# Умные указатели

Все умные указатели доступны через включение соответствующего хедера (#include <memory>). Для краткости во фрагментах кода внутри статьи это было опущено.

Помимо проблем непосредственно с new/delete, существует проблема и с простыми указателями. Она заключается в сложности разделения указателей, которые *владеют* объектом (*owning pointer*), а значит, и ответственны за вызов new/delete, и указателей, которые *используют* объект (*non owning pointer*).

При использование простых указателей (также известных как *raw pointers*) невозможно без дополнительных комментариев или дополнительного изучения кода определить, какой указатель объектом владеет, а какой – только использует.

Все вышеназванные проблемы изящно решаются умными указателями. Умные указатели в C++ – это не что-то магическое, встроенное в синтаксис языка, а не более чем набор классов из стандартной библиотеки. Разберёмся с ними один за одним.

## std::unique\_ptr

Первым умным указателем, с которым мы познакомимся, будет std::unique\_ptr. Он ссылается на объект в динамической памяти и при выходе из области видимости уничтожает хранимый объект. Взглянем на пример кода ниже:

{

std::unique\_ptr<X> ptr(new X()); //Объект класса X создан в динамической памяти

}

Здесь указатель ptr покидает свою область видимости и уничтожается, но перед этим удаляет из памяти объект, на который указывает

Когда std::unique\_ptr выходит из области видимости, утечки памяти не происходит, потому что в своем деструкторе умный указатель вызывает delete для объекта на который ссылается, высвобождая тем самым память.

Важно понять, что внутри умные указатели всё равно используют new/delete, они лишь позволяют программисту не делать этого и, как следствие, защищают его от ошибок.

От проблем с внезапными исключениями использующих умные указатели (в частности std::unique\_ptr) программистов защищает развёртывание стека (*stack-unwinding* ).

std::unique\_ptr<X> ptr(new X());

if (func()) {

func2();

return;

}

Теперь программисту не надо ставить delete, а в случае, если одна из функций выбросит исключение, развёртывание стека защитит нас от утечки памяти.

И всё бы хорошо, но мы по-прежнему используем new. Чтобы правило никогда не использовать new/delete соблюдалось, была придумана функция std::make\_unique , которая позволяет создавать std::unique\_ptr, но с несколькими дополнительными фичами:

Теперь правило никогда не использовать new/delete может быть полностью соблюдено.

* std::make\_unique позволяет не писать имя класса дважды:

Листинг 7

auto ptr = std::make\_unique<X>();

* std::make\_unique решает проблему неопределённого порядка вычисления аргументов (*unspecified evaluation order*). Рассмотрим следующий фрагмент кода:

Листинг 8

void func(std::unique\_ptr<A> a, std::unique\_ptr<B> b) {}

int main() {

func(std::unique\_ptr<A>(new A()), std::unique\_ptr<B>(new B()));

}

Здесь возможен следующий порядок вычисления аргументов:

* new A()
* new B()
* std::unique\_ptr<A>(...)
* std::unique\_ptr<B>(...)

Если при вызове new B() произойдет исключение, занятая при вызове new A() память не освободится, потому что умный указатель для этого объекта ещё не был создан, а delete никто вызывать и не собирался. Использование std::make\_unique решает подобные проблемы.

std::unique\_ptr используется тогда, когда объект должен иметь только одного владельца, однако мы можем передать право на владение кому-то другому. Чтобы это сделать, необходимо использовать std::move. Рассмотрим код:

Листинг 9

void func(std::unique\_ptr<X> a) {}

int main() {

auto a = std::make\_unique<X>();

func(a); //Не скомпилируется, std::unique\_ptr нельзя копировать, потому что иначе у объекта было бы несколько владельцев

func(std::move(a)); //Владение объектом передано в func, main больше не владеет объектом, на выходе из func объект будет уничтожен

}

Раз std::unique\_ptr нельзя копировать, становится непонятно, как разрешить кому-то использовать указываемый объект, не передавая ему право на владение. Очень просто: нужно всего лишь использовать простой указатель на созданный объект, который можно получить с помощью метода get(), как в листинге 10. Однако будьте внимательны: не допускайте ситуации, когда std::unique\_ptr вместе с хранимым объектом были уничтожены, но где-то по-прежнему находится простой указатель указывающий на невалидную (уже) область памяти.

Делайте так, чтобы простые non owning указатели переставали существовать до того, как перестанет существовать объект, на который они указывают.

Листинг 10

void func(X\* a) {}

int main() {

auto a = std::make\_unique<X>();

func(a.get());

}

Когда в коде используются простые указатели и умные, сразу становится понятно, где указатель владеет объектом, а где – только использует.

std::experimental::observer\_ptr

На самом деле в коде с умными указателями не все простые указатели только используют объекты – некоторые из них могут ими владеть. Чтобы полностью отделить non owning, полученные через get() простые указатели от любых других простых указателей, был придуман std::experimental::observer\_ptr, но на данный момент он ещё не вошёл в стандарт.

std::unique\_ptr – это умный указатель, о котором вы должны подумать в первую очередь, когда решите разместить что-нибудь в динамической памяти. Это ваш умный указатель по умолчанию.

std::shared\_ptr и std::weak\_ptr

std::unique\_ptr и правда хорош, но он не поможет в ситуации, когда мы хотим, чтобы несколько объектов работали с одним общим ресурсом и чтобы в момент, когда все эти объекты были выгружены из памяти, за ненадобностью автоматически выгрузился бы и ресурс.

В такой ситуации необходимо использовать std::shared\_ptr . Этот умный указатель разрешает объекту иметь несколько владельцев, а когда все владельцы уничтожаются, уничтожается и объект. Такое поведение достигается за счёт наличия специального счётчика ссылок внутри std::shared\_ptr. Каждый раз, когда такой указатель копируется, счётчик инкрементируется, а когда один из указателей уничтожается – декрементируется. В момент, когда счётчик достигает нуля, объект уничтожается. Посмотрим на код:

Листинг 11

{

std::shared\_ptr<X> ptr = std::make\_shared<X>(); //Создаётся объект

{

std::shared\_ptr<X> ptr2 = ptr; //Теперь у объекта два владельца, выраженных в виде ptr и ptr2

} //ptr2 выходит из области видимости, но объект не освобождается, потому что есть ptr, который по-прежнему ссылается на него

} //ptr выходит из области видимости, и объект уничтожается

## std::make\_shared

является аналогом std::make\_unique для std::shared\_ptr.

Существуют ситуации, когда объект A должен ссылаться на B, а B – на A. Это называется циклической ссылкой (*cyclic reference/circular dependency*). В таком случае оба объекта никогда не будут выгружены из памяти.

Чтобы разорвать цикличность, необходимо использовать std::weak\_ptr. Это фактически умный указатель non owning, предназначенный для использования именно с std::shared\_ptr. Копирование std::weak\_ptr не увеличивает счётчик в std::shared\_ptr, а значит и не защищает объект от уничтожения. При этом всегда имеется возможность проверить, существует ли ещё объект, на который ссылается std::weak\_ptr, или нет. Внимание на код:

Листинг 12

class Owner {

public:

std::shared\_ptr<X> owningPtr;

Owner() {

owningPtr = std::make\_shared<X>();

}

};

class User {

std::weak\_ptr<X> usingPtr;

public:

User(std::weak\_ptr<X> object) {

usingPtr = object;

}

void use() {

if (std::shared\_ptr<X> object = usingPtr.lock()) { //Попытка получить оригинальный std::shared\_ptr из std::weak\_ptr, если возвращён пустой std::shared\_ptr, значит, объект уже был удалён

object->func();

} else {

//Объект уже удалён

}

}

};

int main() {

Owner owner;

User user(owner.owningPtr);

user.use();

}

## Выводы

Каждый программист на C++ должен уметь использовать умные указатели. Умные указатели – это ваш способ управления динамической памятью по умолчанию. std::unique\_ptr – это ваш умный указатель по умолчанию. Использование умных указателей не противоречит использованию простых указателей, в случае, если последние используют объекты, а не владеют ими. std::auto\_ptr – зло.

https://proglib.io/p/umnye-ukazateli-v-c-2021-08-26