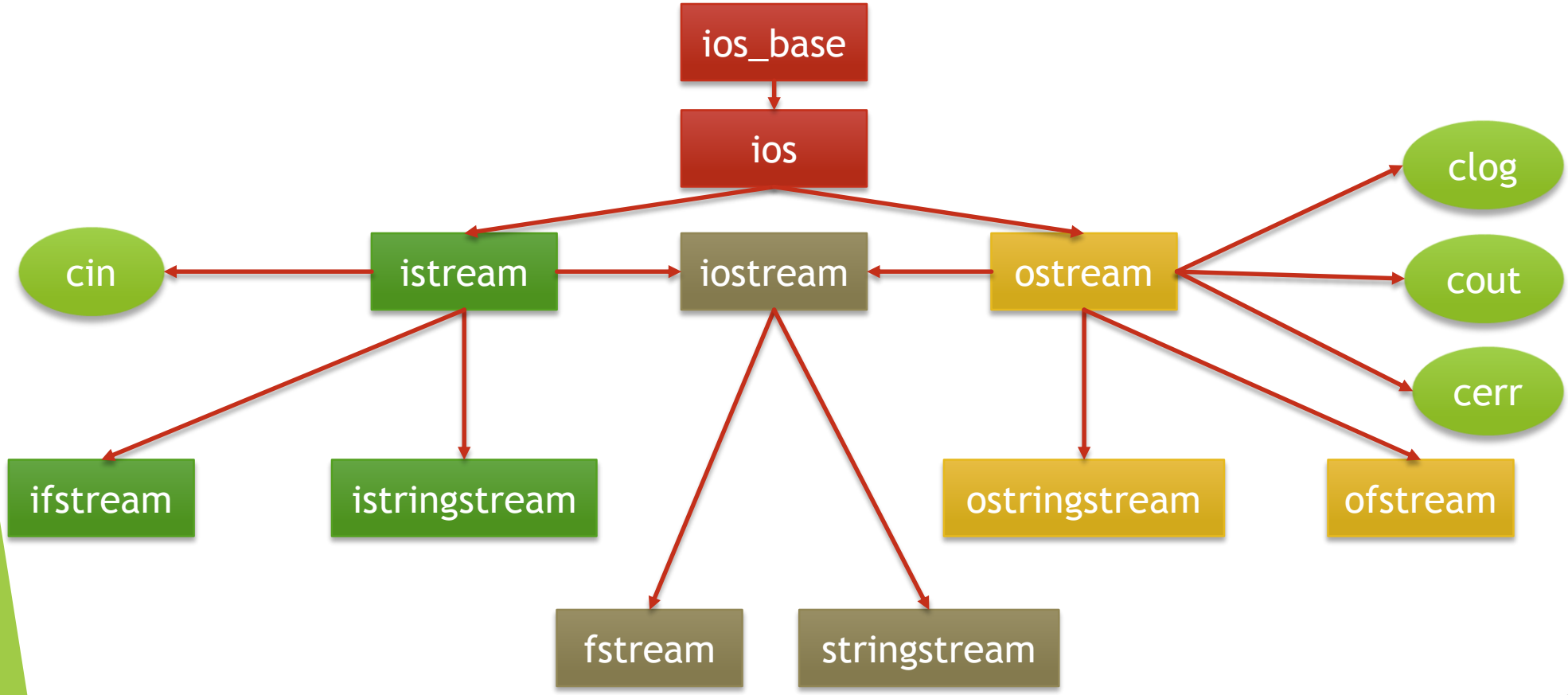
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Medie
App-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	**19901**
App-2	0	16	0	0	16	0	15	0	16	0	**6.3**
App-3	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	**16158**
App-4	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	**15310**
App-5	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	**10006**
App-6	12140	11625	12047	11984	12109	12078	11609	11578	11782	11672	**11862**
App-7	6516	6328	6875	6844	6812	6453	6453	6531	6500	6515	**6582**

STL (containere associative)

- Pe langa containerele presentate, mai exista doua containere associative:
  - unordered_multimap
  - unordered_multiset
- Functioneaza la fel ca si un multimap sau un multiset (diferenta e ca folosesc tabele de dispersie si nu un arbore sortat)
- Trebuie specificat ca utilizarea unuia dintre containerele **map** sau **unordered_map** trebuie facuta tinand cont de cat de rapid vrem sa mearga containerul dar si cat de multa memorie avem disponibila.

- Containeri asociativi
- IOS
- Clasa String

STL (IOS)


ios_base
ios
clog
cout
cerr
cin
istream
iostream
ostream
ifstream
istringstream
ostringstream
ofstream
fstream
stringstream

STL(IOS)

- Din clasele prezentate in ierarhia de mai sus, cele mai folosite sunt istream, ostream si iostream.
- Clasele permit access de tipul I/O la diverse stream-uri (cea mai cunoascuta utilizare fiind accesul la sistemul de fisiere)
- O alta utilizare o reprezinta obiectele definite cin si cout care pot fi folosite pentru a scrie / citi de la terminal
- 2 operatori sunt suprascrisi pentru aceste clase (operator>> si operator<<) reprezentand operatiile de intrare respective iesire din stream.
- La acesti doi operatori s-au adaugat si o serie de manipulatori (elemente care pot schimba modul in care se proceseaza datele care urmeaza dupa ei).

STL (IOS - manipulatori

manipulator	Utilzare
endl	Adauga un terminator de linie (“\n” , “\r\n”, etc) si face flush
ends	Adauga un ‘\0’ (NULL)
flush	Goleste cache-ul intern al stream-ului
dec	Numerele vor fi scrise in baza 10
hex	Numerele vor fi scrise in baza 16
oct	Numerele vor fi scrise in baza 8
ws	Ignora spatiile la intrare
showpoint	Afiseaza punctul zecimal si zerourile
noshowpoint	Nu afiseaza punctul zecimal sau zerourile
showpos	Adauga caracterul “+” in fata numerelor pozitive
noshowpos	Nu adauga caracterul “+” in fata numerelor pozitive

STL (IOS - manipulatori

manipulator	Utilzare
boolalpha	Valorile bool le afiseza cu “true” sau “false”
noboolalpha	Valorile bool le afiseza cu “1” sau “0”
scientific	Notatie stiintifica pentru numere float sau double
fixed	Notatie in punct fix pentru numere
left	Aliniere la stanga
right	Aliniere la dreapta
setfill(char)	Seteaza cu ce character sa se faca fill-ul (altul decat spatiu)
setprecision(n)	Seteaza ce precizie o sa avem pentru numerele reale
setbase(b)	Seteaza baza in care se va afisa

- Containeri asociativi
- IOS
- Clasa String

STL (basic_string)

- Un template definit sa ofere cele mai comune operatii pentru stringuri
- Definita este de felul urmator:

App.cpp

```
template <class CharacterType, class traits = char_traits<CharacterType>>
class basic_string
{
...
}
```

- Cel mai comun obiect derivate obtinut din acest template este **string** si wstring

App.cpp

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

- Alte obiecte introdu-se in Cx11 bazate pe acelasi template sunt:

App.cpp

```
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
```

STL (basic_string)

- “char_traits” este un template care ofera o lista de operatii de lucru pe siruri de caractere (nu neaaparat character de tipul **char**) si care va fi folosit de basic_string pentru a face anumite operatii
- Principalele functii definite in char_traits sunt urmatoarele:

Definitie	Functionalitate
`static bool eq (CType c1, CType c2)`	Returneaza **true** daca c1 si c2 sunt egale
`static bool eq (CType c1, CType c2)`	Returneaza **true** daca c1 este mai mic ca c2
`static size_t length (const CType* sir);`	Returneaza dimensiunea unui sir de caractere
`static void assign (CType& caracter, const CType& value)`	Asignare (caracter = value)
`static int compare (const CType* sir1, const CType* sirt, size_t n);`	Compara doua siruri de caractere. Returneaza 1 daca sir1 > sir2, 0 pentru egalitate si -1 pentru sir1<sir2
`static const char_type* find (const char_type* sir, size_t n, const char_type& car);`	Returneaza un pointer catre primul character egal cu **car** din **sir**.

STL (basic_string)

- Principalele functii definite in char_traits sunt urmatoarele:

Definitie	Functionalitate
`static char_type* move (char_type* dest, const char_type* src, size_t n);`	Muta continutul unui sir de caractere intre doua locatii
`static char_type* copy(char_type* dest, const char_type* src, size_t n);`	Copie continutul unui sir de caractere dintr-o locatie in alta
`static int_type eof()`	Returneaza o valoare pentru EOF (deobicei -1)

- “char_traits” are si specializari pentru <char>, <wchar> care folosesc functii rapide de genu memcpy, memmove, etc.
- In general (pentru utilizarea normal de stringuri) nu este necesara crearea unui nou obiect char_traits. Este insa util pentru cazurile in care dorim un comportament mai special (in special daca dorim ca anumite comparatii sau asignari sa le facem altfel – case insensitive, etc)

STL (basic_string)

- “basic_string” suporta diverse forme de initalizare la nivel de constructor:

App.cpp

```
void main(void)
{
	string s1("Astazi");
	string s2(s1+" am");
	string s3(s1, 3, 3);
	string s4("Azi am examen la matematica", 13);
	string s5(s4.substr(7,6));
	string s6(10, '1');

	printf("%s\n%s\n%s\n%s\n%s\n%s\n%s\n", s1.c_str(),
	                                       s2.c_str(),
	                                       s3.c_str(),
	                                       s4.c_str(),
	                                       s5.c_str(),
	                                       s6.c_str());
}
```

Output

```
Astazi
Astazi am
azi
Azi am examen
examen
1111111111
```

STL (basic_string)

- Functii / operatori suportate de obiectele derivate din template-ul basic_string:

Functie/operator
Asignare (**operator=**)
Apendare (**operator+=** si functiile **append** si **push_back**)
Inserare caractere (functia **insert**)
Access la caractere (**operator[]** si functiile **at** , **front** (C++11) , **back** (C++11))
Substringuri (functia **substr**)
Replace (functia **replace**)
Stergere caractere (functiile **erase** , **clear** , si **pop_back** (C++11))
Cautare in sir (functiile **find**, **rfind**, **find_first_of**, **find_last_of**, **find_first_not_of**, **find_last_not_of**)
Comparatii (**operator>** , **operator<** , **operator==** , **operator!=** , **operator>=** , **operator<=** si functia **compare**)
Iteratori (**begin**,**end**,**rbegin**,**rend**,**cbegin**,**cend**,**crbegin**,**crend** (ultiimii 4 din C++11))
Informatii (**size**, **length**, **empty**, **max_size**, **capacity**)

STL (basic_string)

- Un exemplu de lucru cu obiectele de tipul string:

App.cpp

```
void main(void)
{
	string s1;
	s1 += "Azi";
	printf("Size = %d\n", s1.length());
	s1 = s1 + " " + s1;
	printf("S1 = %s\n", s1.data());
	s1.erase(2, 4);
	printf("S1 = %s\n", s1.c_str());
	s1.insert(1, "_");
	s1.insert(3, "__");
	printf("S1 = %s\n", s1.c_str());
	s1.replace(s1.begin(), s1.begin() + 2, "123456");
	printf("S1 = %s\n", s1.c_str());
}
```

Output

```
Size = 3
S1 = Azi Azi
S1 = Azi
S1 = A_z__i
S1 = 123456z__i
```

STL (basic_string)

- Un exemplu de utilizare char_traits :

App.cpp

```
struct IgnoreCase : public char_traits<char> {
	static bool eq(char c1, char c2) {
		return (upper(c1)) == (upper(c2));
	}
	static bool lt(char c1, char c2) {
		return (upper(c1)) < (upper(c2));
	}
	static int compare(const char* s1, const char* s2, size_t n) {
		while (n>0)
		{
			char c1 = upper(*s1);
			char c2 = upper(*s2);
			if (c1 < c2) return -1;
			if (c1 > c2) return -1;
			s1++; s2++; n--;
		}
		return 0;
	}
};
void main(void)
{
	basic_string<char, IgnoreCase> s1("Salut");
	basic_string<char, IgnoreCase> s2("sAlUt");
	if (s1 == s2)
		printf("Siruri egale !");
}
```