

Програмування GUI

На основі мови C++ та фреймворку Qt

Лекція 12

Класи контейнери

Класи контейнери є шаблонными класами та призначені для зберігання набору елементів довільного типу.



Клас QVector



Є структурою даних, в якій елементи містяться в сусідніх ділянках оперативної пам'яті.

Підтримує довільний доступ до елементів, що здійснюється досить ефективно.

Додавання елементів до кінця вектора та видалення елементів з кінця вектора також здійснюється досить ефективно. Вставка елементів на початок або в середину вектора, а також видалення елементів з цих позицій здійснюється значно повільніше.

Тому клас QVector слід використовувати як альтернативу динамічному масиву.

Клас QVector



Конструктори:

1. `QVector ()` – створює порожній контейнер (не містить елементів)



`QVector <int> vect1;` //Порожній контейнер для зберігання елементів типу int

2. `QVector (int size)` – створює контейнер і резервує місце для зберігання size елементів.



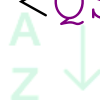
`QVector <double> vect2 (10);` //10 елементів типу double

3. `QVector <T> (int size, T& value)` – створює контейнер із size елементами, значення яких рівні value



`QVector <QString> vect3 (10, "Hello\n");`

4. `QVector (const QVector <T> &other)` – створює контейнер на базі іншого контейнера



`QVector <QString> vect4 (vect3);`

Клас QVector



 Дізнатись розмір вектора можна за допомогою методу

```
int size()
```

```
int n=vect3.size(); // n=10
```



Змінити розмір вектора можна за допомогою методу

```
void resize(int size)
```


```
Vect1.resize(5);
```






Для зберігання в об'єктах класів-контейнерів об'єктів користувальницьких класів, ці класи обов'язково повинні володіти конструктором без аргументів (за замовчуванням), конструктором копіювання та оператором присвоєння.



Клас QVector




```
class MyValue
{ int value;
public:
    MyValue ():value(0){}
    MyValue (int v):value(v){}
    void setValue (int v)
    {value=v;}
    int getValue ()
    {return value;}
};
QVector <MyValue> vv(10, MyValue(5));
```










Клас QVector

Qt



```
class MyValues {
int *values;
int size;
public:
MyValues () {size=1;
values=new int [size];}
MyValues (int s) { size=s; values=new int [size]; }
~MyValues() { delete [] values; }
MyValues(const MyValues &v) {
size=v.size; values=new int [size];
for (int i=0;i<size; i++)
{values[i]=v.values[i]; } }
MyValues &operator=(const MyValues&v)
{ if (this==&v) return *this;
delete []values;
size=v.size;
values=new int [size];
for (int i=0;i<size; i++) {values[i]=v.values[i]; }
return *this; }
void setValues (int i, int v)
{values[i]=v}
int getValues (int i)
{return values[i];}
};
```



Клас QVector



Додати елемент до кінця вектора можна декількома способами:

```
QVector <QString> str;
```

1. `str.append("1"); // str="1"`

2. `str<<"2"<<"3"<<"4"; //str="1" "2" "3" "4"`

3. `str.push_back("5"); //str="1" "2" "3" "4" "5"`



Вставити елемент у початок контейнера можна в такий спосіб:

1. `str.push_front("0");`

2. `str.prepend("0");`



Вставити елемент у довільну позицію в контейнері можна так:

```
str.insert(1, "0.5");
```



Клас QVector



 Перевірити, чи контейнер є порожнім:

```
bool empty=str.isEmpty();
```

 Видалити елементи з контейнера:


1. `str.pop_back();` //видалити з кінця вектора

2. `str.pop_front();` //Видалити з початку

 вектора


3. `str.remove(1);` //Видалити 1-й елемент

4. `str.clear();` //Видалити всі елементи вектора




 5. `str.resize(0);` //Видалити всі елементи з вектора



Клас QListedList <T>



Призначений для створення об'єктів типу "пов'язаний список елементів". У порівнянні з масивом і класом QVector має перевагу - операції вставки та видалення елементів у список виробляються набагато швидше, за однаковий час незалежно від позиції вставки або видалення. Це досягається за рахунок того, що кожен елемент списку зберігається у вузлі, що містить покажчики на наступний та попередній вузли. І при видаленні або вставці елемента необхідно просто змінити значення покажчиків, тоді як в масиві або векторі необхідно виконати переміщення частини елементів в ту чи іншу сторону.

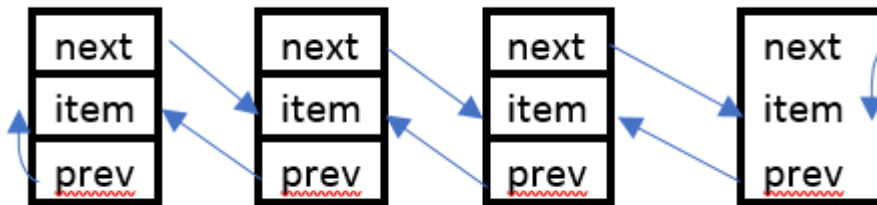


Клас QListedList <T>



Не підтримує оператор довільного доступу [], тому для доступу до довільного елемента списку, видалення або вставки елемента у довільну позицію у списку необхідно виконати прохід по елементах списку з початку або з кінця за допомогою ітераторів.


```
QListedList <double> list; //Порожній список
```



Клас QListedList <T>



Для додавання елементів до початку або до кінця списку, а також для видалення елементів з цих позицій у класі є методи, аналогічні методам класу QVector:




```
list<<0.1<<0.2<<0.3<<0.4<<0.5<<0.6;
```

0.1	0.2	0.3	0.4	0.5	0.6
-----	-----	-----	-----	-----	-----

0.1	0.2	0.3	0.4	0.5
-----	-----	-----	-----	-----


```
list.pop_back();
```




Використання ітераторів




 Класи-контейнери підтримують роботу з ітераторами.

 Ітератор є узагальнення покажчика, що дозволяє йому адресувати різні структури даних.

 Класи-контейнери підтримують ітератори у стилі C++ та у стилі Java.

Ітератори C++ вказують самі елементи списку.

 Ітератори в стилі Java вказують не на самі елементи, а на позиції між елементами.



Ітератори в стилі Java



З класами контейнерами можна використовувати два типи ітераторів у стилі Java:

- ітератор, який використовується тільки для читання (не може змінювати список),
- ітератор для читання та запису (може змінювати список).

Класи ітераторів першого типу:

`QVectorIterator<T>`,
`QLinkedListIterator<T>` і т.д.

Класи ітераторів другого типу у своїй назві мають **Mutable**:

`QMutableLinkedListIterator<T>`,
`QMutableQVectorIterator<T>` і т.д.

Ітератори в стилі Java



Створити ітератор для роботи зі списком:

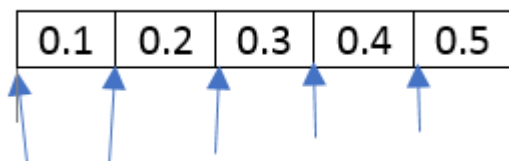
```
QLinkedListIterator<double> i (list);
```



Посилання на список потрібне для ініціалізації ітератора, у цьому випадку ітератор вказуватиме на позицію перед початком (першим елементом) списку.



Ітератори в стилі Java вказують не на елементи, а на позиції між елементами списку.



Ітератори в стилі Java



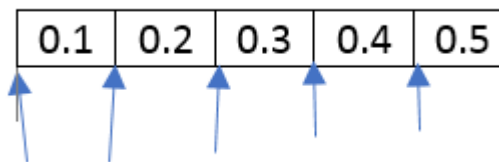
Для проходження списком можна скористатися циклом

`while:`

```
while (i.hasNext())
```

```
cout<<i.next();
```

```
}
```



Метод `bool hasNext()` повертає `true`, якщо є наступний елемент (праворуч від ітератора).

Метод `next()` повертає елемент, розташований праворуч ітератора і переміщає ітератор в наступну позицію.

`i.toBack();` // встановлює ітератор на кінці списку

```
while (i.hasPrevious())
```

```
{cout<<i.previous();}
```


Ітератори в стилі Java



Методи `T& peekNext()` та `T& peekPrevious()` аналогічні методу `next()` і `previous()` – повертають наступний та попередній елементи списку, але не переміщують ітератор на створену позицію. Для пошуку елемента у списку можна використовувати методи `bool findNext (const T&value)`, `bool findPrevious (const T&value)`, які повертають значення `true`, якщо елемент знайдений. Метод `findNext ()` шукає елемент, починаючи з поточної позиції до кінця списку. Якщо елемент знайдено, ітератор вказуватиме на позицію після знайденого елемента, якщо не знайдено – на позицію після останнього елемента списку.

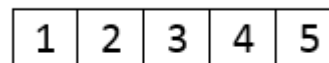
Ітератори в стилі Java



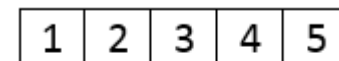
```
LinkedList <int> list;
```

```
list<<1<<2<<3<<4<<5;
```

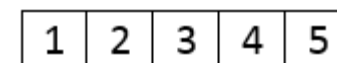
```
LinkedListIterator <int> i(list); // на  
початок списку
```



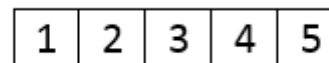
```
bool find=i.findNext(3); //find=true
```



```
bool find=i.findPrevious(1); //find=true
```



```
find=i.findNext(7); //find=false
```



Ітератори в стилі Java



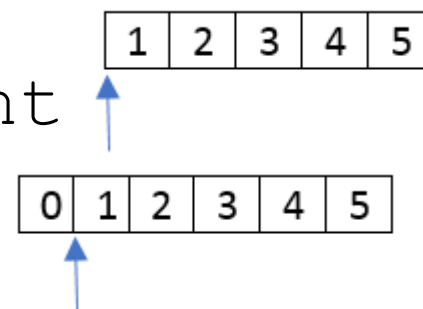
Для вставки елементів, видалення елементів зі списку, а також для зміни значень елементів списку, необхідно використовувати ітератор класу

```
QMutableLinkedListIterator (QLinkedList  
<T> &list).
```

Для переміщення по елементах списку, а також для пошуку елементів у списку даний клас має ті ж методи, що і клас `QLinkedListIterator`.


Метод `void insert(const T& value)` – вставляє елемент після позиції, яку вказує ітератор і переміщає ітератор на позицію після вставленого елемента.

```
QMutableLinkedListIterator <int  
i(list); i.insert (0);
```

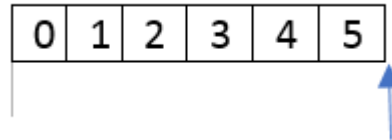


Ітератори в стилі Java

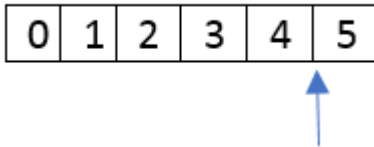


 Метод `void remove()` видаляє елемент, через який щойно пройшов ітератор за допомогою методів `next()`, `previous()`, `findNext()`, `findPrevious()`.

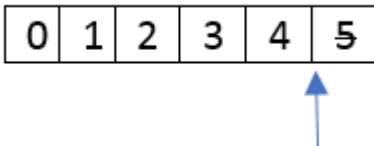
`i.toBack();`



`i.previous();`



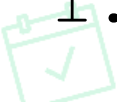
 `i.remove();`



Ітератори в стилі Java



 **Метод** `void setValue (const T&value)` змінить значення елемента, через який щойно пройшов ітератор.

 `i.toFront();`

`i.next();`


0	1	2	3	4
---	---	---	---	---

0	1	2	3	4
---	---	---	---	---

 `i.setValue(5);`

5	1	2	3	4
---	---	---	---	---

Методи `const T&value()` `const` та `T&value()` повертають посилання на елементи, через які щойно пройшов ітератор.

 `i.toFront();`

`i.next();`

`int v=i.value(); //v=5`

5	1	2	3	4
---	---	---	---	---



Використання ітераторів у стилі C++




 Ітератори C++ вказують на самі елементи списку.




Клас QList



 Клас `QList` є контейнером типу масив-список.

 Він підтримує довільний доступ до елементів як клас `QVector`, при цьому вставка та видалення елементів у середині списку здійснюється досить швидко, як у зв'язаному списку (кількість елементів не більше 1000).

 Для видалення елементів, додавання елементів, переміщення елементами контейнер `QList` підтримує всі перелічені методи класу `QVector`.



Клас QList



Конструктори:

1. `QList <T> ();`

2. `QList (const QList <T> &other)`



`QList <int> list; // порожній список`

`list<<1<<2<<3<<4;`



Клас QList



Типи ітераторів:

1. `QList<T>::iterator` – дозволяє модифікувати дані



2. `QList<T>::const_iterator` – не дозволяє модифікувати дані


3. За допомогою методів класу `QList` `iterator` `begin()` та `iterator` `end()` можна отримати ітератори на початок та на кінець списку

```
QList<int>::iterator i=list.begin();  
i=list.end();
```


4. Для переходу між елементами списку можна використовувати оператори `++` та `--`. Для вилучення елемента списку з позиції поточного ітератора необхідно використовувати оператор `*`.




Клас QList





```
i=list.begin();  
while (i!=list.end())  
{cout<<*i<<endl; //cout<<*i++;  
i++;  
}
```




В зворотньому напрямку:





```
i=list.end();  
while (i!=list.begin())  
{cout<<--*i<<endl;  
}  
Qlist <int>::iterator i=list.begin();  
while (i!=list.end()) {  
*i+=10; // збільшуємо значення елемента на 10  
cout<<*i++<<endl;  
}
```



Клас QList



Вставити новий елемент у список можна за допомогою методу `iterator insert (iterator before, const T&value)`, де `before` - ітератор, що вказує на елемент списку, перед яким буде вставлено елемент `value`.




Хоча ітератор `before` передається за значенням, а не за посиланням, після виклику функції його буде змінено, й надалі його використовувати не можна.




Натомість необхідно використовувати ітератор на вставлений елемент, який повертає функцію.




Клас QList



```
QList <double> list;  
list<<0.1<<0.2<<0.3;  
QList <double>::iterator i=list.begin();  
i++;  
i=list.insert (i,0.15); //0.1 0.15 0.2 0.3
```



Видалити елемент зі списку можна за допомогою методу `iterator erase (iterator pos)`, де `pos` - ітератор, що вказує на елемент, що видаляється. Функція повертає ітератор на наступний елемент у списку або позицію після останнього елемента, якщо віддалений елемент єдиний.





```
i=list.erase(i); //0.1 0.2 0.3
```



Клас QQueue



 Клас `QList <T>` є базовим для класу `QQueue <T>`.
Також на його основі створено клас `QStringList`.


 Клас `QQueue` призначений до створення об'єктів, моделюючих роботу черги, тобто структури даних типу FIFO (first-in-first-out). Дані до черги заносяться з кінця, а витягуються з початку черги.

 Методи класу `QQueue`:

1. `T dequeue()` – витягує перший (head) елемент із черги і повертає його.
2. `void enqueue(const T &t)` – додає значення `t` на кінець черги (tail).
3. `T& head()` – повертає посилання на перший елемент черги



Клас QList

A small icon of a network or graph structure.

```
QQueue <int> q;
```

```
q.enqueue(1);
```


```
q.enqueue(2);
```

A small icon of a calendar.

```
q.enqueue(3);
```

```
int i=q.dequeue(); //i=1;
```

```
int head=q.head(); //head=2;
```

A small icon of a computer monitor.

```
QQueue <int> q;
```

```
q.enqueue(1);
```

A small icon of two interlocking gears.

```
q.enqueue(2);
```

```
q.enqueue(3);
```

```
while (!q.isEmpty())
```


A small icon of a code editor window.


```
cout<<q.dequeue();
```

A small icon of a green 'Z' with a downward arrow.A small icon of a cursor pointing at a document.

Клас QStringList



 Призначений для створення об'єктів типу списку, елементом якого є рядки. Успадкований від класу `QList <QString>`.

 `QStringList list;`

`list<<"ABC"<<"abc"<<"432"<<"1ab";`

`list.sort();` // "1ab" "432" "ABC" "abc" сортування елементів списку

 `QComboBox *box=new QComboBox (this);`

`box->addItem(list);` // додавання елементів до комбінованого списку



Клас QStringList



За допомогою методу `QString join (const QString &separator)` можна отримати рядок, що складається з підрядків, кожна з яких є списком елементів, розділених сепаратором, посилання на який є параметром методу.

```
QString str=list.join("_");  
//str=1ab_432_ABC_abc
```

За допомогою оператора `+` можна поєднувати списки.

```
QStringList list1;  
list1<<"1"<<"2"<<"3";  
QStringList list2;  
list2<<"4"<<"5"<<"6";  
list1=list1+list2; //list1="1" "2" "3"  
"4" "5" "6"
```


Клас Qstack<T>



Призначений для створення об'єктів типу стек. Клас `QStack<T>` успадкований від класу `QVector<T>`. Вставка та вилучення елементів зі стека проводиться з верхньої частини стека (top). Стек працює за принципом LIFO (last-in-first-out).

```
QStack <double> s;
```

```
s.push(0.1);
```

```
s.push(0.2);
```

```
s.push(0.3);
```

```
double v=s.pop(); //
```

0.3	←top
0.2	
0.1	

0.2	←top
0.1	

Неявне сумісне використання



Qt у всіх контейнерах, а також для зберігання об'єктів класів типу QString, QImage і т.д. застосовується неявне сумісне використання даних. Це робить дуже ефективним передачу (повернення) об'єктів даних класів в функцію за значенням. Неявне спільне використання даних означає, що для двох і більше ідентичних об'єктів одного класу, в пам'яті зберігатиметься лише один екземпляр даних, і всі об'єкти будуть посилатися на ці дані. Тобто. при копіюванні даних з першого об'єкта на другий, фізичного копіювання даних не буде, другий об'єкт після виконання операції копіювання посилатиметься на дані першого об'єкта.