# Глава 10.
# "Умные" указатели

МГТУ им. Н.Э. Баумана
Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети
Лектор: д.т.н., проф.
Иванова Галина Сергеевна

# 10.1 Проблемы с динамическими объектами

- при использовании указателей ответственность за выделение памяти под объект и ее освобождение целиком лежит на программисте;
- на один и тот же объект может ссылаться несколько не связанных между собой указателей, следовательно возможно наличие указателей, ссылающихся на освобождённые или перемещённые объекты;
- нет никакой возможности проверить, указывает ли ненулевой указатель на корректные данные, либо «в никуда»;
- указатель на единичный объект и указатель на массив объектов в C++ никак не отличаются друг от друга.

# Усложнение кода с учетом корректного освобождения памяти

Исходный код:

```
void proc()
{
    B *temp = new B;      // выделение памяти под объект
    B->func();            // вызов метода для объекта
    delete temp;          // освобождение памяти
}
```

# Код, освобождающий память при аварийном завершении:

```
void proc()
{    B *temp = NULL;
     try    {
          temp = new B;  // выделение памяти под объект
          B->func();     // вызов метода для объекта
     }
     catch (...)         // перехват всех исключений
     {                   // освобождение памяти при
          delete temp;   // аварийном завершении процедуры
          throw;         // возобновление исключения
     }
     delete temp;        // освобождение памяти при нормальном
                         // завершении процедуры
}
```

# Проблема освобождения памяти при ошибке в конструкторе

```
class D
{
private:
    A * a;   B * b;
public:
    D() : a(new A), b(new B) { }
    ~D() {   delete a;   delete b; }
};
```

**Деструктор вызывается только, если конструирование объекта завершено!**

**При ошибке в конструкторе часть памяти останется недоступной!!!**

# Технология *Resource Acquisition Is Initialization* (RAII) – «Захват ресурса есть инициализация».

Очевидное решение – вместо обычного указателя использовать объект-указатель, хранящий адрес и освобождающий память в своём деструкторе.

# 10.2 Шаблон auto_ptr

Методы:

- конструкторы простой и копирующий, а также деструктор;
- **`get()`** – возвращает хранимый адрес;
- **`release()`** – возвращает хранимый адрес и записывает вместо него **`NULL`** в объект-указатель;
- **`reset()`** – освобождает память, адрес которой хранится в указателе-объекте, и, если параметр – новый динамический объект указан, то записывает в него новый адрес, если параметр не указан, то адрес устанавливается в **`NULL`**.

Шаблон переопределяет операции:

- **`operator=()`** – операцию присваивания;
- **`operator*()`** – операцию доступа к объекту по адресу;
- **`operator->()`** – операцию доступа к полям объекта по адресу;
- **`operator auto_ptr<Other>`** – операцию преобразования указателя auto_ptr одного типа к указателю auto_ptr другого типа;
- **`operator auto_ptr_ref<Other>`** – операцию преобразования указателя auto_ptr одного типа к указателю auto_ptr_ref другого типа.

# Пример. Создание объекта-указателя, его использование и перезапись

```
#include <memory>
#include <iostream>
using namespace std;
class A
{
public:
   int x;
   A(int X):x(X){}
   A(){}
   ~A(){cout<<"destructor"<<endl;}
};
```

# Создание объекта-указателя, его использование и перезапись (2)

```
int main()
{  auto_ptr<A> temp1;  // неинициализированный объект-
                                    // указатель
   auto_ptr<A> temp(new A(1));  // инициализированный
                                // объект-указатель
   A &a= *temp;                 // ссылка на хранимый объект
   cout<<"a.x="<<a.x<<endl;
   A *ptr = temp.get();  // указатель на хранимый объект
   cout<<"ptr="<<ptr<<endl;
   temp.reset();  // освобождение указателя
   A *ptr1 = temp.get();  // указатель на хранимый объект
   cout<<"ptr1="<<ptr1<<endl;
   return 0;
}
```

```
a.x=1
ptr=00396520
destructor
ptr1=00000000
```

# Особенность шаблона

Шаблон реализует семантику «владения», при которой всегда существует только один объект-указатель, хранящий адрес выделенного фрагмента динамической памяти.

Для этого реализация шаблона обеспечивает «разрушающее» копирование, при котором копируемый адрес уничтожается после переписи в новый объект-указатель. Таким образом становится невозможным наличие двух и более объектов-указателей, адресующих один объект, что исключает ошибки, связанные с повторным освобождением памяти объекта.

# Пример "разрушения" указателя при копировании

```
#include <conio.h>
#include <memory>

class A { public:    void f (){}};
void main()
{  std::auto_ptr<A> temp1(new A);  // объявление и
                                   // инициализация объекта-указателя
   temp1->f();    // вызов метода - выполняется нормально
   std::auto_ptr<A> temp2(temp1);  // объявление и
                                   // инициализация второго объекта-указателя
   temp1->f();    // вызов метода – не возможен, так как адрес
                                   // разрушен при копировании
   getch();
}
```

# Ограничение

Невозможность создания копий адреса не позволяет использовать указатели, построенные по шаблону **`auto_ptr`**, для создания списковых структур, поскольку в процессе обработки список будет разрушаться. Выходом из этой ситуации является использование умного указателя с подсчётом ссылок – **`shared_ptr`**.