# Лекция ПКШ2018\_Л4

Классы. Выделение памяти, инициализация и зачистка объектовНа предыдущей лекции мы рассматривали процесс *проектирования* объектно-ориентированных программ, т.е. представление предметной области в виде совокупности взаимосвязанных объектов и их интерфейсов, которое было описано в виде UML-диаграмм взаимодействия классов. Это описание не является окончательным, оно может уточняться в процессе разработки программы, но уже позволяет перейти к программированию.  
На этой лекции мы продолжим изучение классов и рассмотрим вопросы, связанные с инициализацией и зачисткой динамически созданных объектов, хранящихся в контейнерах.

В процессе разработки C++ Страуструп пришел к выводу, что источником значительной части ошибок программирования являются некорректные инициализация и зачистка переменных. Такие ошибки выявляются с большим трудом. Для обеспечения гарантированного выполнения инициализации и зачистки памяти, выделяемой объектам, в состав класса он ввёл специальные методы: конструктор и деструктор. Компилятор не позволит создать или уничтожить объект без вызова конструктора или деструктора. С помощью этих методов программист может контролировать происходящие события и создавать более надежные программы.   
Появление и уход объекта из программы — такие же особые события, как рождение и смерть, поэтому компилятор всегда сам вызывает соответствующие функции, не доверяясь программисту. Другой вариант — возможность явного вызова конструкторов и деструкторов — сделал бы их вызов менее надежным.

В каком месте программы вызываются конструкторы и деструкторы объектов?   
Хотя память под объекты выделяется в начале блока операторов, вызов конструктора происходит только в точке определения объекта, поскольку идентификатор до этого момента остается недоступным. Компилятор даже проверяет, не происходит ли определение объекта в точке, мимо которой программа может пройти при некоторых условиях, например, в команде *switch* или в цепочке команд *if-else*.

Деструктор автоматически вызывается компилятором при выходе объекта из области видимости. Признаком вызова деструктора в программе является закрывающая фигурная скобка того блока, в котором находится объект.

## Класс Stack с конструктором и деструктором

Рассмотрим стек на основе связанного списка.   
Реализация связанного списка (внутри класса Stack) с применением конструкторов и деструкторов показывает, как хорошо конструкторы и деструкторы работают в сочетании с операторами new и delete. Заголовочный файл выглядит так:

//файл Stack.h

// Стек с конструктором и деструктором

#ifndef STACK\_H

#define STACK\_H

class Stack {

struct Link {

void\* data;

Link\* next;

Link(void\* dat, Link\* nxt);

~Link();

}\* head; //указатель на вершину стека

public:

Stack();

~Stack();

void push(void\* dat);

void\* peek();

void\* pop();

};

#endif // STACK\_H

// файл Stack.cpp

// Конструкторы и деструкторы

#include "Stack.h"

#include <iostream>

using namespace std;

Stack::Link::Link(void\* dat, Link\* nxt) {//только инициализация

data = dat;

next = nxt;

}

Stack::Link::~Link() { }

Stack::Stack() {

head = 0;

}

void Stack::push(void\* dat) {//выделение памяти и инициализация

head = new Link(dat, head);

}//стек растет вверх (голова ссылается на предыдущее значение)

void\* Stack::peek() {

if(head == 0){

cout<<"Stack empty"<<endl;

return 0;

}

return head->data;

}

void\* Stack::pop() {

if(head == 0)

return 0;

void\* result = head->data;

Link\* oldHead = head;

head = head->next;

delete oldHead;

return result;

}

Stack:: ~Stack() {

if(head != 0)

cout<< "Stack not empty\n";

}

Конструктор Link::Link() просто инициализирует указатели data и next, поэтому в функции Stack::push() следующая строка не только создает новый объект Link (путем динамического выделения памяти оператором new), но и инициализирует указатели в этом объекте:

head = new Link(dat, head);

Почему деструктор класса Link ничего не делает, в частности почему он не вызывает оператор delete для указателя data? По двум причинам.

Во первых, delete не может вызываться для указателя void\*, который ссылается на объект (это будет показано позже в программе BadVoidPointer.cpp ).

Во вторых, если бы деструктор класса Link удалял указатель data, то функция рор() возвращала бы указатель на удаленный объект, что заведомо является ошибкой.

Указатель не «принадлежит» контейнеру, потому что контейнер хранит его в виде типа void\* и не сможет правильно выполнить зачистку, так как не знает тип объекта, адресом которого он инициализирован. Оператор *delete* только освободит память, занимаемую *obj1.* Если этот объект *obj1* содержит указатель на объект *obj2,* то для того, чтобы исключить утечку памяти, необходимо перед удалением *obj1* выполнить деструктор объекта *obj2*.

Класс Link, а следовательно, и класс Stack, только содержит указатели, но *не может отвечать* за их освобождение. В данном примере за инициализацию указателя void\* отвечает функция main(). Говорят, что функция является владельцем объекта некоторого класса, если она выполняет инициализацию указателя void\* адресом этого объекта. Функция – владелец объекта должна отвечать и за его освобождение.

Вы должны очень хорошо понимать, кто за это отвечает. Например, если не вызвать рор() и delete для всех указателей в объекте Stack, они не будут автоматически освобождены деструктором класса Stack. В результате возможны самые серьезные последствия и утечка памяти, а незнание того, кто отвечает за освобождение объектов, может порождать ошибки в программе. Вот почему вызов Stack::~Stack() выводит сообщение об ошибке, если объект Stack на момент уничтожения содержит элементы.

В тестовой программе вы можете освободить указатели, возвращаемые функцией рор():

// файл StackTest.срр

// Конструкторы и деструкторы

#include "Stack.h"

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

int main() {

system("chcp 1251>nul");

ifstream in("Stack.h");

Stack textlines;

string line;

// Чтение файла и сохранение строк в стеке:

while(getline(in, line))

textlines.push(new string(line));

// Извлечение строк из стека и вывод:

string\* s;

while((s = (string\*)textlines.pop()) != 0) {

cout << \*s << endl;

delete s;

} //место вызова деструктора для Stack textlines

system("pause");

}

В данном примере все строки textlines извлекаются из стека и удаляются, а если бы этого не произошло, функция ~Stack() вернула бы сообщение об утечке памяти.

Класс Stack не «владеет» объектами, на которые он ссылается. Иначе говоря, выход объекта Stack из видимости не приводит к вызову delete для тех объектов data, которые в нем хранятся.

Вызов delete для указателя на void приводит только к освобождению памяти, на которую он непосредственно указывает. Из-за отсутствия информации о типе объектов data компилятор не может определить, какой деструктор нужно вызывать для освобождения памяти, занимаемой data.

Чтобы контейнеры был максимально гибкими (то есть могли хранить объекты любого типа), в них должны храниться указатели на void. Это означает, что указатель, полученный от объекта Stack, перед использованием необходимо привести к правильному типу или в программе возникнет утечка памяти. Другая потенциальная проблема с утечкой памяти заключается в том, чтобы обеспечить вызов delete для всех указателей на объекты, хранящихся в контейнере. Указатель не «принадлежит» контейнеру, потому что контейнер хранит его в виде типа void\* и не сможет правильно выполнить зачистку. Ответственность за нее должна быть возложена на пользователя.

## Вызов delete void\* как вероятный источник ошибки

Стоит особо заметить, что вызов delete void\* почти наверняка станет источником ошибки в программе, если только указатель не ссылается на очень простой объект, при создании которого не выделялась динамическая память. Перед освобождением памяти, занимаемой объектом obj1, который содержит указатель на другой объект obj2, нужно сначала вызвать деструктор obj2, а только потом удалять указатель на него в obj1. В противном случае произойдет «утечка» памяти.

Следующий пример показывает, что при этом происходит:

//BadVoidPointer.cpp

//Вызов delete для указателей на void может привести к утечке памяти

#include <iostream>

using namespace std;

class Object {

void\* data; // Некий блок памяти

const int size;

const char id; //идентификатор объекта

public:   
 Object(int sz, char c) : size(sz), id(c) {

data = new char[size];

cout << "Конструктор объекта " << id <<

". size = " << size << endl;

}

~Object() {

cout << "Деструктор объекта " << id << endl;

delete []data; // ОК. просто освобождаем память.

// вызов деструктора для char не нужен.

}

};

int main() {

system("chcp 1251 >nul");

Object\* a = new Object(40, 'a');

delete a;

void\* b = new Object(40, 'b' );

delete b;

system("pause");

return 0;

}

Класс Object содержит указатель void\*, который инициализируется неопределённым типом данных (то есть не указывает на объект, у которого имеется деструктор). В деструкторе Object оператор delete вызывается для void\* без всяких нежелательных эффектов, поскольку нам нужно лишь освободить выделенную память. Однако функция main() показывает, что на самом деле очень важно, чтобы оператор delete знал тип освобождаемого объекта. Результат выполнения программы выглядит так:

Конструктор объекта a. size = 40  
Деструктор объекта a  
Конструктор объекта b. size = 40   
Для продолжения нажмите любую клавишу . . .

Поскольку в случае команды delete а компилятор знает, что а указывает на Object, деструктор благополучно вызывается, а память, выделенная для a, освобождается. Но если работа с объектом производится через void\*, как в случае команды delete b, освобождается лишь память Object, но без вызова деструктора, поэтому блок памяти, на который ссылается b, освобожден не будет. Вероятно, при компиляции программы даже не появятся предупреждения; компилятор полагает, что вам известно, что вы делаете.

Так возникает незаметная утечка памяти.

Если в программе возникает утечка памяти, просмотрите вызовы delete и проверьте тип удаляемого указателя. Если он относится к типу void\*, то вполне вероятно, что вы обнаружили один из источников. Впрочем, в C++ имеются и другие возможности для утечки памяти).