Лекція 10

Тема лекції: Шаблони контейнерних класів. Методи доступу до об’єктів контейнерних класів

Введення до шаблонів контейнерних класів

У контейнерних класах можна запам'ятовувати об'єкти даних будь-яких видів і вибирати будь-який метод зберігання. Наприклад, можна запам’ятати об'єкти у векторі (в масиві з розташованих підряд значень) або у зв'язному списку. Контейнери приховують даталі своєї внутрішньої реалізації, так що можна поміняти метод зберігання даних без подальшого переписування коду користувача. З допомогою контейнерних класів можна вибрати найкращі методи зберігання для будь-якої програми, які легко змінити згодом.

АТД і ФСД

Контейнерний клас - це абстрактний тип даних (АТД). Він забезпечує абстракцію методів зберігання даних. Наприклад, стек - це АТД з такими функціями як *push()* і *pop()* для запам'ятовування і відновлення даних методом стеку. Внутрішня реалізація цих функцій не залежить від концепції стеку. Можна створити стек за допомогою вектора або використовувати список. Стек - це просто метод програмування, абстракція, яка не має нічого спільного з конкретними деталями реалізації внутрішнього зберігання даних в ньому. Однак вектори і списки не можна відокремити від їх внутрішньої структури. Тому вони називаються фундаментальними структурами даних (ФСД). ФСД - це будь-яка ситуація, яка може використовуватися для реалізації АТД. Наприклад, можна реалізувати клас стеку (АТД) з допомогою зв'язного списку (ФСД). Або ж реалізувати стек можна з допомогою іншого фундаментального типу. Часто масиви називають векторами. Але це не одне і те ж саме. Вектор - це фундаментальна структура даних, у якій об'єкти запам'ятовуються один за одним. Масив - це абстракція, яка забезпечує випадковий

доступ до об'єктів шляхом операції індексування. Як контейнер масив можна реалізувати з допомогою вектора, списку або іншого фундаментального механізму запам'ятовування даних. У кожному контейнерному класі комбінується один АТД і одна ФСД. Ім'я результуючого класу може звучати як *TArrayAsVector*. За таким ім’ям можна визначити, що контейнер - це масив (АТД), реалізований вектором (ФСД). Початкова літера Т означає "тип". Для того, щоб скористатися контейнерним класом, слід задати тип даних, які будуть запам'ятовуватися в такому вигляді: *TClass <T>*, де Т - тип даних користувача. Наприклад, *TStackAsList <long>* здатен запам'ятовувати довгі цілі об'єкти у зв'язному списку. *TQuestionAsVector <MyDataType>* створює контейнерний клас черги, в якому можна запам'ятовувати об'єкти типу *MyDataType* з використанням вектора в якості внутрішнього механізму запам'ятовування даних. Контейнери можуть містити прості об'єкти (цілі, дійсні і т.і.), структурні дані (масиви, рядки і структури) та об'єкти класів. Можна зберігати в контейнері самі об'єкти або покажчики на них. Наприклад, щоб скористатися стековим контейнером, слід включити заголовочний файл STACKS.H:

#include <classlib\stacks.h>

Потім потрібно створити контейнерний об'єкт одного з класів, що містяться в заголовочному файлі, наприклад, TStackAsVector. В довіднику по бібліотекам потрібно знайти конструктор:

TStackAsVector(unsigned max = DEFAULT_STACK_SIZE);

Прототип цього конструктора класу свідчить про те, що можна задати необов'язковий параметр, що визначає максимальний розмір стеку, якщо ж скористатися значенням за замовчуванням, визначеному в заголовочному файлі RESOURCE.H. Необхідно також забезпечити шаблон типом даних, які будуть зберігатися там. Наприклад, створити текстовий контейнер для дійсних об'єктів подвійної точності можна так:

TStackAsVector<double> myStack(100);

Це оголошення створить контейнер з ім'ям myStack з шаблону класу TStackAsVector, здатний зберігати до ста дійсних значень подвійної точності. Більшість контейнерів автоматично збільшує свій розмір для зберігання більшої кількості даних, тому значення просто визначають початковий розмір контейнера.

Вони не обов'язково обмежують контейнер фіксованим числом об'єктів. Після створення контейнера можна запам’ятовувати і читати з нього дані за допомогою викликів однієї або кількох функцій-членів, які можуть успадковуватися з класів-предків.

Угоди іменування шаблонів

Імена контейнерних класів характеризують їх можливості і взаємозв'язку. Вони мають вигляд:

Т[префікс][АТД][As][ФСД][суфікс]

Всі імена класів починаються з літери Т. Необов'язковий суфікс вказує на спосіб запам'ятовування даних. Наприклад, сортуються об'єкти (S) або запам'ятовуються побічно у вигляді покажчиків (I). Потім йде такий АТД, як масив (Array) чи стек (Stack). Якщо за ним слідує слово As значить, контейнер реалізований на основі заданої ФСД, яка може бути, наприклад, списком (List), двозв’язним списком (DoubleList) або іншим фундаментальним типом даних. Необов'язковий суфікс вказує на тип супутнього класу. Наприклад, деякі класи закінчуються словом Iterator, що означає, що цей клас виконує ітераційні дії над даними в контейнері.

Таблиця бібліотечних абстрактних типів даних і відповідних їм заголовочних файлів, а також фундаментальних структур даних, використовуються для реалізації кожного контейнера

АТД	Заголовочний файл	Двозв’язний список	Хеш-табл.	Список	Вектор
Array	ARRAY.h				X
Bag (мультимножина)	BAGS.H				X
Deque (Дек-черга з двостороннім доступом	DEQUES.H	X			X
Dictionary (словник)	DICT.H		X		
Queue (черга)	QUEUES.H	X			X
Set (множина)	SETS.H				X

Stack	STACKS.H			X	X

Таблиця фундаментальних структур даних

ФСД	Заголовочний файл	АТД
Двозв’язний список	DLISTIMP.H	дек, черга
Хеш-таблиця	HASHIMP.H	словник
Список	LISTIMP.H	стек
Вектор	VECTIMP.H	масив, мультимножина, дек, черга, множина, стек

Виходячи з таблиць, можна припустити, що існують контейнерні класи з іменами T*DequeAsDoubleList* і *TDequeAsVector*.

Модифікатори способів зберігання

Вони задаються відразу після літери "T" в імені класу. Наприклад, клас *TIStackVector* означає непрямий контейнерний клас стеку, реалізований за допомогою вектора. Непрямі контейнери зберігають покажчики на об'єкти.

C - с підрахунком числа об'єктів.

I - з запам'ятовуванням покажчиків.

IS - з сортуванням і запам'ятовуванням покажчиків.

M - з управлінням пам'яттю.

MC - з управлінням пам'яттю і підрахунком числа об'єктів.

MI - з управлінням пам'яттю і запам'ятовуванням покажчиків.

MIC - з управлінням пам'яттю, запам'ятовуванням покажчиків, підрахунком числа об'єктів.

MIS - з управлінням пам'яттю, запам'ятовуванням покажчиків, сортуванням.

MS - з управлінням пам'яттю, сортуванням.

S - з сортуванням.

Контейнери класу M використовуються для заміни стандартного управління пам'яттю. Контейнери з сортуванням повинні мати можливість сортувати свої об'єкти. Контейнери зі зберіганням об'єктів повинні бути здатні їх копіювати. Тобто,

потрібно забезпечити клас конструктором копії і перевантаженими операторами == і <<.

Розглянемо суфікси в іменах контейнерів. Класи, імена яких закінчуються словом *Element*, забезпечують "спискові оболонки" об'єктів. Існує лише два таких класи: *TMDoubleListElement* і *TMListElement.* Вони дають можливість створювати списки об'єктів, які не мають полів - покажчиків. Ці класи використовуються тільки фундаментальними структурами даних. Інший модифікатор типу класу - *Imp* означає реалізацію. Класи, імена яких закінчуються на *Imp* - це фундаментальні структури даних, що використовуються для реалізації абстрактних контейнерів. Наприклад, клас *TDoubleListImp* реалізує двозв’язний список. Третій тип модифікатора, *Iterator* означає, що клас постачає засобами ітерації фундаментальну структуру або контейнер зі схожим ім'ям. Ітератор забезпечує доступ до всіх об'єктів, що містяться в контейнері. Класи ітераторів, що закінчуються на *Imp*, забезпечують засобами ітерації фундаментальні структури. Наприклад, клас *TDoubleListIteratorImp* - реалізація ітератора для об'єктів, що зберігаються в структурі *TDoubleListImp.*

Клас *TSouldDelete* повідомляє, повинен контейнер видаляти об'єкти, які він зберігає, чи слід перекласти цей обов'язок на користувача.

Опосередковане і безпосереднє запам'ятовування об'єктів в контейнерах

```
#include<iostream.h>
#include<cstring.h>
#include<classlib\arrays.h>
#define TRUE 1
#define FALSE 0
TArrayAsVector <string> dStrings(10);
TIArrayAsVector <string> iStrings(10);
int main()
{
int done = FALSE;
int i;
string s;
char buf[81];
while(!done){
cout << "введіть рядок";
cin.getline(buf,sizeof(buf));
```

```
s = buf;
if(s.length() == 0) dobe = TRUE;
else{
dStrings.Add(s);
iStrings.Add(new string(s));
}
}
cout << "Безпосередній об'єкт :" << endl;
for(i=0;i<dStrings.GetItemsInCounter();i++)
cout << dStrings[i] << endl;
cout << "Об'єкт з непрямим запам'ятовуванням:";
for(i=0;i<iStrings.GetItemsInCounter();i++)
cout << *iStrings[i] << endl;
return 0;
}
```

У програмі оголошуються 2 контейнера, здатні запам'ятовувати об'єкти класу string, оголошення якого міститься в заголовочному файлі cstring.h. Перше оголошення створює контейнер, запам'ятовує безпосередньо самі об'єкти. Друге оголошення створює контейнер, який запам'ятовує покажчики на динамічні рядки, створені за допомогою оператора new. Обидва контейнери *dStrings і iStrings,* володіють однаковими наборами функцій і можливостями. Вони розрізняються лише тим, що в *dStrings* створюється копія рядка, а в *iStrings* запам'ятовується покажчик на рядок. Далі програма запитує рядок, який поміщає до символьного буфера. Після використання буфера для створення рядкового об'єкта S, програма вставляє його в контейнер *dStrings* за допомогою звернення до функції *Add()* контейнера класу. Вона запам'ятовує новий об’єкт в масиві контейнера. Не всі контейнерні класи мають функцію *Add*(), але всі вони мають хоча б один спосіб запам'ятовування об'єктів. Для вставки рядка в контейнер *iStrings* в програмі використовується та ж сама функція *Add().* Було б некоректно передавати адресу об'єкта S функції непрямого контейнера *Add().* Символьна змінна S - лише тимчасовий об'єкт, який розміщується в стеку програми. Передача адреси цього об'єкта непрямого контейнеру буде причиною виникнення проблем при знищенні контейнера. При використанні непрямих контейнерів завжди слід знати, де розміщуються об'єкти. Можна, наприклад, скористатися оператором створення

динамічних рядкових об'єктів new для запам'ятовування їх у контейнері. Оскільки в цьому прикладі контейнерами служать масиви, вони забезпечуються функцією *GetItemInContainer()*, яка повідомляє кількість збережених рядків. Ця функція використовується для відображення об'єктів у контейнері з запам'ятовуванням безпосередньо об'єктів і для відображення розіменованих елементів масиву у непрямому контейнері.

Ітератори

Ітератори - це альтернативний метод доступу до об'єктів контейнера. Ітераторами забезпечуються абстрактні контейнери масивів і черг, а також фундаментальні структури, списки, вектори. За допомогою ітератора можна пересуватися по об'єктах в структурі без їх видалення. Наприклад, зазвичай тільки верхній об'єкт стеку доступний для перегляду. Для отримання інших об'єктів необхідно видаляти їх, викликаючи функцію *Pop()*. За допомогою ітератора можна пересуватися по стеку для виконання операцій над його вмістом.

Приклад. Ітератор використовується для перегляду об'єктів у черзі. Набір рядків запам'ятовується непрямим чином, у контейнері класу *TIQueueAsDoubleList*.

```
TIQueueAsDoubleList <string> iQueue;
int main()
{
iQueue.Put(new string("Line up"));
iQueue.Put(new strinf("for linr"));
iQueue.Put(new string("Logical"));
TIQueueAsDoubleListIterator <string> iterator <iQueue>;
//Застосування ітератора
cout << "Використання ітератора:" << endl;
while(iterator!=0){
cout << *iterator.Current() << endl;
iterator++;
}
//Застосування стандартного методу вилучення з черги
while(!iQueue.IsEmpty()){
string *P = iQueue.Get();
```

```
cout << *P << endl;
delete p;
}
return 0;
}
```

В якості контейнера в програмі оголошується глобальний об'єкт з ім'ям *iQueue*. Так як контейнер - непрямий, в ньому запам'ятовуються покажчики на рядкові об’єкти. Для додавання рядків в чергу програма передає покажчики, що повертаються оператором new, функції-члену *Put()* всіх контейнерів-черг. Функція *Put()* для черги має те ж саме значення, що і *Add()* для масивів. Потім у програмі створюється ітератор класу, розроблений для спільного використання з класом *TIQueueAsDoubleList*. Для формування імені класу в кінці додається слово *Iterator*. Виходить занадто довге ім'я. Тому корисно створити більш короткий аліас:

typedef TIQueueAsDoubleListIterator <string> TIterator;

Оголошення контейнера тепер буде виглядати так:

TIterator iterator (iQueue);

Передача об'єкта *iQueue* конструктору ітератора пов'язує ітератор з цим контейнером. Операції з використанням ітератора будуть виконуватися над даними контейнера *iQueue*. Вираз *iterator.Current()* повертає поточний елемент контейнера *Queue* першого об'єкта при першому зверненні. Якщо контейнер порожній, ітератор повертає ціле значення. Це можливо завдяки тому, що в класах-ітераторах перевантажується оператор перетворення до типу *int()*. У всіх ітераторах реалізований оператор інкременту (постфіксний і префіксний). Оператор ++ повертає об'єкт того ж типу, що і контейнер. Всі ітератори мають принаймі один конструктор, функцію *Restart()* (для встановлення ітератора в початкове положення), перевантажені оператори *int()* та ++. Деякі ітератори забезпечуються також додатковими спеціальними функціями. Наприклад, в ітераторах масивів перевантажується функція *restart()* з двома параметрами:

void Restart(unsigned start, unsigned stop);.

Можна передати функції *Restart()* значення ітераторів для того, щоб почати ітераційний процес в обмеженому діапазоні об'єктів, що зберігаються в контейнері-масиві.

Приналежність об'єктів

Можна задати приналежність об'єктів, створивши непрямий контейнер, подібний *TIArrayAsVector*, і викликавши функцію, успадковану з класу *TShouldDelete.*

Наприклад, розглянемо непрямий контейнер, що запам'ятовує покажчики на рядкові об'єкти:

TIArrayAsVector <string> stuff(100);

Для того, щоб визначити, чи володіє *stuff* своїми об'єктами, слід викликати функцію *OwnsElements()*, успадковану з класу *TShouldDelete:*

cout << stuff.OwnsElements() << endl;

Функція *OwnsElements()* повертає *true*, якщо контейнер володіє своїми об'єктами або *false,* якщо ні. Для зміни статусу власності в контейнері слід викликати перевантажену функцію *OwnsElements(),* яка нічого не повертає і передати їй *true* або *false*. Наприклад:

stuff.OwnsElements(FALSE);

Скасування прав власності контейнера означає, що об'єкти не будуть видалятися автоматично. Статус володіння можна задати тільки для непрямих контейнерів, які зберігають покажчики на об'єкти.

Приклад. Використання приналежності об'єктів:

```
#include <classlib\arrays.h>
//визначити аліаси імен типів
typedef TIArrayAsVector <string> TContainer;
typedef TIArrayAsVectorIterator <string> TIterator;
//Прототипи функції відображення вмісту контейнера
void ShowMe(const char *msg, TContainer &cr);
int main()
{
TContainer *cp; //покажчик на контейнер
string S(Об'єкт 1"); //рядковий об'єкт
//створити контейнер і відобразити його права володіння
cp = new TContainer(10,0,10);
```

```
if(cp->OwnsElements()) cout << "TRUE";
else cout << "FALSE";
//Запам'ятати кілька рядкових об'єктів у контейнері:
cp -> Add(new string("Об'єкт 2"));
cp -> Add(new string("Об'єкт 3"));
cp -> Add(&S);
ShowMe ("Вміст контейнера:",*cp);
//Видалити об'єкт з контейнера і знищити його за індексом
cp -> Detach(2,TSouldDelete::Delete);
ShowMe("Знищити за індексом ",*cp);
//Видалити об'єкт з контейнера не видаляючи його при цьому
cp -> Detach(&S,TSouldDelete::NoDelete);
ShowMe("Не видаляючи об'єкт",*cp);
//Видалити останній об'єкт і, можливо, знищити його
int n = cp -> GetItemsInContainer();
cp -> Detach(n-1,TSouldDelete::DefDelete);
ShowMe("Останній об'єкт",*cp);
//Видалити всі об'єкти з контейнера і, можливо, знищити
cp -> Flush();
ShowMe():
//Видалити контейнер, не видаляючи об'єкти
delete cp;
return 0;
}
```

У програмі тип *TContainer* визначається як непрямий масив рядків, реалізований за допомогою вектора. Тип *TIterator* - ітератор для цього контейнера. Далі оголошується покажчик на тип *TContainer* і за допомогою оператора new створюється об'єкт контейнера. Після цього у контейнері запам'ятовуються кілька рядкових об'єктів, в т.ч. - тимчасовий рядковий об'єкт S, який розміщується в системному стеку:

cp >Add(&S);

Цей прийом є небезпечним. Потрібно не дозволяти контейнеру знищити цей об'єкт. Така операція може зв'язати частину стекового простору з резервом вільної пам'яті купи. Це призведе до псування стеку і купи при виділенні пам'яті в області верхівки стеку, де зберігаються адреси повернення функцій. Для видалення об'єкта з

контейнера застосовується оператор:

cp->Detach(2,TSouldDelete::Delete);

Аргумент 2 означає, що третій об'єкт має бути видалений з контейнера. Інші об'єкти зсуваються вгору для ущільнення звільненого об'єктом місця. Другий аргумент задає, що цей об'єкт буде знищений, незалежно від статусу володіння.

Можна не знищувати об'єкт незалежно від статусу володіння при видаленні його з контейнера. У програмі локальний рядковий об'єкт видаляється з контейнера таким чином:

cp -> (&S,TShouldDelete::NoDelete);

Аналогічні дії необхідно виконати для кожного локального об'єкта в непрямому контейнері до виклику будь-якої іншої функції, яка може призвести до знищення такого об'єкта. Наприклад, якщо в програмі спочатку не виконується видалення з контейнера локального об'єкта, то оператор

cp ->Flush();

зіпсує купу. При виклику функції *Flush()* видаляються всі об'єкти, якими володіє контейнер. У разі, якщо контейнер адресує локальні об'єкти, то перед викликом функції *Flush()* потрібно або скасувати володіння об'єктами, або вилучити такі об'єкти з контейнера.

Можна зробити і так, щоб об'єкти вилучалися за умови, що контейнер володіє ними. Наприклад, у програмі видаляється останній об'єкт, індексований цілим n. Передача функції *Detach()* параметра *DefDelete* вказує функції, що необхідно знищити його при видаленні тільки тоді, коли контейнер володіє ним. Якщо контейнер не володіє своїми об'єктами, оператор видаляє з нього об'єкт, але не звільнить займаєму цим об'єктом пам'ять. В кінці програми знищується контейнер, для якого раніше виділялася пам'ять. Але цей оператор не знищує всі об'єкти, що залишилися в контейнері.

Для того, щоб бути впевненим в тому, що кожен об'єкт видалений, потрібно викликати функцію *Flush()* перед видаленням контейнера або вивести новий контейнерний клас і написати до нього деструктор, який викликає функцію *Flush().*