МИНИСТЕРСТВО РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

«КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ»

РЕФЕРАТ

по дисциплине: Основы алгоритмизации и программирования

на тему: Умные указатели С++

Выполнил:

студент 1 курса 7 группы

факультет ИТ

Кишко Иван Петрович

Проверил:

доцент, к.т.н.

Белодед Николай Иванович

Минск 2020

Оглавление

[Чем плохи обычные указатели? 3](#_Toc41857050)

[Умные указатели 4](#_Toc41857051)

[unique\_ptr 4](#_Toc41857052)

[Shared\_ptr 6](#_Toc41857053)

[Тест производительности 8](#_Toc41857054)

# Чем плохи обычные указатели?

l. Их объявление не дает информации о том, указывают ли они на один объект или на массив.

2. Их объявление ничего не говорит о том, должны ли вы уничтожить то, на что он указывает, когда завершите работу, т.е. владеет ли указатель тем, на что указывает.

3. Если вы определили, что должны уничтожить то, на что указывает указатель, нет никакого способа указать, как это сделать. Должны ли вы использовать delete или имеется иной механизм деструкции (например, специальная функция уничтожения, которой следует передать этот указатель)?

4. Если вам удалось выяснить, что требуется использовать оператор delete, то причина l означает, что нет никакого способа узнать, следует ли использовать оператор для удаления одного объекта (delete) или для удаления массива (delete[]). Если вы используете оператор неверного вида, результат будет неопределенным.

5. Если вы определили, что указатель владеет тем, на что указывает, и выяснили, каким образом уничтожить то, на что он указывает, оказывается очень трудно обеспечить уничтожение ровно один раз на каждом пути вашего кода (включая те, которые возникают благодаря исключениям). Пропущенный путь ведет к утечке ресурсов, а выполнение уничтожения более одного раза - к неопределенному поведению.

6. Обычно нет способа выяснить, не является ли указатель висячим, т.е. не указывает ли он на память, которая больше не хранит объект, на который должен указывать указатель. Висячие указатели образуются, когда объекты уничтожаются, в то время как указатели по-прежнему указывают на них.

Совет - **ВСЕГДА** предпочитайте обычным указателям, интеллектуальные.

# Умные указатели

Почти те же указатели, только умнее:

1. Часто поддерживают тот же интерфейс, что и обычные указатели :op->,op\*
2. Сами управляют временем жизни объекта – вовремя вызывают деструкторы и освобождают память

В этом и заключается их польза:

1. Автоматическое освобождение памяти при удалении самого указателя
2. Безопасность исключений

Популярные умные указатели:

1. std::unique\_ptr
2. std::shared\_ptr
3. std::weak\_ptr(помощник shared\_ptr)

Все они находятся в библиотеке <memory>

## unique\_ptr

1. владеет бъектом эксклюзивно(т.е. другие указатели не могут ссылаться на этот объект)
2. При деструкции ненулевой std::uni que\_pt r освобождает ресурс, которым владеет. По умолчанию освобождение ресурса выполняется с помощью оператора delete, примененного ко встроенному указателю в std::unique ptr.
3. Нельзя копировать, но можно перемещать
4. Удобно использовать при возврате из функции
5. Легко преобразуется в shared\_ptr
6. Есть функция release()(благодаря этой функции указатель становится нулевым,при обычном обнулении указателя данные должны затереться)

Использует шаблон следующего вида:

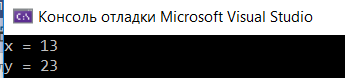
template<  
 **typename** T,  
 **typename** Deleter = std::default\_delete<T>  
> **class** unique\_ptr;

std::default\_delete<T> является функциональным объектом, который вызывает delete при вызове. Но это только тип по умолчанию для Deleter, и его можно изменить для пользовательского удаления.

Cоздается unique\_ptr так:

**#include**<memory>  
**#include**<iostream>  
**using namespace** std;  
**struct** Point {  
 **int** x;  
 **int** y;  
 Point(**int** x,**int** y)  
 {  
 **this->**x = x;  
 **this->**y = y;  
 }  
};  
  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 std::unique\_ptr<Point> p1(**new** Point(13,23));  
 cout << **"x = "** << p1**->**x<<endl;  
 cout << **"y = "** << p1**->**y<<endl;  
 **return** 0;  
}

Результат:



А такой код не скомпилируется(Должен владеть объектом эксклюзивно!!!)

#include <iostream>

#include<memory>

using namespace std;

int main()

{

setlocale(LC\_ALL, "ru");

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

unique\_ptr<int>p2(p1); *//ошибка компиляции*

return 0;

}

Но обычно используют шаблон make\_unique. Использование std::make\_unique() является необязательным, но рекомендуется вместо использования std::unique\_ptr. Дело в простоте. Кроме того, std::make\_unique() решает проблему безопасности использования исключений, которая может возникнуть в результате неопределённого порядка обработки аргументов функции (так как С++ явно не указывает этот порядок).

Правило: Используйте std::make\_unique() вместо создания std::unique\_ptr и использования оператора new.

**#include**<memory>  
**#include**<iostream>  
**using namespace** std;  
**struct** Point {  
 **int** x;   
 **int** y;  
 Point(**int** x,**int** y) *//конструктор с параметрами*  
 {  
 **this->**x = x;  
 **this->**y = y;  
 }  
};  
  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **auto** p1 = make\_unique<Point>(13,23); *//p1 - unique\_ptr*  
 cout << **"x = "** << p1**->**x<<endl;  
 cout << **"y = "** << p1**->**y<<endl;  
 **return** 0;  
}

## Shared\_ptr

1. Поддерживает общий счетчик ссылок на выделенный объект
2. Удаляет объект только тогда,когда последний из ссылающихся на этот объект shared\_ptr’ов удаляется или принимает указатель на другой объект
3. Указатель можно как перемещать, так и копировать.
4. Размер std::shared \_ptr в два раза больше размера обычноrо указателя, поскольку данный интеллектуальный указатель содержит обычный указатель на ресурс и другой обычный указатель на счетчик ссылок
5. Освобождение ресурсов по умолчанию выполняется с помощью оператора delete, однако поддерживаются и пользовательские удалители. Тип удалителя не влияет на тип указателя std::shared\_ptr.

Использует шаблон следующего вида:

template< **class** T > **class** shared\_ptr;

Работа с ним мало отличается от работы с unique\_ptr, за тем исключением, что shared\_ptr можно смело копировать.

Интересные грабли при использовании shared\_ptr заключаются в том, что с его помощью можно создать циклические ссылки. Например, есть два объекта. Первый ссылается при помощи shared\_ptr на второй, а второй — на первый. Даже если ни на один из объектов нет других ссылок, счетчики ссылок никогда не обнулятся, и объекты никогда не будут уничтожены.

Эта проблема обходится при помощи weak\_ptr, так называемого слабого указателя. Класс weak\_ptr похож на shared\_ptr, но не участвует в подсчете ссылок. Также у weak\_ptr есть метод lock(), возвращающий временный shared\_ptr на объект.

Пример особенности (общий счетчик):

**#include**<memory>  
**#include**<iostream>  
**using namespace** std;  
**void** foo()  
{  
 std::shared\_ptr<**int**> sp\_a(**new int**(1));  
 cout << **"count of owners= "** << sp\_a.use\_count()*//количество ссылающихся указателей* << endl; *// 1* {  
 std::shared\_ptr<**int**> sp\_b(sp\_a);  
 cout << **"count of owners= "** << sp\_a.use\_count()*//количество ссылающихся указателей* << endl; *// 2* {  
 std::shared\_ptr<**int**> sp\_c;*//блок,при выходе из которого указатель sp\_c исчезнет* sp\_c = sp\_b;  
 cout << **"count of owners= "** << sp\_a.use\_count()*//количество ссылающихся указателей* << endl; *// 3* }  
 cout << **"count of owners= "** << sp\_a.use\_count()*//количество ссылающихся указателей* << endl; *// 2* }  
 cout << **"count of owners= "** << sp\_a.use\_count()*//количество ссылающихся указателей* << endl; *// 1*}  
  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 foo();  
 **return** 0;  
}

## Тест производительности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Обычный указатель | shared\_ptr | make\_shared | unique\_ptr | make\_unique |
| 4.56521 | 18.4388 | 17.6444 | 8.90934 | 17.3979 |

Тестируемый код:

**#include** <chrono>  
**#include** <iostream>  
  
**static const long long** numInt = 10000000;  
  
**int** main() {  
 **auto** start = std::chrono::system\_clock::now();  
  
 **for** (**long long** i = 0; i < numInt; ++i) {  
 **int**\* tmp(**new int**(i));  
 **delete** tmp;  
 *// std::shared\_ptr<int> tmp(new int(i));  
 // std::shared\_ptr<int> tmp(std::make\_shared<int>(i));  
 // std::unique\_ptr<int> tmp(new int(i));  
 // std::unique\_ptr<int> tmp(std::make\_unique<int>(i));* }  
  
 std::chrono::duration<**double**> dur = std::chrono::system\_clock::now() - start;  
 std::cout << **"time native: "** << dur.count() << **" seconds"** << std::endl;  
}