Лекция ПКШ2018\_Л5

Инициализация и зачистка объектов, хранящихся в контейнерахВо время написания программы часто бывает неизвестно, сколько объектов вам потребуется. Типичный случай – таблицы БД информационной системы.  
Можно создать в программе огромный глобальный массив – заведомо больший, чем может понадобиться программе, и выделять из него память для строк таблиц. Конечно, такие решения избавляют программиста от необходимости думать. Однако, помимо неэффективного использования памяти, работа с огромным массивом может существенно замедлиться из-за издержек, связанных с вызовами конструкторов и деструкторов.

Другой подход, поддерживаемый C++, работает гораздо лучше: когда вам потребуется объект, вы создаете его с помощью оператора new в динамически выделяемой памяти и сохраняете указатель в контейнере. В этом случае создаются только те объекты, которые действительно используются в программе.   
Кроме того, часто все данные, необходимые для инициализации объекта, недоступны при запуске программы. Например, они хранятся в файлах БД. Оператор new позволяет отложить создание объектов до создания некоторых внешних условий.   
Итак, в типичной ситуации контейнер содержит *указатели* на объекты. Программа создает объекты при помощи оператора new, сохраняет полученные указатели в контейнере и извлекает их позднее, когда с объектом нужно выполнить операцию. Этот подход позволяет создавать наиболее гибкие и универсальные программы.

Однако при использовании контейнеров, хранящих динамически созданные объекты, возникают проблемы с зачисткой памяти после удаления из них объектов.

Контейнеры являются основными компонентами программ в информационных системах. Вы уже познакомились с контейнерами map и vector на лекции 2.   
При разработке информационных систем вам практически не потребуется разрабатывать собственные контейнеры. Достаточно научиться правильно использовать контейнеры, имеющиеся в стандартной библиотеке шаблонов C++.

Шаблоны идеально подходят для реализации контейнеров. Вы создаете шаблон контейнера и указываете в программе его специализированную версию для своей задачи, а остальную работу выполняет компилятор. В результате вы получаете именно тот контейнер, который необходим для решения вашей задачи.

У шаблонного решения есть и другое преимущество: даже программист, незнакомый с наследованием или не имеющий опыта работы с ним, сможет немедленно приступить к работе с контейнерными классами. В этом вы смогли убедиться, выполняя лабораторные работы сразу после того, как только ознакомились с шаблонами контейнерных классов vector и map из стандартной библиотеки С++.

На этой лекции мы реализуем основе класса Stack (файл Stack.cpp из лекции Л4) шаблон контейнерного класса *TStack2* и на его примере рассмотрим, как можно решить проблему зачистки памяти с помощью шаблонов.

Параллельно мы познакомимся с шаблонами функций, с определением дружественного класса *iterator* в контейнерах и с перегрузкой операций.

Вот как будет выглядеть шаблон класса *TStack2* с добавлением вложенного в него дружественного класса итератора:   
//файл TStack2.h

#ifndef TSTACK2\_H

#define TSTACK2\_H

template<class T> class TStack2 {

struct Link {

T\* data;

Link\* next;

Link(T\* dat, Link\* nxt): data(dat),next(nxt){}

}\* head;

public: TStack2() : head(0) {}

~TStack2 ();

//занесение в стек указателя на объект

void push(T\* dat) {

head = new Link(dat,head);

}

//чтение содержимого вершины стека

T\* peek() const {

return head ? head->data : 0;

}

//извлечение объекта из стека с освобождением памяти

T\* pop();

// Вложенный дружественный класс итератора:

class iterator; // Необходимо объявление

friend class iterator; // Дружественный класс

class iterator { // Теперь можно определять

Link \* p;

public: iterator (const Stack2<T> &ti) : p(ti.head) {}

// Копирующий конструктор:

iterator(const iterator& ti) : p(ti.p) {}

// Конечный итератор:

iterator() : p(0) {}   
//(постфиксный и префиксный ++ для итераторов реализуются одинаково, как в STL)

bool operator++() {//постфиксный инкремент:

if(p->next)

p = p->next;

else

p = 0; // Признак конца списка

return bool(p);

}

//префиксный инкремент:

bool operator++(int) { return operator++(); }

//-----------------------------------------------------------

T\* current() const {

if(!p) return 0;

return p->data;

}

// Вызов функций объекта, на который ссылается итератор

T\* operator->() const {return current();}

T\* operator\*() const { return current(); }

// Сравнение для проверки выхода в конец контейнера:

bool operator==(const iterator&) const {

return p == 0;

}

bool operator!=(const iterator&) const {

return p != 0;

}

};//конец описания шаблона класса iterator

//внутреннее определение функций-членов шаблона класса Stack2

iterator begin() const {

return iterator(\*this);

}

iterator end() const { return iterator(); }

};//конец описания шаблона класса Stack2

//внешнее определение функций-членов шаблона класса Stack2

template<class T> TStack2<T>::~TStack2() {

while(head)

delete pop();

}

template<class T> T\* TStack2<T>::pop() {

if(head == 0) return 0;

T\* result = head->data;

Link\* oldHead = head;

head = head->next;

delete oldHead;

return result;

}

#endif // TSTACK2\_H

Обратите внимание, что объекты теперь принадлежат контейнеру, потому что ему известен их точный тип (*class T*). Эта память *динамически* выделяется методом *push(T\*dat).*   
За счет вызова (по умолчанию) деструктора после выхода контейнера из операторного блока, в котором он определен, контейнер может уничтожить свои объекты. Компилятор знает, как освобождать память, занимаемую объектами встроенных в С++ типов. Но если объект типа *class T* содержит объект или указатель на объект другого (не встроенного в С++) типа, то для его удаления должен быть вызван деструктор объекта этого типа. Цепочка вложенных вызовов деструкторов будет продолжаться до тех пор, пока не достигнет объекта, содержащего только встроенные в С++ типы. Освобождение памяти будет выполняться в обратном порядке.

Почему нельзя просто освободить память, занимаемую объектом, не вызывая деструктора? Размер памяти, занимаемой объектом *obj1*, всегда можно определить с помощью операции *sizeof(obj1)*. Но что будет, если *obj1* содержит указатель, который инициализирован адресом объекта *obj2* размером 1мбайт? Будет освобождено 4 байта, занимаемых указателем, а 1мбайт останется в ОП в качестве «мусора».

*Не забывайте, что компилятор вставляет в программу вызов деструкторов «по умолчанию». Деструктор по умолчанию ничего не делает. Для того, чтобы деструктор что-то делал, он должен быть перегружен программистом.*

В приведенном выше примере память под объекты, помещаемые в стек, динамически выделяется в функции

void push(T\* dat) {

head = new Link(dat,head);

}

Освобождение памяти выполняется деструктором

template<class T> TStack2<T>:: ~TStack2() {

while(head)

delete pop();

}

Вы должны уметь объяснить, как работают функции push(T\* dat) и ~TStack2().

Всегда ли шаблоны контейнеров могут правильно освободить динамически выделенную под объекты память? Оказывается, не всегда. Если в шаблоне в качестве параметра использован тип void\*, то компилятор не знает тип объекта, на который указывает указатель, и поэтому не может вставить в программу код вызова соответствующего деструктора.   
Это демонстрирует программа *BadVoidPointer.cpp,* которая была рассмотрена в лекции 4. В этом примере за освобождение памяти должна отвечать функция *main()*, которая инициализировала указатели *void\** типом *string*, то есть она знает тип объекта, на который будет указывать *void\**. Для освобождения памяти необходимо выполнить явное приведение типа  
delete (Object\*) b;

Аналогичная ситуация возникает при разработке макета СУБД в классе DBTableTxt.  
Строки таблицы хранятся в контейнере map<string, void\*> и тип объекта определяется только при чтении таблицы. Имя типа сохраняется в заголовке таблицы. На этапе компиляции тип не известен и поэтому контейнер не может сам освободить память.  
В этом случае освобождение памяти нужно выполнять в деструкторе класса, используя явное приведение типа.

# Шаблон контейнера TStack2<T> и класс iterator

Итератором называется объект, который позволяет перебирать содержимое контейнера и в любой момент времени указывает на один из его элементов. Итераторы занимают важное место в архитектуре стандартных контейнеров C++ и их практическом применении. Во многих отношениях итератор может рассматриваться как «умный» указатель. И действительно, итераторы поддерживают многие операции, характерные для указателей. Итератор прост и занимает минимальный объем физической памяти, равный размеру одного указателя.

При создании итератор устанавливается в начало связанного списка (указывает на вершину стека) и в дальнейшем может перемещаться только вперед (в глубину стека). Если потребуется снова вернуться к началу списка, вы создаете новый итератор, а для перехода к определенной позиции списка новый итератор создается на базе существующего, установленного в нужную позицию. Итератор можно инициализировать только значением итератора того же типа.  
 Для вызова функций объекта, на который ссылается итератор, можно воспользоваться функцией current(), оператором \* или оператором -> (стандартный прием при работе с итераторами).   
Класс iterator создается вложенным внутри контейнерного класса, содержит конструкторы для создания итератора, указывающего на элемент контейнера, и «конечного» итератора, для получения которых в классе контейнера определяются методы begin() и end().  
Реализация итератора приведена в файле TStack2.h. Задействованные имена iterator, begin() и end() позаимствованы из стандартных контейнерных классов.  
Вся реализация содержится в заголовочном файле, поэтому отдельный .срр- файл не требуется.   
Ниже приведена небольшая тестовая программа для экспериментов с итератором:

//файл TStack2Test.срр

#include <iostream>

#include <fstream>

#include <string>

#include "TStack2.h"

using namespace std;

int main() {

system("chcp 1251>nul");

ifstream fin("TStack2Test.cpp");

if(!fin.is\_open()){

cout<<"Ошибка открытия файла TStack2Test.cpp"<<endl;

system("pause");

return 0;

}

Stack2<string> textlines;

// Чтение файла и загрузка строк в Stack2:

string line;

while(getline(fin,line))

textlines.push(new string(line));

int i =0;

// Использование итератора для вывода строк из списка:

Stack2<string>::iterator it1 = textlines.begin();

Stack2<string>::iterator\* it2 = NULL;

while(it1 != textlines.end()) {

cout << \*\*it1<< endl;

it1++;

if(++i == 10) // Создание дополнительного итератора для 10-й строки

it2=new Stack2<string>::iterator(it1);//вызов конструктора копирования

}

cout << \*\*\*it2 << endl;

//cout << \*(it2->current()) << endl;

delete it2;

system("pause");

return 0;

};

Шаблон Stack специализируется для объектов string и заполняется строками файла. Затем программа создает итератор и использует его для перебора элементов. Чтобы запомнить десятую строку, мы создаем новый итератор на базе существующего, применяя копирующий конструктор; затем эта строка выводится, а динамически созданный итератор уничтожается. Здесь механизм динамического создания объекта определяет его жизненный цикл.

Контрольные вопросы и задания.

1. Перегрузите деструктор для класса DBTableTxt (в нём строки таблицы хранятся в векторе из контейнеров map<string, void\*>).