Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Стратегії інтелектуальних указників

Текстова частина до курсової роботи

за спеціальністю «Комп'ютерні науки та інформаційні технології»- 122

Керівник курсової роботи

Кандидат фізико-математичних наук,

доцент

Бублик В.В.

_____

(підпис)

"____" _____ 2019 р.

Виконав студент КНІТ-4:

Баранов Д.О.

"____" _____ 2019 р.

Київ 2019

Календарний план виконання роботи:

Тема: Стратегії інтелектуальних указників

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової	09.10.2019	
2.	Пошук тематичної наукової літератури	20.10.2019	
3.	Ознайомлення з науковою літературою	11.12.2019	
4.	Повторення матеріалів про указники	25.12.2019	
5.	Повторення синтаксису C++	01.01.2020	
6.	Різні стратегії указників	30.01.2020	
7.	Визначення структури програми	05.02.2020	
8.	Написання першої частини курсової роботи	1.04.2020	
9.	Написання другої частини курсової роботи	15.04.2020	
10.	Написання висновків курсової роботи	16.04.2020	
11.	Перегляд змісту роботи з керівником	17.04.2020	
12.	Внесення змін до роботи	18.04.2020	
17.	Створення презентації	19.04.2020	
18.	Завантаження курсової роботи	19.04.2020	

Зміст

Вступ

Постановка задачі

Опис різних стратегій для реалізації інтелектуальних указників, порівняння їх в нових редакціях C++.

Вимоги:

1. Опис різних стратегій інтелектуальних указників.
2. Опис і вирішення проблем різних стратегій.
3. Інтелектуальні указники і багатопоточність.
4. Приклади їх реалізації.
5. Порівняння їх з різними редакціями C++.

Коротко про проблеми «звичайних» указників та навіщо потрібні «інтелектуальні» указники:

Інтелектуальні указники це клас який імітує синтаксис і семантику звичайного указника і виконує дуже багато іншої корисної роботи. Головна особливість інтелектуальних указників – це змога без втрат і суттєвого перетворення програмного коду замінити звичайні указники.

Але все-таки навіщо потрібні інтелектуальні указники? Інтелектуальні указники мають семантику значень, а звичайні – ні. Об’єкт має семантику значень, коли його можна спокійно копіювати і присвоювати. Наприклад числа Int.

Зовсім інша ситуацію якщо створити об’єкт:

```
1. widget  *p = new Widget;
```

В цьому випадку змінна p не тільки указує на об’єкт Widget а й володіє їм. І якщо написати delete p то об’єкт буде видалиний а пам’ять звільниться. Але якщо написати наступний вираз:

```
1. widget  *p = new Widget;
2. p = 0;
```

То володіння об’єктом буде втрачено і ми ніяк не зможемо повернути цей об’єкт.

Інша проблема з’являється якщо скопіювати указник p в іншу змінну. Ми отримаємо два указника які відсилаються до одного і того самого об’єкту в пам’яті. В наслідок цього може виникнути ситуація коли буде видалено один і той же самий об’єкт два рази – критична помилка.

```
1. Widget* p= new Widget;
2. Widget* p2 = p1;
3. delete p1;
4. delete p2;
```

Тож інтелектуальні указники (в залежності від використання) вирішують ці або інші проблеми. Вони управляють володінням об’єктом.

Наприклад стандартний інтелектуальний указник std:auto_ptr після копіювання новому указнику присвоює значення об’єкта а старому нуль.

Типи інтелектуальних указників:

1. Тип зберігання (storage type) – це тип змінної _pointee, це тип звичайних указників.
2. Тип указника (pointer type) – це тип об’єкта, що повертається оператором [->].
3. Тип відсилки (reference type) – це тип, що повертається оператором [*].

Розділ 1. Стратегії володіння

1.1. Глибоке копіювання

Володіння об’єктом – це причина існування інтелектуальних указників. Вони самі видаляють об’єкти на які посилаються.

При копіюванні найпростіша стратегія – це просто копіювання об’єкта на який він посилається. При цьому на кожен об’єкт буде посилатися лише один указник. Ось так це буде виклядати:

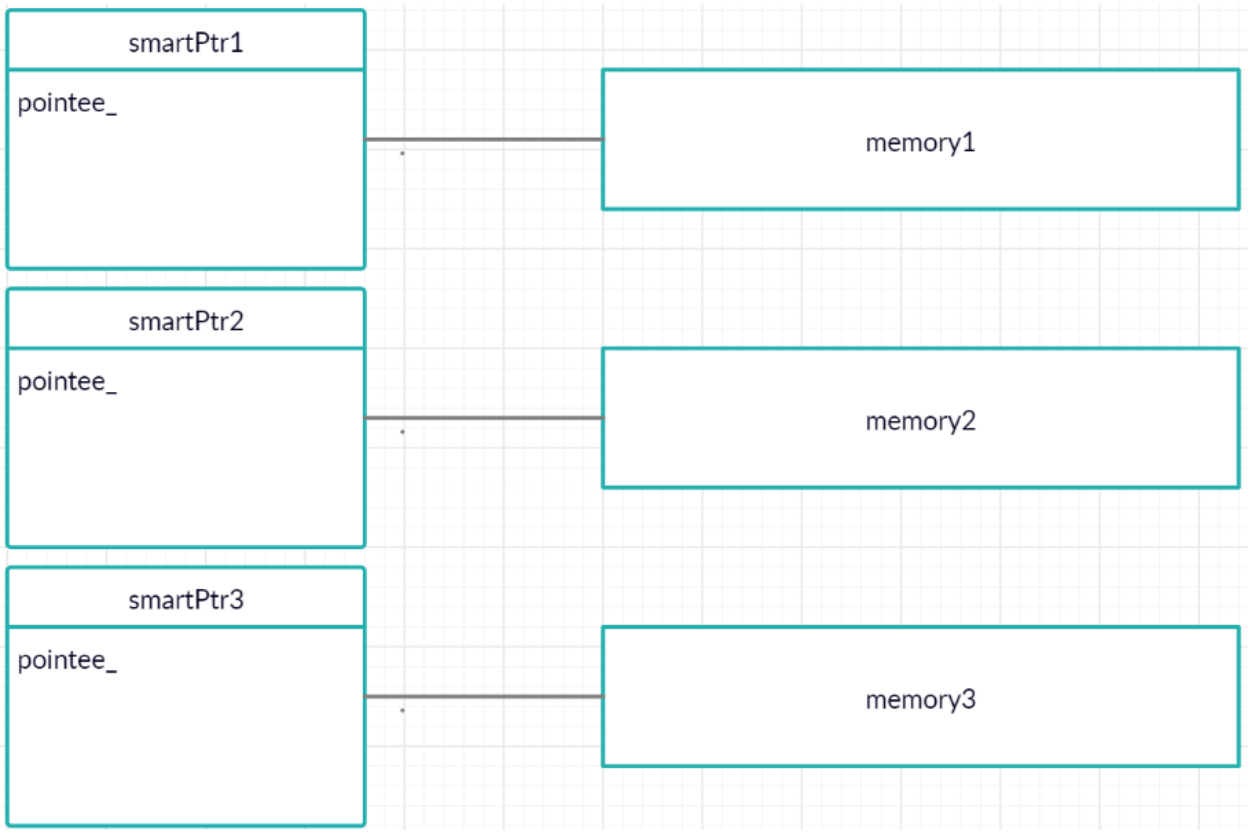

Рисунок 1 Схема розподілу пам’яті для указників при глибоку

Глибоке копіювання застосовується для підтримки поліморфізма, а саме безпечна передача поліморфних об’єктів. Реалізувати це можна наступним чином [1]:

```
1. template <class T>
2. class SmartPtr
3. {
4. public:
5. SmartPtr(const SmartPtr& other)
6. : _pointee(new T(*other._pointee))
7. {}
8. };
```

Однак при такій реалізації представимо що ми копіюємо об’єкт типу SmartPtr<Widget>, а його інтелектуальний указник other посилається на екземпляр класу OtherWidget, клас унаслідуваний від Widget. При такому копіюванні відбудеться зрізання (slicing) і копіюється лише та частина класу OtherWidget яка унаслідувана від Widget. Ця проблема вирішується створенням віртуальної функції Clone.

1.2. Копіювання при запису

Copy on write - це спосіб оптимизації, який заключається, що поки ми не будемо модифікувати об’єкт на нього може посилатися багато указників, але при спробі модифікування, об’єкт буде клоновано.

Проблема закладається в тому що інтелектуальні указники не розуміють коли ми звертаємося до константних і не константних функцій і не може вирішити використовувати стратегію COW чи ні:

```
1. class Widget
2. {
3. public:
4. void constFunc() const;
5.         void nonConstFunc();
6. };
7. SmartPtr<Widget> smrPtr1;
8. smrPtr1->constFunc();
9. smrPtr1->nonConstFunc();
```

1.3. Підрахунок відсилок

Підрахунок відсилок – це найпопулярніша стратегія при якій рахуються інтелектуальні указники і якщо їх кількість дорівнює нулю то об’єкт видаляється. Проте необхідно щоб на один і той же об’єкт не посилалися звичайні і інтелектуальні показники.

Даний підхід можна реалізувати декількома способами. Наприклад кожен інтелектуальний указник може зберігати указник на число відсилок об’єкта. Схематично це виглядає так:

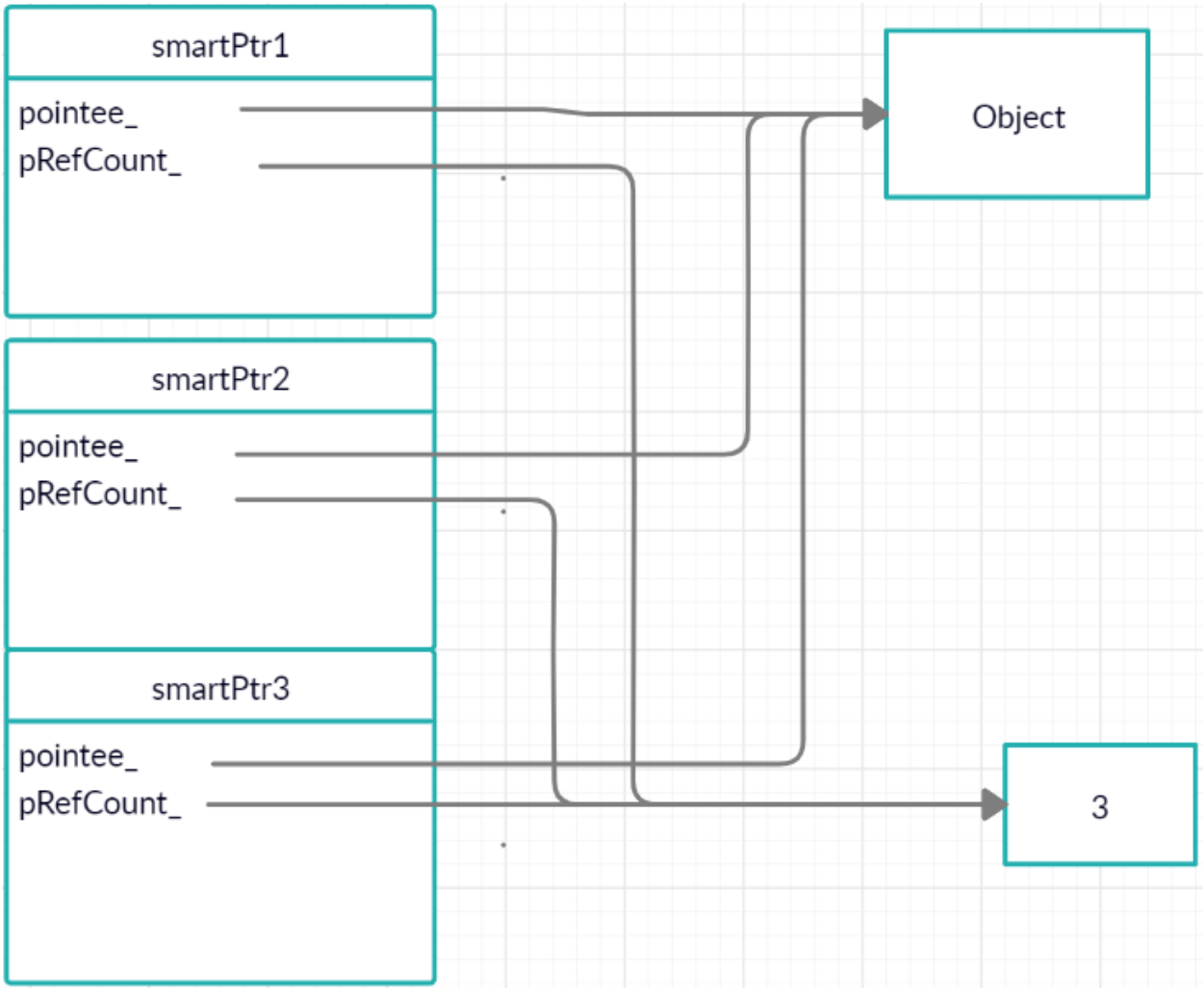

Рисунок 2 Три інтелектуальних указника які посилаються на один об'єкт

Цю схему можна оптимізувати, якщо зберігати кількість указників і об’єкт в окремій структурі а указники будуть зберігати відсилку до цієї структури:

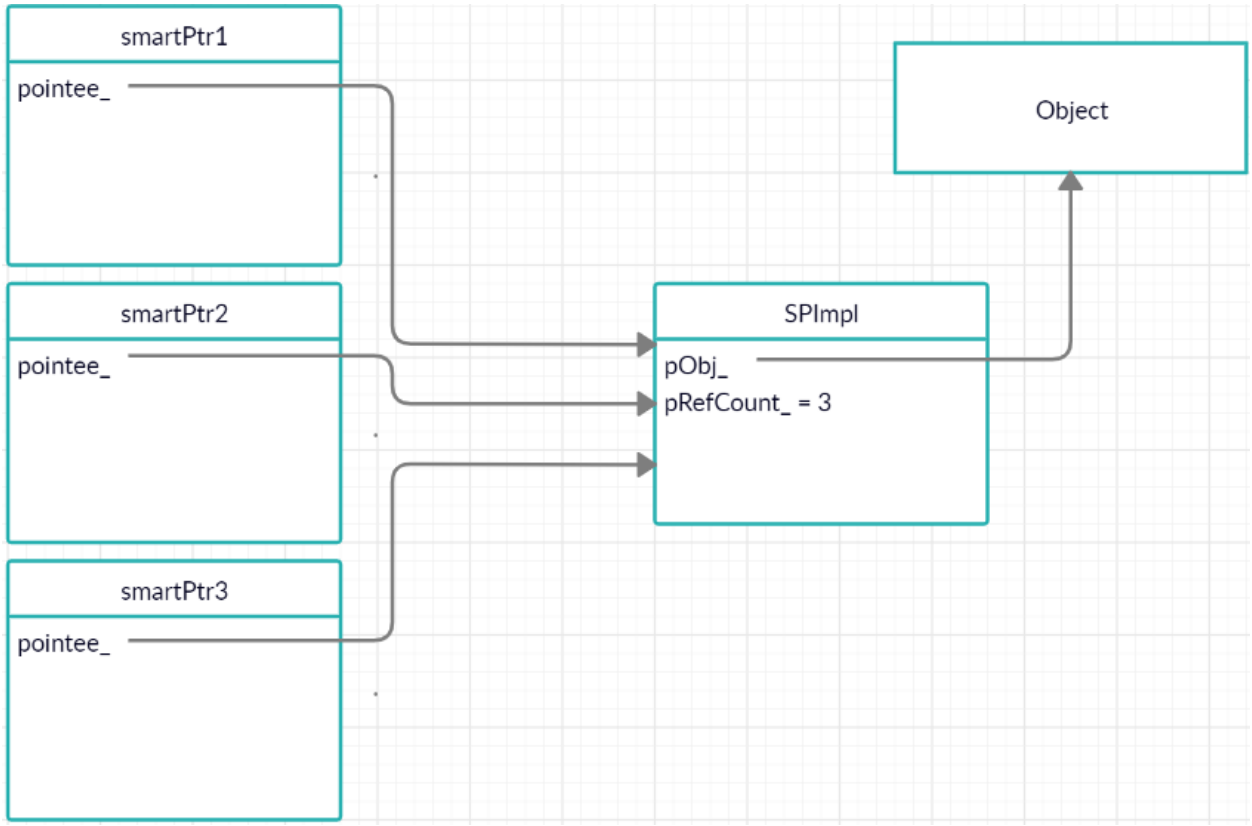

Рисунок 3 Три указника, що посилаються на один об'єкт(альтернатива)

Ця структура дозволяє зменшити швидкість доступу. Об’єкт на який посилається інтелектуальний указник створює новий рівень непрямої адресації. Однак це значний недолік, бо інтелектуальні указники використовуються багато разів а знищуються і створюються лише один раз.

Однак, найкращий спосіб це зберігати підрахунок відсилок в самому об’єкті(хоч для цього необхідно змінити сам об’єкт, що не є добрим тоном):

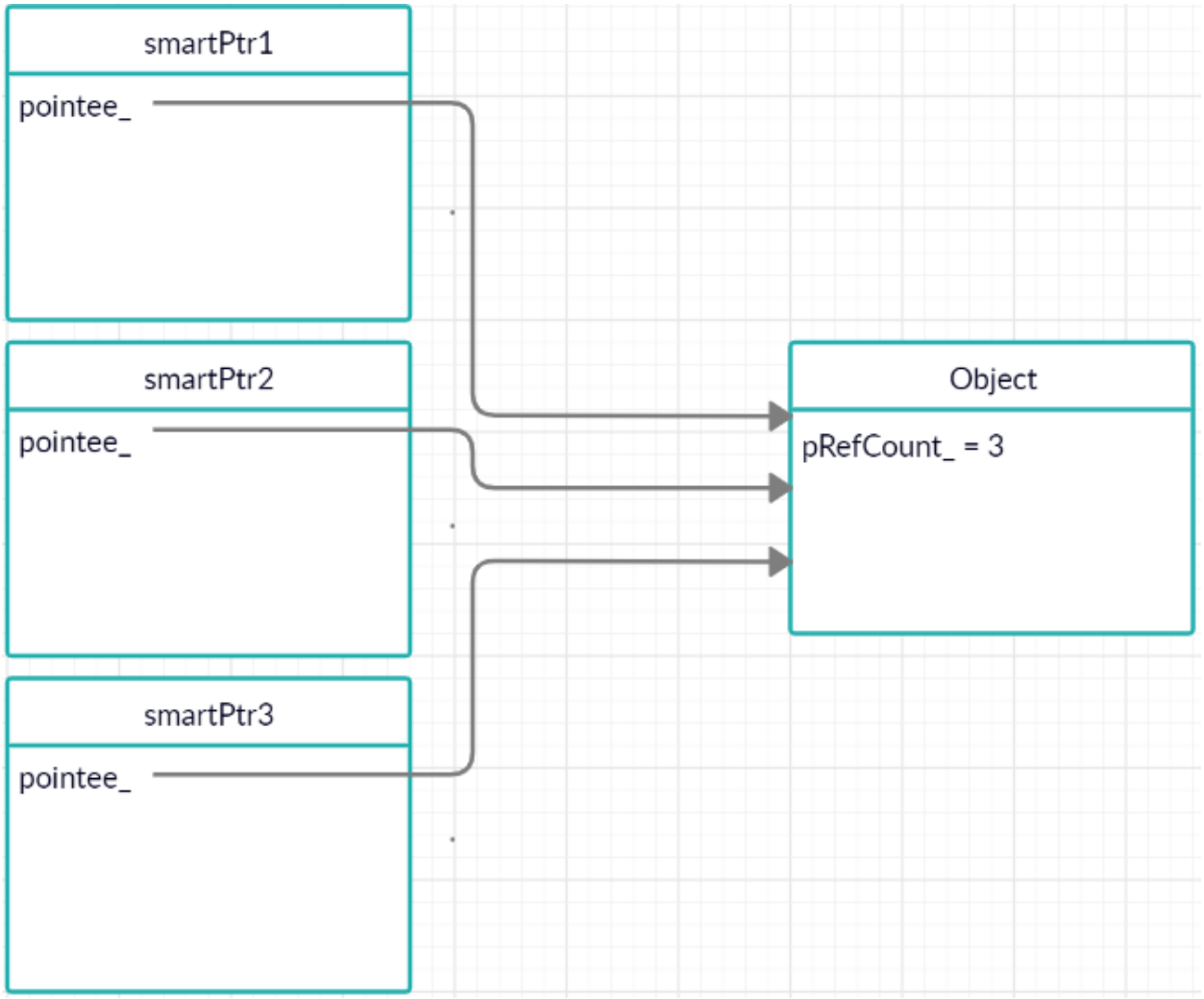

Рисунок 4 Три указника на один об'єкт(intrusive reference counting)

В цьому випадку інтелектуальні указники будуть займати пам’ять як звичайні указники. Це називається впровадженим способом підрахунку відсилок. І очевидно що при першій можливості необхідно використовувати цей спосіб підрахунку.

1.4. Зв’язування відсилок

Цей тип з’явився через те, що немає необхідності підраховувати число відсилок. Потрібно лише дізнатися коли їх число буде нулем. Тож всі об’єкти класа SmartPtr потрібно занести в двохсторонній зв’язаний список. Нові об’єкти заносяться в список, а деструктор видаляє об’єкти зі списку. І коли список дорівнює нулю об’єкт на який відсилалися ці указники видаляється.

Тож в інтелектуальний указник потрібно додати два додаткові указники на наступний і попередній елемент:

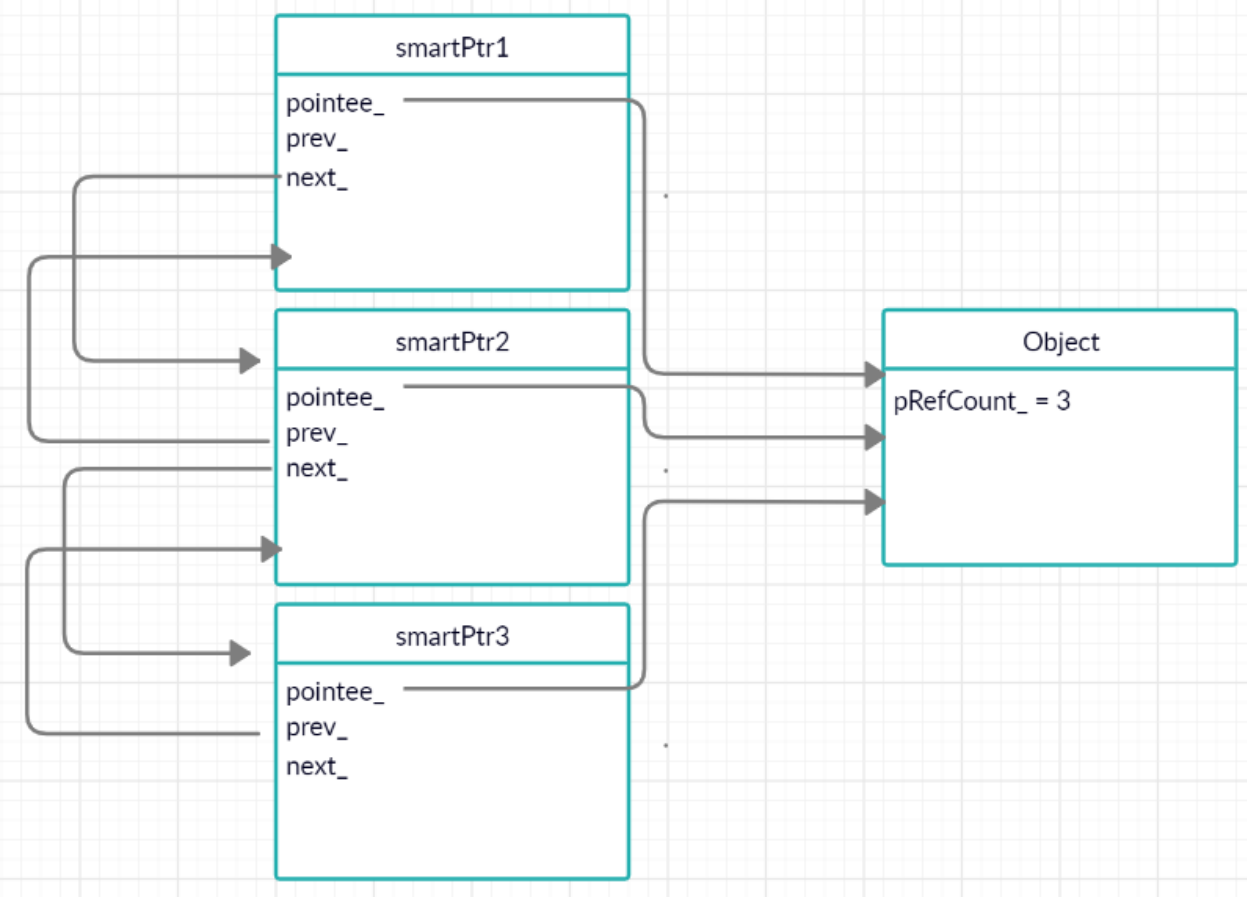

Рисунок 5 Зв'язування відсилок

Ця стратегія дуже надійна, але має ряд недоліків. Серед яких: складність управління списком, більше пам’яті при реєстрації, втрачання ресурсі(циклічні посилання). І хоч об’єкти ніким не використовуються, вони самі використовують один одного.

1.5. Руйнівне копіювання

Руйнівне копіювання – підчас копіювання оригінал знищується. Раніше в стандартному інтелектуальному указнику std:auto_ptr застосовується саме такий підхід.

```
1.  template <class T>
2.  class SmartPtr
3.  {
4.  SmartPtr(SmartPtr& src)
5.     {
6.        _pointee = src._pointee;
7.        src._pointee = 0;
8.     }
9.  SmartPtr& operator=(SmartPtr& src)
10.   {
11.     if (this != &src)
12.       {
13.          if(_pointee != 0)
14.          delete _pointee;
15.          _pointee = src._pointee;
16.          src._pointee = 0;
17.        }
18.  return *this;
19.   }
20. };
```

Але що вийде, якщо виконати наступний фрагмент:

```
1.  void display(SmartPtr<Widget> smrPtr);
2.  SmartPtr<Widget> smrPtr(new Widget);
3.  display(smrPtr);
```

Після виклику в третьому рядку, указник smrtPtr зберігає нульовий адрес. Це називається ефектом коловорота – все що попаде в цю функцію буде знищено. Що означає, що цей указник неможливо помістити в контейнер. Тим не менш ця стратегія дозволяє не витрачати додаткової пам’яті.

1.6. Багатопотоковість і інтелектуальні указники

Оскільки інтелектуальні указники дозволяють спільне використання об’єктів, вони тісно пов’язані багатопотоковістю. Взаємодія між ними відбувається на двох рівнях: pointee object level, bookkeeping data level.

Коли на один об’єкт посилаються багато потоків і багато указників, які можуть бути в різних потоках. Необхідно використовувати об’єкт-замінитель, або просто proxy object. Тобто ідея полягає в тому щоб указники посилалися на деякий об’єкт наприклад LockingProxy, який предназначений для керування механізмом Lock/Unlock. Ось так, наприклад, це може виглядати.

Існує деякий клас Widget:

```
1.  class Widget
2.  {
3.      void lock();
4.      void unlock();
5.  };
```

Далі створюємо шаблонний клас який буде керувати ним:

```
1.  template <class T>
2.  class LockingProxy
3.  {
4.      public:
5.      LockingProxy(T* pObj): pointee_(pObj)
6.      {
7.          pointee_->lock();
8.      }
9.      ~LockingProxy();
10.     {
11.         pointee_->unlock();
12.     }
13.     T* operator->() const
14.     {
15.         return pointee_;
16.     }
17.     private:
```

```
18.     LockingProxy& operator=(const LockingProxy&);
19.     T* pointee_;
20. };
```

Також додаємо обгортку над цим класом – інтелектуальний указник:

```
1.  template <class T>
2.  class SmartPtr
3.  {
4.      public:
5.      LockingProxy<T> operator->() const;
6.      {
7.          return LockingProxy<T>(pointee_);
8.      }
9.      private:
10.     T* pointee_;
11. };
```

Коли ми напишемо наступний вираз:

```
1.  SmartPtr<Widget> smrPtr = new Widget;
2.  sp->doSmth();
```

То створиться екземпляр класу LockingProxy<T> який буде володіти об’єктом класа Widget. І після завершення виконання виклику функції LockingProxy буде видалено а об’єкт Widget буде звільнено.

Однак наприклад, якщо спробувати реалізувати підрахунок відсилок в різних потоках. При копіюванні інтелектуальних указників з різних потоків їх каунтер буде змінюватися непередбачувано. Так само буде і з звязаними указниками. Ці проблеми вирішуються атомарними операціями

1.7. Клас SmartPtr

```
template
<
typename T,
template <class> class OwnershipPolicy = RefCounted,
class ConversionPolicy = DisallowConversion,
template <class> class CheckingPolicy = AssertCheck,
template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

OwnershipPolicy – задає стратегію володіння:

- DeepCopy – 1.1
- RefCounted – 1.3
- RefCountedMT - 1.3
- COMRefCounted - 1.3
- RefLinked - 1.4
- NoCopy
- DestructiveCopy 1.5

ConversionPolicy – визначає, чи допускається неявне перетворення в звичайні указники: AllowConversion. DisallowConversion.

CheckingPolicy – стратегія перевірки помилок: AssertCheck, AssertCheckStrict, RejectNullStatic, RejectNullStrict, NoChek.

StoragePolicy – механізм зберігання і доступу до об’єктів: DefaultSPStorage, ArrayStorage, LockedStorage, HeapStorage.

Розділ 2. Інтелектуальні указники в нових редакціях с++

2.1. Ще раз про інтелектуальні указники

Головна мета інтелектуальних указників – це копіювання поведінки звичайних указників і вирішення проблем через які використання звичайних указників інколи приводить до критичних помилок.

Головні причини чому звичайні указники не підходять[2 - 125]:

- Їх об’явлення не дає інформацію про об’єкт який зберігається (масив чи ні)
- Нічого не говориться про те чи необхідно знищити те на що відсилається указник
- Який механізм видалення використати та яку функцію delete чи delete[]?
- Неможливо точно бути впевненим, якщо про пункти вище все відомо, чи буде видалено об’єкт лише один раз
- Чи став цей указник висячим?

В C++11 існує чотири варіанта інтелектуальних указників: std::auto_ptr, std::unique_ptr, std::shared_ptr, std::weak_ptr.

Як було сказано у вступі std::auto_ptr це стара версія інтелектуальних указників яку було створено в C++98. Вона реалізує стратегію з деструктивним копіюванням тобто знищує оригінал і залишає копію. Тобто при копіюванні указник перетворювався в нуль і його неможливо зберігати в контейнерах. Семантика копіювання замінювалася семантикою переміщенняю., що і відбулося з виходом указника в с++11 std::unique_ptr.

Цей указник буквально в усьому кращий за старий, і єдина причина для якої потрібен std::auto_ptr це необхідність компілювання коду компіляторо с++98.

2. 2. Інтелектуальний указник повного володіння std::unique_ptr

Інтелектуальний указник повного володіння std::unique_ptr має той же розмір що і звичайний указник і більшість операцій виконуються за той же час.

Оскільки цей указник підтримує семантику повного володіння то ненульовий указник завжди володіє тим на що вказує. Переміщення це передача володіння від поточного указника до цільового. І початковий указник стає рівним нулеві. Копіювання не підтримується бо тоді буде створено два указника які будуть володіти одним і тим же об’єктом. Таким чином std::unique_ptr це лише преміщуваний тип. Звільнення ресурсом відбувається через оператор delete зазвичай:

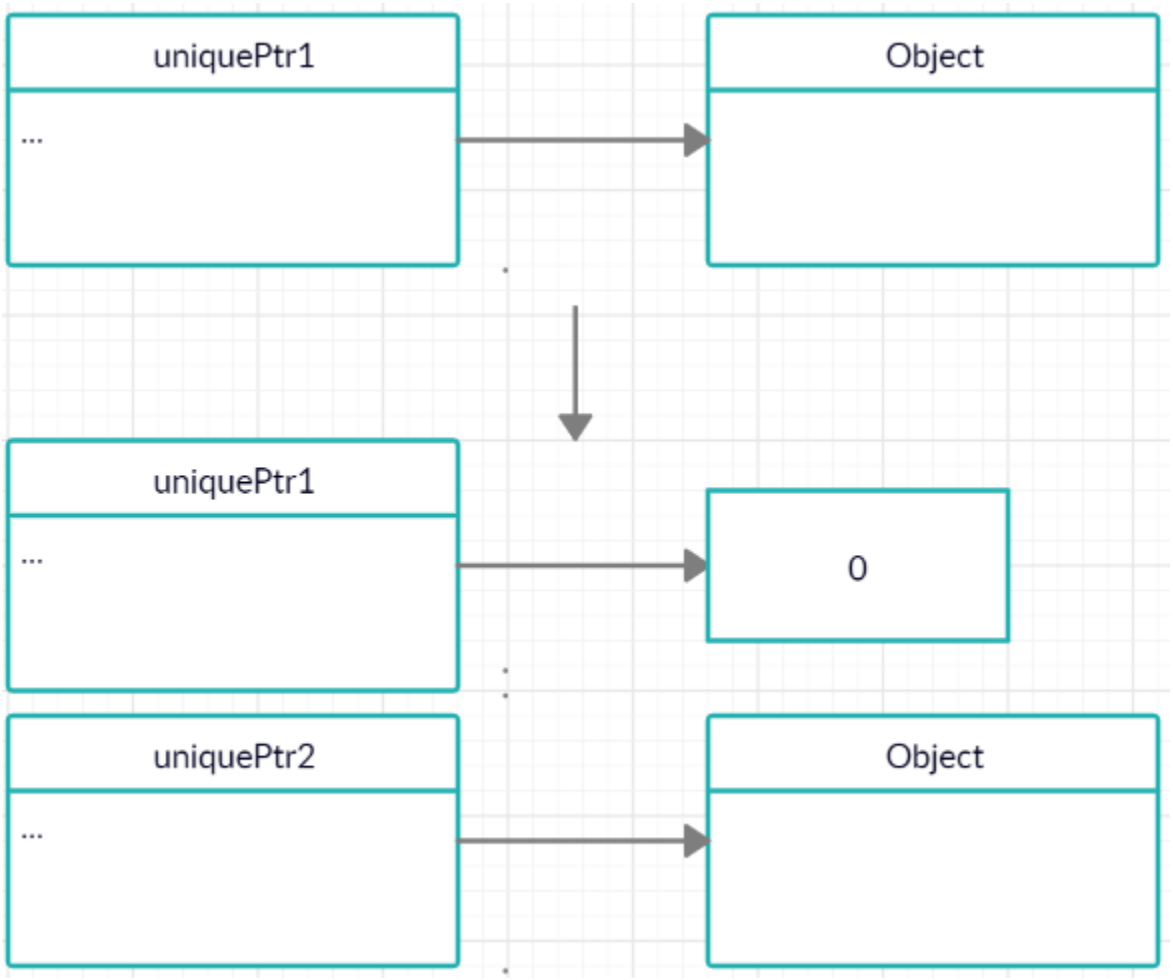

Рисунок 6 Приклад роботи std::unique_ptr

Як говорилось вище std::unique_ptr використовують зазвичай стандартну процедуру видалення через delete, але є можливість використовувати клієнтські «видалятори» які передаються в другому аргументі std::unique_ptr. Їх часто використовуються в фабричних функціях, коли перед видаленням потрібно зробити якийсь запис.

Цей указник має дві різновидності один для масивів а інший для індивідуальних об’єктів std::unique_ptr<T[]> і std::unique_ptr<T> відповідно. Тож немає неясності коли ми використовуємо масиви, а коли ні. Проте використовувати std::unique_ptr для масивів завжди гірше ні звичайні std::array, std::vector і std::string, які завжи поводять себе краще.

Тож, std::unique_ptr це швидкий і маленький інтелектуальний указник який використовує принцип повного володіння і використовується лише для переміщення. Також одна з головних особливостей це те, що цей указник завжди можна перетворити в std::shared_ptr.

2.3. Інтелектуальний указник спільного володіння std::shared_ptr

В світі програмування завжди йшли дискусію над тим використовувати автоматичні системи збирання сміття (як в Java) чи потрібно досконало керувати часом звільнення ресурсів і передбачуваністю деструкторів? Спроба поєднати ці два поняття і створила std::shared_ptr.

[3] Інтелектуальний указник std::shared_ptr – об’єкт до якого ці указники посилаються має час життя, управління яким здіснюється через стратегію спільного володіння.

Як і з збиранням сміття клієнти не переживають за час існування об’єкту.

Цей указник це покращена версія підрахунку відсилок описаним в розділі 1. Конструктор std::shared_ptr збільшує рахунок відсилок, а деструктор зменшує. Оператор копіювального присвоювання робить і те і інше. І коли кількість відсилок дорівнює нулю цей ресурс звільнюється.

Деякі особливості std::shared_ptr:

- Розмір std::shared_ptr в два рази більше звичайного указника. Оскільки указник зберігає додатково звичайний указник на рахунок відсилок
- Пам’ять для підрахунку повинна виділятися динамічно
- Інкремент і декримент повинні бути атомарні – через підтримку роботи в різних потоках

Але особливістю в редакції с++11 є те що підрахунок відсилок збільшується не завжди. Після виклику конструктора і конструктора з присвоєння виконується переміщуване конструювання . Тобто як і std::unique_ptr в std::shared_ptr після копіювання старий указник перестає указувати на об’єкт і тому рахунок не збільшується.

Інша відмінність від попереднього указника – це те що функції видалення не є частиною типа указника:

```
auto loggingDel = [] (Widget *pw) //Лямбада вираз
{
    makeLogEntry (pw) ;
    delete pw;
};
std::unique_ptr<Widget,decltype(logqinqDel)>
 upw (new Widget,loggingDel); // видалення частина типу
std::shared_ptr<Widget>
spw (new Widget,loggingDel) //видалення не є частиною типа
```

Тож можливо помістити указники з різними «делетерами» в контейнер

```
auto cstDel1 [] (Widget *pw) {...};
auto cstDel2 [] (Widget *pw) {...};

std::shared_ptr<Widget> pwl (new Widget, cstDel1);
std::shared_ptr<Widget> pw2 (new Widget, cstDel2);

std::vector<std::shared_ptr<Widge t>> vpw { pwl, pw2 };
```

Структура указника спільного володіння виглядає цікаво. Передача клієнтський видалячів не збільшує розмірів указника, оскільки воно разом с лічильником зберігається в окремій стрктурі:

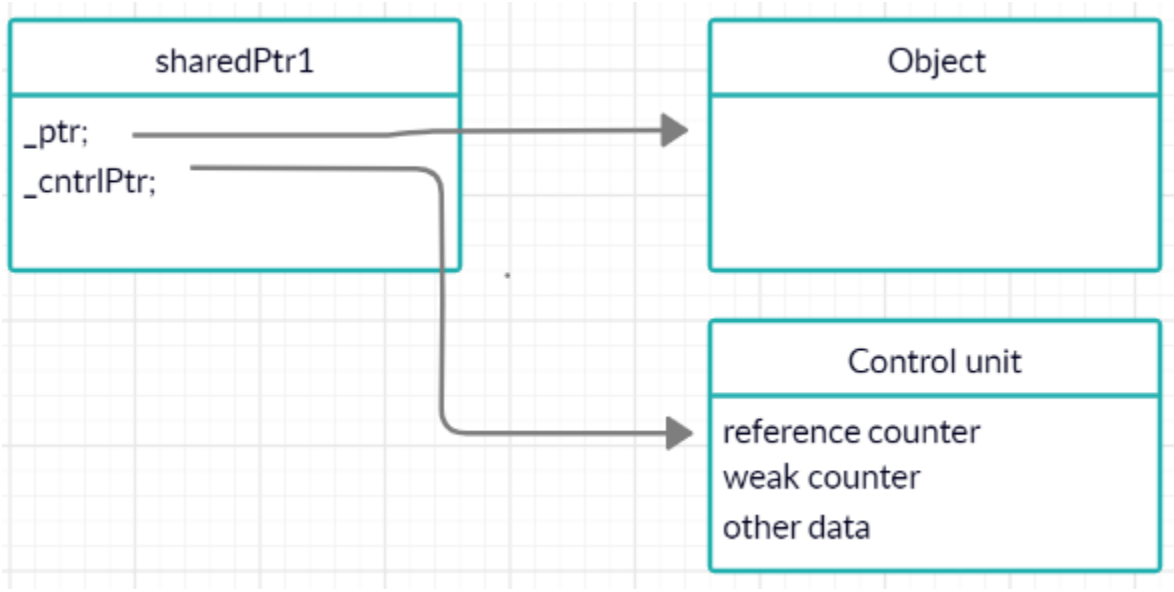

Рисунок 7 схема std : : shared ptr

Керуючий блок створюється після створення першого указника std::shared ptr. І по очевидних причинах керуючий блок повинен бути лише один. Тому виникають такі правила при створенні керуючого блоку:

- Функція std::make_shared завжди створює керуючий блок
- Керуючий блок створюється коли std::shared ptr створюється з указника з повним володінням (std::unique_ptr, std::auto_ptr)
- Керуючий блок не створюється, якщо він уже існує. Тобто в якості аргументу для std::shared ptr було передано std::shared ptr чи std::weak_ptr

Відразу до проблем, якщо створити два std::shared_ptr з звичайного указника то створиться два керуючих блока для одного об’єкту що приведе обов’язково до видалення вже видаленого об’єкту.

```
1. Widget *p = new Widget;
2. std::shared_ptr<Widget> spwl(p,cstDel1);
3. std::shared_ptr<Widget> spw2(p,cstDel1);
```

Вирішення цієї проблеми – це не передавати звичайні указники в std::shared_ptr, а якщо це необхідно то робити це без використання проміжної змінної p. А одразу писати через new Widget.

2.4. Для інтелектуальних указників які можуть бути висячими- std:weak_ptr

Необхідність мати указник як shared_ptr, але не приймати участь в спільному володінні ресурсом. Цей указник повинен боротися з проблемами які не може вирішити shared_ptr. std:weak_ptr намагається вирішити їх.

Іноді об’єкт на який посилається shared_ptr був знищений і справжній інтелектуальний указник повинен відстежити коли він став висячим.

weak _ptr – не може бути розіменованим чи перевіреним на нуль. Створюється він з shared_ptr. Але він вже не впливає на лічильник відсилок

Розділ 3. Приклади роботи інтелектуальних указників та їх проблеми

3.1. Безпека потоку shared_ptr

[4]"Кілька потоків можуть одночасно читати і записувати різні об'єкти shared_ptr, навіть коли об'єкти є копіями, які мають спільне володіння."

Об'єкт неможливо читати і записувати одночасно оскільки [5] shared_ptr<>- це механізм, який забезпечує власників декількох об'єктів гарантію того, що об'єкт знищений, а не механізм для забезпечення правильного доступу до об'єкта кількох потоків.

```
1. void thread_fcn()
2. {
```

Це безпечно для потоків і буде добре працювати, хоча і марно. Багато короткочасних указників буде створено та знищено.

```
3.      for (int i = 0; i < 10; i++)
4.      {
5.          shared_ptr<int> temp = global_instance;
6.      }
7.
```

Це не є безпечним для потоків. Хоча всі потоки однакові, "остаточне" значення цього майже точно не буде дорівнювати number_of_threads * 10 = 30. Це буде щось інше. Якщо додати більше потоків і збільшити цикл

```
8.      for (int i = 0; i < 10; i++)
9.      {
10.         *global_instance = *global_instance + 1;
11.     }
12.     cout << *global_instance << endl;
13. }
```

3.2 Приклад руйнування циклічних залежностей, які виникли через shared_ptr за допомогою weak_ptr

[6]

Припустиму, що у нас є код

```
1.  class Bar;
2.  class Foo
3.  {
4.  public:
5.      Foo() { std::cout << "Foo()" << std::endl;}
6.      ~Foo() { std::cout << "~Foo()" << std::endl;}
7.      std::shared_ptr<Bar> bar;
8.  };
9.  class Bar
10. {
11. public:
12.     Bar() { std::cout << "Bar()" << std::endl; }
13.     ~Bar() { std::cout << "~Bar()" << std::endl; }
14.     std::shared_ptr<Foo> foo;
15. };
16.
17. int main()
18. {
19.     auto foo = std::make_shared<Foo>();
20.     foo->bar = std::make_shared<Bar>();
21.     foo->bar->foo = foo;
22.     return 0;
23. }
```

При виході з блоку main видаляються локальні об’єкти. Тут локальним об’єктом є foo але ресурс не буде видалений бо на нього є посилання з боку Bar і так далі.

Рішення - weak_ptr працює не напряму, але створює shared_ptr, за допомогою метода lock(). Таким чином звільнимо наш ресурс.

```
std::shared_ptr<Foo> ptr = std::make_shared<Foo>();
std::weak_ptr<Foo> w(ptr);

if (std::shared_ptr<Foo> foo = w.lock())
{
    foo->doSmth();
}
```

Потоково безпечний зв’язаний список

```
#pragma once

#include <memory>
#include <atomic>

template <typename T> class ThreadLinkedList
{
struct Node
{
T value;
std::shared_ptr<Node> next;
};
std::shared_ptr<Node> head;
ThreadLinkedList(const ThreadLinkedList &) = delete;
void operator = (const ThreadLinkedList &) = delete;
public:
ThreadLinkedList() = default;
~ThreadLinkedList() = default;
```

для повернення значень у find () та front (), тому воно повинно бути загальнодоступним

```
class reference
{
std::shared_ptr<Node> p;
public:
reference(std::shared_ptr<Node> &&pp) : p(pp)
```

```
24. {
25. }
26. T& operator * ()
27. {
28. return p->t;
29. }
30.
31. T* operator -> ()
32. {
33. return &p->t;
34. }
35. };
```

повертає посилання на об'єкт, який обгортає shared_ptr <Node> замість повернення shared_ptr <Node>

```
36. reference find(const T &t) const
37. {
38.
39. auto p = std::atomic_load(&head);
40.
41. while (p && p->t != t)
42. p = p->next;
43. return reference(std::move(p));
44. }
45. reference front() const
46. {
47. return reference(std::atomic_load(&head));
48. }
49. void push(T t)
50. {
51. auto p = std::make_shared<Node>();
52. p->t = t;
53. p->next = std::atomic_load(&head);
54. while (!std::atomic_compare_exchange_weak(&head, &p->next, p))
55. {
56. }
57. }
58. void pop()
59. {
```

```
60. auto p = atomic_load(&head);
61. //compare m_head to p, set p to m_head if false, set m_head to p-
    >next if true, in an atomic operation
62. while (p && !atomic_compare_exchange_weak(&head, &p, p->next))
63. {
64. }
65. }
66. };
```

Висновок

Інтелектуальні указники є потужним інструментом для контролю над пам’яттю. Різні стратегії їх реалізації можуть бути застосовані в залежності від теми завдання.

«Використовуйте інтелектуальні указники скрізь де б ви використовували звичайні указники»[7]

Підсумуємо переваги інтелектуальних указників:

- Звичайні указники завжди без втрат можна замінити на інтелектуальні
- Інтелектуальні указники володіють об’єктом
- Різноманітність сфер застосування
- Звільняють від необхідності повного керування пам’яттю.
- Збір сміття
- Зберігають інформацію про об’єкт на який посилаються
- Зрозуміло коли використовувати delete delete[]
- Вирішують проблему висячих указників
- Атомарні указники дозволяють використовувати їх безпечно в багатопотоковому середовищі

Список літератури:

1. alexandrescu modern c++ design c 140
2. Skott_Meyers_Effektivnyiy_i_sovremennyiy_C++ Part 4.
3. Skott_Meyers_Effektivnyiy_i_sovremennyiy_C++ Part 4.
4. https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/c9ceah3b(v=vs.100)?redirectedfrom=MSDN
5. https://stackoverflow.com/questions/14482830/stdshared-ptr-thread-safety
6. http://archive.kalnytskyi.com/2011/11/02/smart-pointers-in-cpp11/
7. alexandrescu modern c++ design c P7
8.