|  |
| --- |
| **Page 1** |

**лекция 10**

**Тема лекции:** Шаблоны контейнерных классов. Методы доступа к объектам

контейнерных классов

***Введение к шаблонам контейнерных классов***

В контейнерных классах можно запоминать объекты данных любых видов и

выбирать любой метод хранения. Например, можно запомнить объекты в

векторе (в массиве из расположенных подряд значений) или в связном списке.

Контейнеры скрывают датали своей внутренней реализации, так что можно поменять

метод хранения данных без дальнейшего переписывания кода пользователя. С

помощью контейнерных классов можно выбрать лучшие методы хранения для

любой программы, которые легко изменить позже.

***АТД и ФСД***

Контейнерный класс - это абстрактный тип данных (АТД). он обеспечивает

абстракцию методов хранения данных. Например, стек - это АТД с такими функциями

как *push ()* и *pop ()* для запоминания и восстановления данных методом стеке.

Внутренняя реализация этих функций не зависит от концепции стека. можно создать

стек с помощью вектора или использовать список. Стек - это просто метод

программирования, абстракция, не имеющая ничего общего с конкретными деталями

реализации внутреннего хранения данных в нем. Однако векторы и списки нельзя

отделить от их внутренней структуры. Поэтому они называются

фундаментальными структурами данных (ФСД). ФСД - это любая ситуация, которая может

использоваться для реализации АТД. Например, можно реализовать класс стека

(АТД) с помощью связного списка (ФСД). Или реализовать стек можно с

помощью другого фундаментального типа. Часто массивы называют векторами. но

это не одно и то же. Вектор - это фундаментальная структура данных, в которой объекты

запоминаются один за другим. Массив - это абстракция, которая обеспечивает случайный

|  |
| --- |
| **Page 2** |

доступ к объектам путем операции индексирования. Как контейнер массив можно

реализовать с помощью вектора, списка или иного фундаментального механизма

запоминания данных. В каждом контейнерном классе комбинируется один АТД и

одна ФСД. Имя результирующего класса может звучать как *TArrayAsVector.* по такому

именем можно определить, что контейнер - это массив (АТД), реализован вектором

(ФСД). Начальная буква T означает "тип". Для того, чтобы воспользоваться контейнерным

классом, следует задать тип данных, которые будут запоминаться в таком виде: *TClass*

*<T>,* где T - тип данных. Например, *TStackAsList <long>* способен

запоминать длинные целые объекты в связном списке. *TQuestionAsVector*

*<MyDataType>* создает контейнерный класс очереди, в котором можно запоминать

объекты типа *MyDataType* с использованием вектора в качестве внутреннего механизма

запоминания данных. Контейнеры могут содержать простые объекты (целые, действительные и

т.д.), структурные данные (массивы, строки и структуры) и объекты классов. можно хранить

в контейнере сами объекты или указатели на них. Например, чтобы воспользоваться

стековым контейнером, следует включить заголовочный файл STACKS.H:

*#include <classlib \ stacks.h>*

Затем нужно создать контейнерный объект одного из классов, содержащиеся в

заголовочный файл, например, TStackAsVector. В справочнике по библиотекам

нужно найти конструктор:

*TStackAsVector (unsigned max = DEFAULT\_STACK\_SIZE)*

Прототип этого конструктора класса свидетельствует о том, что можно задать

необязательный параметр, определяющий максимальный размер стека, если же

воспользоваться значением по умолчанию, определенном в заголовочный файл

RESOURCE.H. Необходимо также обеспечить шаблон типом данных, которые будут

храниться там. Например, создать текстовый контейнер для настоящих объектов

двойной точности можно так:

*TStackAsVector <double> myStack (100)*

Это объявление создаст контейнер с именем myStack из шаблона класса

TStackAsVector, способный хранить до ста действительных значений двойной точности.

Большинство контейнеров автоматически увеличивает свой размер для хранения большей

количества данных, поэтому значение просто определяют начальный размер контейнера.

|  |
| --- |
| **Page 3** |

Они не обязательно ограничивают контейнер фиксированным числом объектов. после

создание контейнера можно запоминать и читать с него данные с помощью

вызовов одной или нескольких функций-членов, которые могут наследоваться с классов-

предков.

***Соглашения именования шаблонов***

Имена контейнерных классов характеризуют их возможности и взаимосвязи.

Они имеют вид:

T [префикс] [АТД] [As] [ФСД] [суффикс]

Все имена классов начинаются с буквы T. Дополнительный суффикс указывает на

способ запоминания данных. Например, сортируются объекты (S) или

запоминаются косвенно в виде указателей (I). Затем идет такой АТД, как массив

(Array) или стек (Stack). Если за ним следует слово As значит, контейнер

реализован на основе заданной ФСД, которая может быть, например, списком (List),

двозвьязним списком (DoubleList) или другим фундаментальным типом данных.

Необязательный суффикс указывает на тип сопутствующего класса. Например, некоторые классы

заканчиваются словом Iterator, что означает, что этот класс выполняет итерационные действия над

данным в контейнере.

Таблица библиотечных абстрактных типов данных и соответствующих им заголовочный

файлов, а также фундаментальных структур данных, используемых для реализации

каждого контейнера

АТД

заголовочный

файл

Двозвьязний

список

хеш

табл.

список Вектор

Array

ARRAY.h

X

Bag (мультимножество)

BAGS.H

X

Deque (Дек-очередь с

двусторонним доступом

DEQUES.H

X

X

Dictionary (словарь)

DICT.H

X

Queue (очередь)

QUEUES.H

X

X

Set (множество)

SETS.H

X

|  |
| --- |
| **Page 4** |

Stack

STACKS.H

X

X

Таблица фундаментальных структур данных

ФСД

Заголовочный файл АТД

Двозвьязний список

DLISTIMP.H

дек, очередь

Хеш-таблица

HASHIMP.H

словарь

список

LISTIMP.H

стек

вектор

VECTIMP.H

массив, мультимножество, дек, очередь, множество, стек

Исходя из таблиц, можно предположить, что существуют контейнерные классы

именами *T* DequeAsDoubleList и *TDequeAsVector.*

***Модификаторы способов хранения***

Они задаются сразу после буквы "T" в имени класса. Например, класс

*TIStackVector* означает косвенное контейнерный класс стека, реализован

помощью вектора. Косвенные контейнеры хранят указатели на объекты.

С - c подсчетом числа объектов.

I - с запоминанием указателей.

IS - с сортировкой и запоминанием указателей.

M - с управлением памятью.

MC - с управлением памятью и подсчетом числа объектов.

MI - с управлением памятью и запоминанием указателей.

MIC - с управлением памятью, запоминанием указателей, подсчетом

числа объектов.

MIS - с управлением памятью, запоминанием указателей, сортировкой.

MS - с управлением памятью, сортировкой.

S - с сортировкой.

Контейнеры класса M используются для замены стандартного управления

памятью. Контейнеры с сортировкой должны иметь возможность сортировать свои

объекты. Контейнеры с хранением объектов должны быть способны их копировать. Есть,

|  |
| --- |
| **Page 5** |

нужно обеспечить класс конструктором копии и перегруженными операторами == и

<<.

Рассмотрим суффиксы в именах контейнеров. Классы, имена которых заканчиваются

словом *Element,* обеспечивают "списке оболочки" объектов. Существует только два таких

класса: *TMDoubleListElement* и *TMListElement.* Они дают возможность создавать

списки объектов, которые не имеют полей - указателей. Эти классы используются только

фундаментальными структурами данных. Другой модификатор типа класса - *Imp* означает

реализацию. Классы, имена которых заканчиваются на *Imp* - это фундаментальные структуры

данных, используемых для реализации абстрактных контейнеров. К примеру,

класс *TDoubleListImp* реализует двозвьязний список. Третий тип модификатора, *Iterator*

означает, что класс поставляет средствами итерации фундаментальную структуру или

контейнер с похожим именем. Итератор обеспечивает доступ ко всем объектам, содержащиеся

в контейнере. Классы итераторов, оканчивающиеся на *Imp,* обеспечивают средствами

итерации фундаментальные структуры. Например, класс *TDoubleListIteratorImp* -

реализация итератора для объектов, хранящихся в структуре *TDoubleListImp.*

Класс *TSouldDelete* сообщает, должен контейнер удалять объекты, которые он

сохраняет, следует перевести эту обязанность на пользователя.

***Опосредованное и непосредственное запоминание объектов 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 << "введите строку";

cin.getline (buf, sizeof (buf))

|  |
| --- |
| **Page 6** |

s = buf;

if (s.length () == 0) dobe = TRUE;

else {

dStrings.Add (s)

iStrings.Add (new string (s));

}

}

cout << "Непосредственный объект:" << endl;

for (i = 0; i <dStrings.GetItemsInCounter (); i ++)

cout << dStrings [i] << endl;

cout << "Объект с косвенным запоминанием:";

for (i = 0; i <iStrings.GetItemsInCounter (); i ++)

cout << \* iStrings [i] << endl;

return 0;

}

В программе объявляются 2 контейнера, способны запоминать объекты класса

string, объявление которого содержится в заголовочный файл cstring.h. первое

объявления создает контейнер, запоминает непосредственно сами объекты. Второе

объявления создает контейнер, который запоминает указатели на динамические строки,

созданные с помощью оператора new. Оба контейнеры *dStrings и iStrings,*

обладают одинаковыми наборами функций и возможностями. они различаются

тем, что в *dStrings* создается копия строки, а в *iStrings* запоминается

указатель на строку. Далее программа запрашивает строку, помещает в символьного

буфера. После использования буфера для создания строчной объекта S, программа

вставляет его в контейнер *dStrings* посредством обращения к функции *Add ()*

контейнера класса. Она запоминает новый объект в массиве контейнера. Не все

контейнерные классы имеют функцию *Add* (), но все они имеют хотя бы один способ

запоминания объектов. Для вставки строки в контейнер *iStrings* в программе

используется та же функция *Add ().* Было бы некорректно передавать адрес

объекта S функции косвенного контейнера *Add ().* Символьная переменная S - только

временный объект, который находится в стеке программы. Передача адреса этого

объекта косвенного контейнера будет причиной возникновения проблем при уничтожении

контейнера. При использовании косвенных контейнеров всегда следует знать, где

размещаются объекты. Можно, например, воспользоваться оператором создания

|  |
| --- |
| **Page 7** |

динамических строчных объектов new для запоминания их в контейнере. поскольку в

этом примере контейнерами служат массивы, они обеспечиваются функцией

*GetItemInContainer (),* которая сообщает количество сохранившихся строк. эта функция

используется для отображения объектов в контейнере с запоминанием

непосредственно объектов и для отображения розименованих элементов массива в

косвенном контейнере.

***итераторы***

Итераторы - это альтернативный метод доступа к объектам контейнера.

Итераторами обеспечиваются абстрактные контейнеры массивов и очередей, а также

фундаментальные структуры, списки, векторы. С помощью итератора можно

передвигаться по объектам в структуре без их удаления. Например, обычно только

верхний объект стеке доступен для просмотра. Для получения других объектов

необходимо удалять их, вызывая функцию *Pop ().* С помощью итератора можно

передвигаться по стеку для выполнения операций над его содержанием.

Пример. Итератор используется для просмотра объектов в очереди. набор

строк запоминается косвенным образом, в контейнере класса *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>;

// Применение итератора

cout << "Использование итератора:" << endl;

while (iterator! = 0) {

cout << \* iterator.Current () << endl;

iterator ++;

}

// Применение стандартного метода извлечения из очереди

while (! iQueue.IsEmpty ()) {

string \* P = iQueue.Get ();

|  |
| --- |
| **Page 8** |

cout << \* P << endl;

delete p;

}

return 0;

}

В качестве контейнера в программе объявляется глобальный объект с именем

*iQueue.* Так как контейнер - косвенный, в нем запоминаются указатели на

строчные объекты. Для добавления строк в очередь программа передает указатели,

возвращаются оператором new, функции-члену *Put ()* всех контейнеров-очередей. функция

*Put ()* для очереди имеет то же значение, что и *Add ()* для массивов. Затем в программе

создается итератор класса, разработанный для совместного использования с классом

*TIQueueAsDoubleList.* Для формирования имени класса в конце добавляется слово *Iterator.*

Получается слишком длинное имя. Поэтому полезно создать более короткий алиас:

*typedef TIQueueAsDoubleListIterator <string> TIterator;*

Объявления контейнера теперь будет выглядеть так:

*TIterator iterator (iQueue)*

Передача объекта *iQueue* конструктору итератора связывает итератор с этим

контейнером. Операции с использованием итератора будут выполняться над данными

контейнера *iQueue.* Выражение *iterator.Current ()* возвращает текущий элемент контейнера

*Queue* первого объекта при первом обращении. Если контейнер пустой, итератор

возвращает целое значение. Это возможно благодаря тому, что в классах-итераторы

перегружается оператор преобразования к типу *int ().* Во всех итераторов

реализован оператор инкремента (постфиксный и префиксный). оператор ++

возвращает объект того же типа, что и контейнер. Все итераторы имеют по крайней мере один

конструктор, функцию *Restart ()* (для установки итератора в исходное положение),

перегружены операторы *int ()* и ++. Некоторые итераторы обеспечиваются также

дополнительными специальными функциями. Например, в итераторы массивов

перегружается функция *restart ()* с двумя параметрами:

*void Restart (unsigned start, unsigned stop) ;.*

Можно передать функции *Restart ()* значения итераторов для того, чтобы начать

итерационный процесс в ограниченном диапазоне объектов, хранящихся в контейнери-

массиве.

|  |
| --- |
| **Page 9** |

***принадлежность объектов***

Можно задать принадлежность объектов, создав косвенный контейнер,

подобный *TIArrayAsVector,* и вызвав функцию, унаследованную из класса

*TShouldDelete.*

Например, рассмотрим косвенный контейнер, запоминает указатели на

строчные объекты:

*TIArrayAsVector <string> stuff (100)*

Для того, чтобы определить, обладает *stuff* своими объектами, следует вызвать

функцию *OwnsElements ()* унаследованную из класса *TShouldDelete:*

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

Функция *OwnsElements ()* возвращает *true,* если контейнер обладает своими

объектами или *false,* если нет. Для изменения статуса собственности в контейнере следует

вызвать перегруженную функцию *OwnsElements (),* которая ничего не возвращает и

передать ей *true* или *false.* К примеру:

*stuff.OwnsElements (FALSE)*

Отмена прав собственности контейнера означает, что объекты не будут

удаляться автоматически. Статус владения можно задать только для косвенных

контейнеров, сохраняют указатели на объекты.

Пример. Использование принадлежности объектов:

#include <classlib \ arrays.h>

// определить алиасы имен типов

typedef TIArrayAsVector <string> TContainer;

typedef TIArrayAsVectorIterator <string> TIterator;

// Прототипы функции отображения содержимого контейнера

void ShowMe (const char \* msg, TContainer & cr)

int main ()

{

TContainer \* cp; // указатель на контейнер

string S (Объект 1 "); // строчный объект

// создать контейнер и отобразить его права владения

cp = new TContainer (10,0,10)

|  |
| --- |
| **Page 10** |

if (cp-> OwnsElements ()) cout << "TRUE";

else cout << "FALSE";

// Запомнить несколько строчных объектов в контейнере:

cp -> Add (new string ( "Объект 2"))

cp -> Add (new string ( "Объект 3"))

cp -> Add (& S);

ShowMe ( "Содержание контейнера:" \* cp)

// Удалить объект из контейнера и уничтожить его по индексу

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

ShowMe ( "Уничтожить по индексу", \* cp)

// Удалить объект из контейнера без удаления его при этом

cp -> Detach (& S, TSouldDelete :: NoDelete)

ShowMe ( "Не удаляя объект", \* cp)

// Удалить последний объект и, возможно, уничтожить его

int n = cp -> GetItemsInContainer ();

cp -> Detach (n-1, TSouldDelete :: DefDelete)

ShowMe ( "Последний объект", \* cp)

// Удалить все объекты из контейнера и, возможно, уничтожить

cp -> Flush ();

ShowMe ():

// Удалить контейнер, не удаляя объекты

delete cp;

return 0;

}

В программе тип *TContainer* определяется как косвенный массив строк,

реализован с помощью вектора. Тип *TIterator* - итератор для этого контейнера.

Далее объявляется указатель на тип *TContainer* и с помощью оператора new

создается объект контейнера. После этого в контейнере запоминаются несколько

строчных объектов, в т.ч. - Временный строчный объект S, который размещается в

системном стеке:

*cp> Add (& S);*

Этот прием опасен. Нужно не позволять контейнера уничтожить этот

объект. Такая операция может связать часть стекового пространства с резервом свободной

памяти вместе. Это приведет к порче стека и кучи при выделении памяти в области

верхушки стека, где хранятся адреса возврата функций. Для удаления объекта с

|  |
| --- |
| **Page 11** |

контейнера применяется оператор:

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

Аргумент 2 означает, что третий объект должен быть удален из контейнера. другие

объекты сдвигаются вверх для уплотнения освобожденного объектом места. второй

аргумент задает, что этот объект будет уничтожен, независимо от статуса владения.

Можно не уничтожать объект независимо от статуса владения при удалении

его из контейнера. В программе локальный строчный объект удаляется из контейнера

таким образом:

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

Аналогичные действия необходимо выполнить для каждого локального объекта в

косвенном контейнере до вызова любой другой функции, которая может привести к

уничтожения такого объекта. Например, если в программе изначально не выполняется

удаление из контейнера локального объекта, то оператор

*cp -> Flush ();*

испортит кучу. При вызове функции *Flush ()* удаляются все объекты, которыми

обладает контейнер. В случае, если контейнер адресует локальные объекты, то перед

вызовом функции *Flush ()* нужно или отменить владения объектами, или изъять

такие объекты из контейнера.

Можно сделать и так, чтобы объекты изымались при условии, что контейнер обладает

ними. Например, в программе удаляется последний объект, индексированный целым n.

Передача функции *Detach ()* параметра *DefDelete* указывает функции, необходимо

уничтожить его при удалении только тогда, когда контейнер владеет им. если

контейнер не владеет своими объектами, оператор удаляет из него объект, но не

освободит занимаемой этим объектом память. В конце программы уничтожается контейнер,

для которого ранее выделялась память. Но этот оператор не уничтожает все объекты,

остались в контейнере.

Для того, чтобы быть уверенным в том, что каждый объект удален, нужно

вызвать функцию *Flush ()* перед удалением контейнера или вывести новый

контейнерный класс и написать к нему деструктор, который вызывает функцию *Flush ().*