Curs 4 – Templates

- Structuri de date implementate orientat obiect
  - Listă – implementată secvențial pe vector dinamic (Dynamic Array)
  - Iterator
  - Listă – implementată folosind o listă înlănțuită

- Șabloane (Templates)
- Diagrame UML de clase

Tipuri abstracte de date implementate folosind clase obiecte

- **Tip abstract de date**
  - sparare interfață de implementare
  - specificare abstractă (independent de reprezentare/implementare)
  - ascunderea detaliilor de implementare

- **Clasă**
  - header : conține doar declarații (interfața). Implementarea în fisier separat (cpp)
  - fiecare metodă specificată
  - folosind modificatori de acces se poate controla accesul la atribute (metode,variabile membre) , reprezentarea este protejată (nu poate fi accesat din afara clasei)

Listă – implementată secvențial pe vector dinamic (Dynamic Array)

Vector dinamic - tablou unidimensional, lungimea se modifică în timp

```
typedef int TElem;
/**
 * List implemented using a dynamic array data structure
 */
class DynamicArray {
public:
	/**
	 * Add an element to the dynamic array to the end of the array
	 * r - is a rational number
	 */
	void addE(TElem r);
	/**
	 * Delete the element from the given position
	 * poz - the position of the elem to be deleted, poz>=0;poz<size
	 * return the deleted element
	 */
	TElem deleteElem(int poz);

	/**
	 * Access the element from a given position
	 * poz - the position (poz>=0;poz<size)
	 */
	TElem get(int poz);
	/**
	 * Give the size of the array
	 * return the number of elements in the array
	 */
	int getSize();
private:
	TElem *elems;
	int capacity;
	int size;
	/**
	 * Create enough space to hold nrElem elements
	 * nrElems - the number of elements that we need to store
	 */
	void ensureCapacity(int nrElems);
```

Regula celor trei (rule of tree)

```
void testCopy() {
	DynamicArray ar1;
	ar1.addE(3);
	DynamicArray ar2 = ar1;
	ar2.addE(3);
	ar2.set(0, -1);
	printElems(&ar2);
	printElems(&ar1);
}
```

Daca o clasa gestioneaza (este responsabil de) o resursă (memorie heap, fisiere, etc) trebuie sa definească:

- constructor de copiere

```
DynamicArray::DynamicArray(const DynamicArray& d) {
	this->capacity = d.capacity;
	this->size = d.size;
	this->elems = new TElem[capacity];
	for (int i = 0; i < d.size; i++) {
		this->elems[i] = d.elems[i];
	}
}
```

- operatorul de atribuire

```
DynamicArray& DynamicArray::operator=(const DynamicArray& ot) {
	if (this == &ot) {
		return *this;// protect against self-assignment (a = a)
	}
	delete this->elems; //delete the allocated memory
	this->elems = new TElem[ot.capacity];
	for (int i = 0; i < ot.size; i++) {
		this->elems[i] = ot.elems[i];
	}
	this->capacity = ot.capacity;
	this->size = ot.size;
	return *this;
}
```

- destructor

```
DynamicArray::~DynamicArray() {
	delete[] elems;
}
```

Iterator

- utilizați pentru a parcurge un container de obiecte
- oferă un mecanism generic (abstract) pentru accesul la elemente

Iteratorul va contine

- referință spre containerul pe care-l iterează
- referință spre elementul curent din iteratie (cursor).

Avantaj

permite accesul la elemente fără a expune reprezentarea internă

Clase/Metode prietene (friends)

- Clasa B este clasa prieten cu clasa A dacă B are acces la membrii privați din clasa A
- Util dacă avem nevoie într-o clasa la acces la elementele private dintr-o alta clasă
- Similar este posibil sa avem o funcție prieten cu o clasă.
- Funcția prieten are acces la metode/variabile membre private

Clasă prieten

```
class ItLista {
public:
	friend class Lista;
```

Metodă prieten

```
class List {
public:
	friend void friendMethodName(int param);
```

Implementare iterator

```
/**
 * Itarotor over the DynamicArray
 */
class Iterator {
public:
	void urmator() {
		pozCurrent++;
	}
	int valid() {
	  return pozCurrent < l->getSize();
	}
	TElem element() {
		return l->elems[pozCurrent];
	}

	Iterator(DynamicArray* _l) {
		this->l = _l;
		this->pozCurrent = 0;
	}
private:
	DynamicArray* l;
	 //current element in the iteration
	int pozCurrent;
};
void testIterator() {
	DynamicArray ar1;
	ar1.addE(2);
	ar1.addE(3);
	ar1.addE(4);
	Iterator* it = ar1.begin();
	while (it->valid()) {
		cout << it->element() << " ";
		it->urmator();
	}
	delete it;
	cout << "\n";

}
```

```
class DynamicArray {
	friend class Iterator;
...
```

Iterator – suprascriere operatori

```
DynamicArray ar1;
ar1.addE(2);
ar1.addE(3);
ar1.addE(4);
Iterator it = ar1.begin();
while (it != ar1.end()) {
	cout << (*it) << " ";
	it++;
}
cout << "\n";
```

```
//for like
for (Iterator i = ar1.begin(); i != ar1.end();i++)
{
  cout << (*i) << " ";
}

//backward
it = ar1.end();
--it;
--it;
cout << (*it) << " ";
```

Operatori

```
/**
 * Overload dereferencing operator
 */
TElem& operator*() {
	return l->elems[pozCurrent];
}
/**
 * Overload ++ (prefixed version)
 */
Iterator& operator ++() {
	pozCurrent++;
	return *this;
}
/**
 * Overload ++
 * Postfix version use a dummy param
 */
Iterator& operator ++(int dummy) {
	pozCurrent++;
	return *this;
}
```

```
/**
 * Overload !=
 */
bool operator!=(const Iterator& ot) {
	return ot.pozCurrent != pozCurrent;
}
/**
 * Overload -- (prefixed version)
 */
Iterator& operator --() {
	pozCurrent--;
	return *this;
}
```

Listă – implementată înlănțuit

```
class Nod;

typedef Nod *PNod;
/**
 * Reprezinta un nod din inlantuire
 */
class Nod {
public:
	friend class Lista;

	Nod(E e, PNod urm = 0) {
		this->e = e;
		this->urm = urm;
	}

	E element() {
		return e;
	}

	PNod urmator() {
		return urm;
	}

private:
	//elementul curent
	E e;
	//referinta la nodul urmator
	PNod urm;
};
```

```
/**
 * Lista , implementare folosim reprezentare inlantuita
 */
class Lista {
public:
	friend class ItLista;
	Lista() :prim(0) {
	}

	~Lista();
	/**
	 * Adauga la sfarsitul listei
	 */
	void adaugaSfarsit(E e);
	/**
	 * Adauga dupa pozitia data de iteraror
	 */
	void adaugaDupa(ItLista i, E e);
	/**
	 * Iterator
	 */
	ItLista* iterator();

private:
	PNod prim;
};
```

Iterator – Lista implementată înlănțuit

```
/**
 * Iterator pentru lista inlantuita
 */
class ItLista {
public:

	friend class Lista;
	void urmator() {
		curent = curent->urmator();
	}
	int valid() {
		return curent != 0;
	}
	E element() {
		return curent->element();
	}

private:
	ItLista(Lista& _l) :
			l(_l), curent(l.prim) {
	}
	Lista& l;
	PNod curent;
};

void tiparire(Lista& l) {
	ItLista *i = l.iterator();

	while (i->valid()) {
		cout << i->element() << " ";
		i->urmator();
	}
	cout << "\n";
	delete i;
}
```

Lista generica (funcţionează cu orice tip de elemente)

- typedef Telem = <type name> Ex. `typedef int TElem;`
    - nu putem avea liste cu elemente de tipuri diferite în acelaşi program

- Folosinf void* `typedef void* TElem2;`
    - nu putem adauga constante
    - trebuie sa folosim operatorul cast cand luam elementele

Șabloane (Template)

```
int sum(int a, int b) {
     return a + b;
}

sum(2,3);
```

```
double sum(double a, double b) {
     return a + b;
}

sum(2.6,3.121);
```

- creare de funcții / clase care folosesc același cod sursă pentru diferite tipuri de date
- în loc sa rescriem funcția / clasa pentru fiecare tip de dată putem folosi mecanismul de șabloane pentru a folosi același cod pentru tipuri diferite
- o modalitate de a refolosi codul

Funcţii:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

```
template<typename T> T sum(T a, T b) {
     return a + b;
}
int sum = sumTemp<int>(1, 2);
cout << sum;
double sum2 = sumTemp<double>(1.2, 2.2);
cout << sum2;
```

- T este un parametru pentru şablon, cand folosim
- instanţiere de şablon + procesul de generare a funcţiei pentru un tip de date

```
int sum = sumTemp<int>(1, 2);
```

Clase template:

Un macro (şablon, skeleton) care descrie o mulţime de clase similare.
Clasa template indică compilatorului ca definiţia clasei poate acomoda unul sau mai multe tipuri de date care se vor specifica la momentul utilizării (creare de instanţe)
La momentul utilizării compilatorul crează o clasa actuală înlocuind parametrii cu tipul actual furnizat.

```
template<typename Element>
class DynamicArray {
public:
	/**
	 * Add an element to the dynamic array to the end of the array
	 * e - is a generic element
	 */
	void addE(Element r);
	/**
	 * Delete the element from the given position
	 * poz - the position of the elem to be deleted, poz>=0;poz<size
	 * returns the deleted element
	 */
	Element deleteElem(int poz);

	/**
	 * Access the element from a given position
	 * poz - the position (poz>=0;poz<size)
	 */
	Element get(int poz);
	/**
	 * Give the size of the array
	 * return the number of elements in the array
	 */
	int getSize();
	/**
	 * Clear the array
	 * Post: the array will contain 0 elements
	 */
	void clear();
private:
	Element *elems;
	int capacity;
	int size;
};
```

Lista implemenată folosind șabloane (template)

```
template<typename Element>
class DynamicArray3 {
public:
	/**
	 * Add an element to the dynamic array to the end of the array
	 * e - is a generic element*/
	void addE(Element r);
	/**
	 * Delete the element from the given position
	 * poz - the position of the elem to be deleted, poz>=0;poz<size
	 * returns the deleted element*/
	Element deleteElem(int poz);
	/**
	 * Access the element from a given position
	 * poz - the position (poz>=0;poz<size)*/
	Element get(int poz);
....
private:
	Element *elems;
	int capacity;
	int size;
}
/**
 * Add an element to the dynamic array
 * r - is a rational number
 */
template<typename Element>
void DynamicArray3<Element>::addE(Element r) {
	ensureCapacity(size + 1);
	elems[size] = r;
	size++;
}
/**
 * Access the element from a given position
 * poz - the position (poz>=0;poz<size)
 */
template<typename Element>
Element DynamicArray3<Element>::get(int poz) {
	return elems[poz];
}
template<typename Element>
void DynamicArray3<Element>::set(int poz, Element el) {
	elems[poz] = el;
}
```

Instanţiere DynamicArray pentru diferite tipuri de date

```
void testAddDouble() {

	DynamicArray3<double> ar1;
	ar1.addE(1.3);
	double elem = ar1.get(0);
	assert(elem==1.3);
	assert(ar1.getSize()==1);

	ar1.addE(2.5);
	elem = ar1.get(1);
	assert(elem==2.5);
	assert(ar1.getSize()==2);
}


void testAddIntParam2() {

	DynamicArray3<int> ar1;
	assert(ar1.getSize()==0);
	ar1.addE(1);
	int elem = ar1.get(0);
	assert(elem==1);
	assert(ar1.getSize()==1);

	ar1.addE(2);
	elem = ar1.get(1);
	assert(elem==2);
	assert(ar1.getSize()==2);
}
```

```
void testRationalAdd() {
	DynamicArray3<Rational> ar1;
	Rational r1(1, 1);
	ar1.addE(r1);
	Rational elem = ar1.get(0);
	assert(elem.getUp()==1);
	assert(elem.getDown()==1);
	assert(ar1.getSize()==1);

	Rational r2(2, 3);
	ar1.addE(r2);
	elem = ar1.get(1);
	assert(elem.getUp()==2);
	assert(elem.getDown()==3);
	assert(ar1.getSize()==2);
}
```

Elemente statice (metode şi variabile membre)

Atributele declarate **`static`** aparțin clasei nu instanțelor

Ele descriu caracteristici ale clasei nu fac parte din starea obiectelor
Pot fi privite ca și variabile globale definite în interiorul clasei

- Pot fi accesate de toate instanțele
- pot fi accesate folosind operatorul scope `::`

```
/**                                          Rational::nrInstances
 * New data type to store rational
numbers
 * we hide the data representation
 */
class Rational {
public:
       /**
        * Get the nominator
        */
       int getUp();
       /**
        * get the denominator
        */
       int getDown();
private:
       int a;
       int b;
       static int nrInstances = 0;
};
```

Diagrame UML .

- UML (Unified Modeling Language)
- Standard folosit la scară largă pentru a specifica, vizualiza, construi, documenta sisteme software
- Este independent de limbajul de programare
- Permite modelarea sistemelor soft, oferă un limbaj comun

UML Diagrame de clase

descrie clasele din program și relațiile între ele

Conține:

- numele clasei
- variable membre – (nume + tip)
- metode - (nume + parametri + tipul returnat)
- modificatori de acces
  - membrii privați cu “-”,
  - membrii publici cu “+”