int\* ptr = new int(5);

delete ptr;

Начнём с того, что такое вообще указатели. Они указывают на динамическую память, где хранятся данные. С помощью оператора new мы показываем, что мы такую память под наши данные выделяем. При создании указателей важно позаботиться о том, чтобы динамическую память освобождать. Для этого мы прописываем оператор delete. Польза такого подхода заключается в том, что мы сами контролируем, где мы память выделили, где освободили, как что работает и т.д.

Однако программисты решили, что было бы неплохо не следить каждый раз за тем, где выделяется память, где используется – чтобы этот процесс происходил как-то автоматически. Для этого придумали специальные умные указатели. Это гораздо примитивней, чем, например, сборщик мусора, который есть в джаве, но работает схожим образом.

template <typename T>

class SmartPointer

{

public:

SmartPointer(T \*ptr)

{

this->ptr = ptr;

std::cout << "constructor" << std::endl;

}

~SmartPointer()

{

delete ptr;

std::cout << "destructor" << std::endl;

}

T& operator\*()

{

return \*ptr;

}

private:

T\* ptr;

};

Пропишем самый примитивный такой указатель. По сути smart pointer – это конкретная оболочка под разные типы данных. Поскольку он должен уметь работать по-хорошему даже с теми типами данных, о которых он ничего не знает, делаем шаблон. В public методах реализуем конструктор и деструктор. Конструктор – получаем указатель на объект и адрес в памяти, на который указывает этот указатель, записываем его в наш указатель. В деструкторе нужно позаботиться о том, чтобы память, на которую указывает наш указатель, очищалась.

Чтобы получить данные, перегрузим оператор \*. В результате мы должны вернуть данные, на которые указывает указатель мы должны возвращать ссылку на Т, потому что если мы вернём просто Т, у нас создастся новый объект, который находится в динамической памяти и мы получим его копию. Если мы хотим только получить данные это ещё работает, но если мы захотим изменить эти данные, то мы изменим копию. Возвращаем разыменованный указатель.

В private – указатель, оболочку для которого мы строим.

SmartPointer<int> pointer = new int(5);

\*pointer = 1553;

std::cout << \*pointer << std::endl;

Посмотрим, что наш указатель работает и нам не приходится самостоятельно очищать память.

SmartPointer<int> sp1 = new int(5);

SmartPointer<int> sp2 = sp1;

Однако с умными указателями существует проблема. Когда мы пытаемся новому указателю присвоить значение старого указателя, у нас возникает ошибка. Два указателя не могут ссылаться на одну и ту же область памяти.

#include <memory>

Посмотрим теперь на встроенные умные указатели. Для этого подключим библиотеку memory.

// auto\_ptr

std::auto\_ptr<int>ap1(new int(5));

std::auto\_ptr<int>ap2(ap1);

// There is data in ap2

std::cout << \*ap2;

Начнём с auto\_ptr. Данный код не выдаст ошибку, поскольку в момент создания ap2, ар1 теряет связь с данными и его указатель затирается и теперь работать с данными может только ар2.

std::cout << \*ap1;

Но проблему это не решает. Если нам потребуется, чтобы на одну область памяти указывали два указателя, auto\_ptr нам не поможет, поскольку в первом указателе уже ничего нет.

На самом деле стоит сказать, что auto\_ptr устарел и его никто сейчас не использует.

//unique\_ptr

std::unique\_ptr<int>p1(new int(5));

std::unique\_ptr<int>p2(p1);

// Error

Следующий unique\_ptr. В данном случае проблема решена радикально. Ситуации, когда два указателя ссылаются на одну область памяти просто запретили. Нельзя передать одному unique\_ptr другой unique\_ptr.

//function move

std::unique\_ptr<int>p1(new int(5));

std::unique\_ptr<int>p2;

p2 = move(p1);

std::cout << \*p2;

Приравнять два unique\_ptr так же нельзя:

int\* raw1 = new int(5);

std::unique\_ptr<int>uPtr(raw1);

std::unique\_ptr<int>uPtr2;

uPtr2 = uPtr;

//Error

Чтобы сменить владельца данных в случае с unique\_ptr можно использовать функцию move(). И тогда ситуация аналогичная auto\_ptr. Собственно, проблему мы так и не решили.

// shared\_ptr

std::shared\_ptr<MyClass>shPtr1 = std::make\_shared<MyClass>();

std::cout << "Shared count: " << shPtr1.use\_count() << std::endl;

//use\_count() - number of pointers for one object

{

std::shared\_ptr<MyClass>shPtr2 = shPtr1;

std::cout << "Shared count: " << shPtr1.use\_count() << std::endl;

}

//память будет освобождена, когда на объект не останется ни одного указателя

std::cout << "Shared count: " << shPtr1.use\_count() << std::endl;

Но существуют указатели, которые могут между собой владеть одними и теми же данными – shared\_ptr.

При выходе их из зоны видимости ошибки не возникает. Данные будут уничтожены в тот момент, когда будет уничтожен последний shared\_ptr, который ссылался на эти данные. Это работает так, что в классе shared\_ptr начинает работать счётчик. В момент, когда первый shared\_ptr получает ссылку на какие-то данные, переменная в классе, отвечающая за подсчёт ссылок, принимает значение 1. Когда создаётся новый объект shared\_ptr, которому присваивается значение первого указателя, у наших данных становится уже два владельца и переменная увеличивается на 1. И так каждый раз, когда добавляется новый объект типа shared\_ptr.

Проблема с уничтожением решается следующим образом:

В деструкторе класса shared\_ptr есть проверка, которая смотрит, сколько ссылок осталось на данные. Если мы уничтожаем какой-то указатель, в деструкторе вместо уничтожения данных, выполняется уменьшение счётчика. И только если счётчик уже 1, то данные уничтожаются.

Работа с объектами через shared\_ptr происходит так же, как и через сырые указатели:

int SIZE = 5;

std::shared\_ptr<int[]> ptr(new int[SIZE]);

for (int i=0; i<SIZE; i++)

{

ptr[i] = rand() % 10;

std::cout<<ptr[i]<<std::endl;

}

Итак, у нас есть автоматизированное уничтожение динамической памяти и кроме того с помощью shared\_ptr мы можем работать с разными указателями, которые владеют одними и теми же данными.

Распространенной практикой является использование shared\_ptr вместе со слабым указателем weak\_ptr. weak\_ptr не владеют данными, с помощью них мы только можем проверить, сохранился ли какой-либо объект в памяти.

//weak\_ptr

std::cout << std::endl << "WEAK POINTERS" << std::endl;

std::shared\_ptr<int> sptr = std::make\_shared<int>(10);

std::weak\_ptr<int> weak1 = sptr;

sptr.reset(new int);

\*sptr = 5;

std::weak\_ptr<int> weak2 = sptr;

// weak1 is expired!

if(auto tmp = weak1.lock())

std::cout << \*tmp << '\n';

else

std::cout << "weak1 is expired\n";

// weak2 points to new data (5)

if(auto tmp = weak2.lock())

std::cout << \*tmp << '\n';

else

std::cout << "weak2 is expired\n";

Получаем слабый указатель на данные из sptr. Методом reset() затираем данные, на которые указывает sptr, sptr становится пустым, назначаем ему новые данные. Получаем слабый указатель на эти новые данные. Методом lock() для weak1 проверяем, существует ли в памяти 10 (нет). То же самое для weak2 (объект еще существует в памяти).