Лекция 5. Интелектуальные указатели

ИУ8

September 29, 2016

Содержание

На текущей лекции рассмотрим:

- exceptions
- smart pointers

Ключевые слова

- try
- throw
- catch

Азы работы с исключением

```
1  void some_function() {
2    throw std::logic_error();
3  }
4
5  try {
6    some_function();
7  } catch(std::logic_error & e) {
8    std::cout << e.what();
9  } catch(std::exceprion & e) {
10   std::cout << e.what();
11 }
```

Вложенная обработка исключений

```
try {
  try {
    throw std::logic_error();
  } catch(std::logic_error & e) { std::cout << e.what(); }
} catch(std::exceprion & e) {std::cout << e.what(); }
```

Проброс исключения

```
try {
  try {
    throw std::logic_error();
  } catch(std::logic_error & e) {
    std::cout << e.what();
    throw;
  } catch(std::runtime_error & e) { std::cout << e.what(); }
} catch(std::exceprion & e) { std::cout << e.what(); }
```

Если есть try, должен быть и catch

```
try {
  some_function();
} // compilation error
```

На один try выполняется только один блок catch

```
try {
  throw std::runtime_error();
} catch(std::runtime_error & e) {
  throw std::logic_error();
} catch(std::exceprion & e) {
  std::cout << e.what();
}
```

Проверка на тип исключения идет сверху вниз

```
try {throw std::runtime_error();}
catch(std::exceprion & e) { std::cout << e.what();}
catch(std::runtime_error & e) {
  // never execute
  std::cout << e.what();
}
```

```cpp
struct Sample {
  Sample() {
    std::cout << "Sample::Sample"
        ;
  }
  ~Sample() {
    std::cout << "Sample::~Sample
        ";
  }
};
// ...
try {
  Sample obj;
  throw std::logic_error();
} catch(...) {
  std::cout << "catching";
}
// std::cout :
// Sample::Sample
// Sample::~Sample
// catching
```

```cpp
struct Sample {
  Sample() {
    std::cout << "Sample::Sample"
        ;
  }
  ~Sample() {
    std::cout << "Sample::~Sample
        ";
  }
};
// ...
try {
  Sample obj;
  throw std::logic_error();
}
// call destructors of all
    objects
// created in a block 'try'
catch(...) {
  std::cout << "catching";
}
```

Утечка ресурсов

```
1 try {
2   int * arr = new int[1000];
3   throw std::logic_error();
4   delete[] arr;
5 }
6 catch(...) {
7 }
```

```
1 int * arr = nullptr;
2 try {
3   arr = new int[1000];
4   throw std::logic_error();
5 } catch (std::logic_error&) {
6   delete[] arr;
7 } catch(...) {
8   delete[] arr;
9 }
```

RAII

Идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.

Smart pointers

Smart pointer - объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, реализует идиому RAII и предоставляет некоторый дополнительный функционал.

В стандарте C++11 появились следующие "умные" указатели: **unique_ptr**, **shared_ptr** и **weak_ptr**. Все они объявлены в заголовочном файле <**memory**>.

Библиотека **boost** предоставляет дополнительные 4 клааса умных указателей – **boost::scoped_ptr**, **boost::scoped_array**, **boost::shared_array** и **boost::intrusive_ptr**.

auto_ptr

Еще один тип умного указателя, который достался в наследство от C++98 это auto_ptr. Если вы разрабатываете на C++11, то никогда и нигде **не используйте auto_ptr**. На замену ему пришел более совершенный **unique_ptr**.

std::unique_ptr

Класс **std::unique_ptr** воплощает в себе семантику исключительного владения.

По умолчанию, std::unique_ptr имеет тот же размер, что и обычные указатели, и для большинства операций выполняются точно такие же команды. Это означает, что такие указатели можно использовать даже в ситуациях, когда важны расход памяти и времени.

std::unique_ptr всегда владеет тем, на что указывает. Перемещение std::unique_ptr передает владение от исходного объекта целевому.

Копирование std::unique_ptr **запрещено**. std::unique_ptr является **исключительно перемещаемым типом**.

"Умные" указатели std::unique_ptr легко и эффективно можно преобразовать в std::shared_ptr.

Освобождение ресурса

При разрушении объекта ненулевой std::unique_ptr освобождает ресурс, которым владеет. По умолчанию освобождение ресурса выполняется через оперетор delete. Но данное поведение можно настроить при создании std::unique_ptr.

Custom deleters

При конструировании std::unique_ptr можно указать произвольную функцию (или функциональный объект, он же функтор), которая будет вызываться для освобождения ресурса.

Все функции пользовательских удалителей принимают обычный указатель на удаляемый объект и затем выполняют все необходимые действия по его удалению.

Указатель на функции или функтор?

Когда пользовательский удалитель может быть реализован как функция или как лямбда-выражение, то реализация в виде лямбда выражения предпочтительнее.

Удалители, которые реализованы через *указатели на функции*, которые в общем случае **приводят к увеличению размера** std::unique_ptr. Для удалителей, являющихся функциональными объектами, изменение размера зависит от того, какое состояние хранится в функциональном объекте. *Функциональные объекты без состояний* (например, получающиеся из лямбда-выражений без захватов) **не приводят к увеличению размеров**.

Пример 1

```
struct Image {
  Image(const std::string & path) { /*...*/ }
};
auto pImg = std::make<Image>("~/Puss_in_Boots.png");
```

Пример 2

```
auto customDeleter = [](Image * p) {
  std::cout << "debug mode";
  delete p;
};
std::unique_ptr<Image,
  decltype(customDeleter)>
    pImg {nullptr, customDeleter};
pImg.reset(new Image("~/Puss_in_Boots.png"));
```

Пример 3

```
std::unique_ptr<int[]> array = std::make<int[]>(1000);
```

std::shared_ptr

Указатель **std::shared_ptr** используется для управления ресурсами путем **совместноro** впадения

Никакой конкретный указатель std::shared_ptr не владеет объектом, на который указывают. Все указатели std::shared_ptr, сотрудничают для обеспечения гарантии, что уничтожение целевого объекта произойдет в точке, где он станет более ненужным.

Размер std::shared_ptr в два раза больше размера обычноro указателя

Класс std::shared_ptr имеет API, предназначенное только для работы с указателями на единственные объекты. Не существует std::shared_ptr<T[]>.

Счетчик ссылок

Указатель std::shared_ptr может сообщить, является ли он последним указателем, указывающим на ресурс, с помощью **счетчика ссылок**, значения, связанного с ресурсом и отслеживающего, какое количество указателей std::shared_ptr указывает на него.

Память для счетчика ссылок должна выделяться динамически. Счетчик ссылок связан с объектом, на который указывает std::shared_ptr, однако сам целевой объект об этом счетчике ничего не знает и в нем нет места для хранения счетчика ссылок.

Конструкторы std::shared_ptr увеличивают этот счетчик (в случае конструктора перемещения это не так)
Деструкторы std::shared_ptr уменьшают его
Операторы копирующего присваивания делают и то, и другое (у копируемого объекта счетчик увеличивается, у получателя уменьшается)

Производительность

Инкремент и декремент счетчика ссылок должны быть **атомарными**. Атомарные операции обычно медленнее неатомарных, так что несмотря на то, что обычно счетчики ссылок имеют размер в одно слово, следует рассматривать их чтение и запись как относительно дорогостоящие операции.

При передаче std::shared_ptr по значению происходит его копирование и к его внутреннему счетчику прибавляется единица, это нужно сделать атомарно, что влияет на производительность. Передавайте std::shared_ptr по ссылке, где возможно.

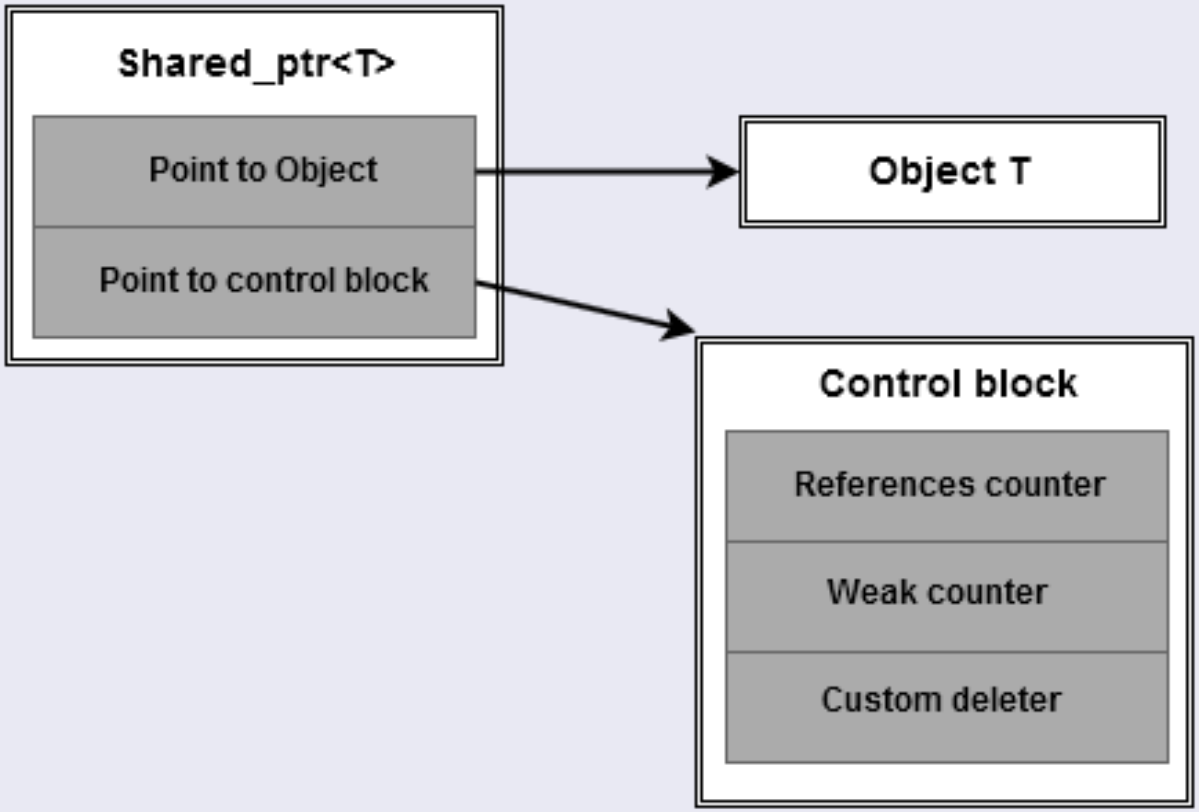
Custom deleter

Подобно std::unique_ptr, std::shared_ptr в качестве механизма удаления ресурса по умолчанию использует delete

Для std::shared_ptr тип удалителя НЕ является частью типа интеллектуального указателя. Это делает std::shared_ptr более гибким указателем. Например, можно добавить два std::shared_ptr с разными деаллокаторами в один вектор, или присвоить один другому.

Другим отличием от std::unique_ptr является то, что указание пользовательского удалителя не влияет на размер объекта std::shared_ptr. Независимо от удалителя объект std::shared_ptr имеет размер, равный размеру двух указателей.

Control block



Управляющий блок имеется для каждого объекта, управляемого указателями std::shared_ptr.

Создание управляющего блока

Управляющий блок объекта настраивается функцией, создающей первый указатель std::shared_ptr на объект.

- функция std::make_shared всегда создает управляющий блок;
- управляющий блок создается тогда, когда указатель std::shared_ptr создается из указателя с исключительным владением;
- когда конструктор std::shared_ptr вызывается с обычным указателем, он создает управляющий блок.

std::make_shared

Утечка ресурсов

```
1 void someFunc(std::shared_ptr<Widget> ptr, int command);
2
3 someFunc(std::shared_ptr<Widget>(new Widget),
4     readCommand());
```

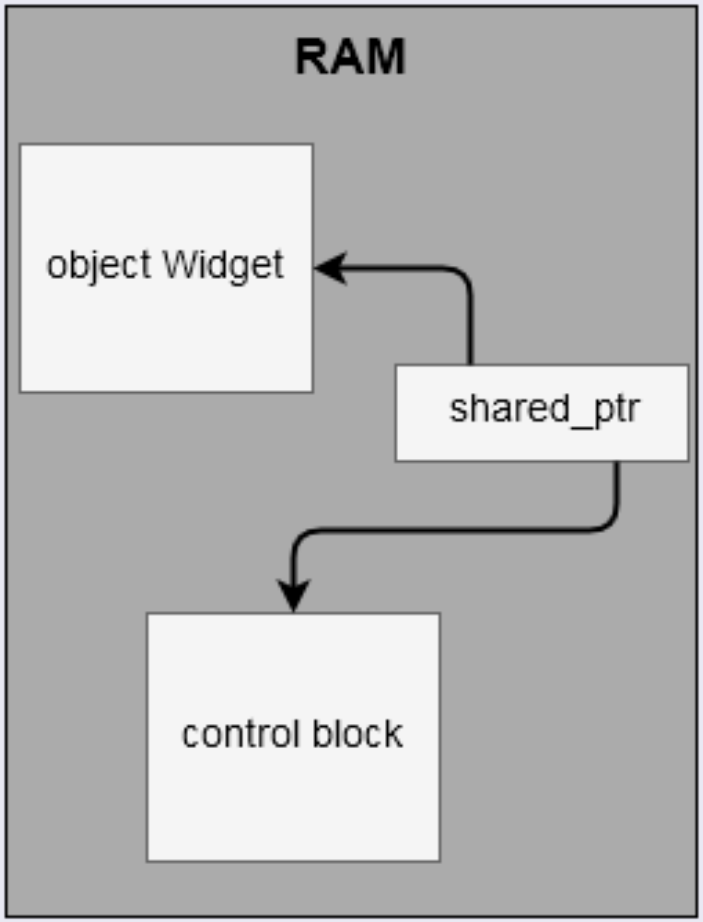
Возможна следущая ситуация:

- создается новый Widget
- выполняется readCommand(), которая генерирует исключение
- утечка памяти, созданный Widget, никогда не будет удален

Утечка ресурсов невозможна

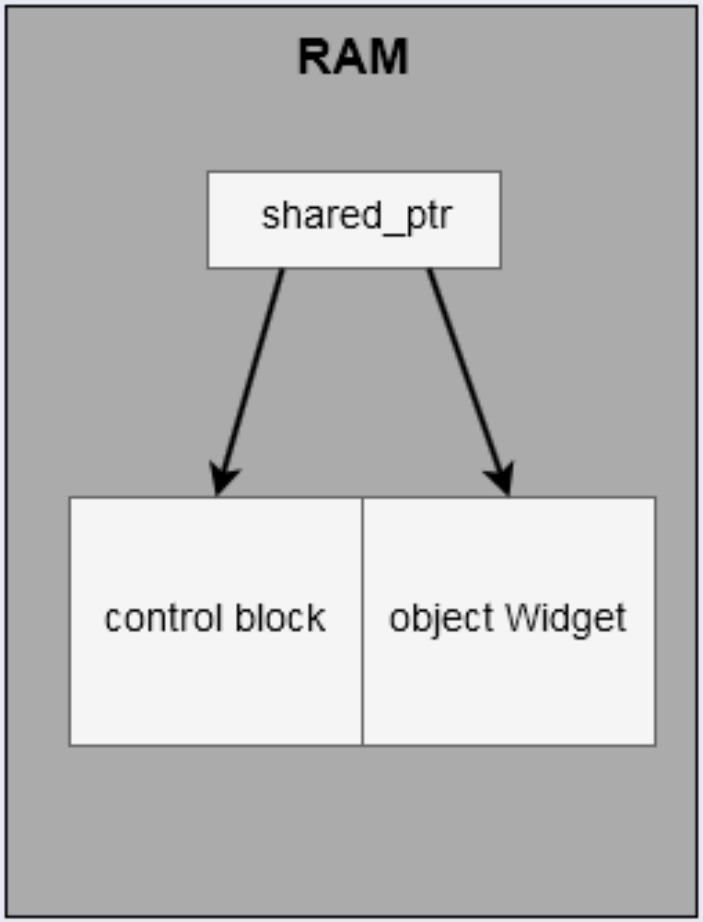
```
1 void someFunc(std::shared_ptr<Widget> ptr, int command);
2
3 someFunc(std::make_shared<Widget>(),
4     readCommand());
```

```
1 auto ptr = new Widget();
2 std::shared_ptr<Widget> sp(ptr);
```



- Выделяется память для Widget
- Выделяется память для управляющего блока
- Происходит два выделения памяти, что является не оптимальным решением

```
1 auto sp = std::make_shared<Widget>();
```



- Память выделяется один раз
- Управляющий блок и целевой объект создаются в едином участке памяти
- Удаление управляющего блока и целевого объекта происходит одновременно

В общем случае, используй make-функции

- make-функции устраняют дублирование кода, повышают безопасность кода по отношению к исключениям и в случае функций std::make_shared генерируют меньший по размеру и более быстрый код;
- Ситуации, когда применение make-функций неприемлемо, включают необходимость указания пользовательских удалителей и необходимость передачи инициализаторов в фигурных скобках;
- Для указателей std::shared_ptr дополнительными ситуациями, в которых применение make-функций может быть неблагоразумным, являются классы с пользовательским управлением памятью и системы, в которых проблемы с объемом памяти накладываются на использование очень больших объектов и наличие указателей std::weak_ptr, время жизни которых существенно превышает время жизни указателей std::shared_ptr.

Избегайте создания указателей из обычный встроенных указателей

В общем случае функция, создающая указатель std::shared_ptr на некоторый объект, не может знать, не указывает ли на этот объект некоторый другой указатель std::shared_ptr. Создание более одного std::shared_ptr из единственного обычного указателя приведет к **неопределенному поведению**.

Пример

```
1 auto ptr = new Widget;
2
3 std::shared_ptr<Widget> spw1(pw);
4
5 std::shared_ptr<Widget> spw2(pw);
```

Объясните код.

Избегайте создания указателей из обычный встроенных указателей

В общем случае функция, создающая указатель std::shared_ptr на некоторый объект, не может знать, не указывает ли на этот объект некоторый другой указатель std::shared_ptr. Создание более одного std::shared_ptr из единственного обычного указателя приведет к **неопределенному поведению**.

Плохой код - не делайте так!

```
1 auto ptr = new Widget;
2
3 std::shared_ptr<Widget> spw1(pw);
4
5 std::shared_ptr<Widget> spw2(pw);
```

Еще пример

```
1  struct Bad {
2      std::shared_ptr<Bad> getptr() {
3          return std::shared_ptr<Bad>(this);
4      }
5      ~Bad() { std::cout << "Bad::~Bad() called\n"; }
6  };
7
8  int main() {
9      // Bad, each shared_ptr thinks it's the only owner of the object
10     std::shared_ptr<Bad> bp1(new Bad);
11     std::shared_ptr<Bad> bp2 = bp1->getptr();
12     std::cout << "bp2.use_count() = " << bp2.use_count() << '\n';
13 } // UB: double-delete of Bad
```

Конструируемый таким образом указатель std::shared_ptr будет создавать новый управляющий блок для объекта **this*. Соответственно в данном примере присутствует та же самя проблема, что и в предыдущем.

shared_from_this

Шаблон std::enable_shared_from_this определяет функцию-член, которая создает std::shared_ptr для текущего объекта, но делает это, не дублируя управляющие блоки.

Исправленный пример

```
1  struct Good: std::enable_shared_from_this<Good> {
2      std::shared_ptr<Good> getptr() {
3          return shared_from_this();
4      }
5  };
6  int main() {
7      // Good: the two shared_ptr's share the same object
8      std::shared_ptr<Good> gp1(new Good);
9      std::shared_ptr<Good> gp2 = gp1->getptr();
10     std::cout << "gp2.use_count() = " << gp2.use_count() << '\n';
11 }
```

Как работает std::enable_shared_from_this::shared_from_this

Внутри себя shared_from_this ищет управляющий блок текущего объекта и создает новый std::shared_ptr, который использует этот управляющий блок.

- Класс enable_shared_from_this<T> имеет член weak_ptr<T> weakPtr.
- В свою очередь, конструктор shared_ptr<T> может определить является ли тип T наследником от enable_shared_from_this<T>. (а вот и type_traits пригодились)
- Если это так, то конструктор shared_ptr<T> свяжет weakPtr с объектом *this (используя friend-классы).
- После этого shared_from_this() может беззаветно создавать shared_ptr<T> из weakPtr.

Резюме

- – Требуется динамически выделенная память для хранения управляющего блока
- – Операции, требующих работы со счетчиком ссылок, из-за своей атомарности дороги
- – Применение указателей std::shared_ptr берет на себя также стоимость механизма виртуальной функции, используемой управляющим блоком
- \+ Разыменование std::shared_ptr не более дорогостояще, чем разыменование обычного указателя
- \+ Механизм виртуальных функций в управляющем блоке обычно используется только однажды для каждого объекта, когда происходит уничтожение объекта
- \+ Операции, требующих работы со счетчиком ссылок, потокобезопасные
- \+ Автоматическое управление временем жизни динамически выделяемых ресурсов с совместным владением

Резюме

Если вам достаточно или даже может быть достаточно исключительного владения, лучшим выбором является std::unique_ptr.

Его производительность близка к производительности для обычных указателей, а преобразование в std::shared_ptr выполняется очень легко.

В большинстве случаев *применение std::shared_ptr* значительно **предпочтительнее**, чем *ручное управление* временем жизни объекта с **совместным владением**.

Бонус

Пример

```
1  struct X{
2      Y y;
3  };
4
5  struct do_nothing_deleter{
6      template<typename> void operator()(T*){}
7  };
8
9  void store_for_later(std::shared_ptr<Y>);
10
11 void foo(){
12     std::shared_ptr<X> px(std::make_shared<X>());
13     std::shared_ptr<Y> py(&px->y,do_nothing_deleter());
14     store_for_later(py);
15 } // our X object is destroyed
```

Секретный конструктор shared_ptr: конструктор псевдонима

Конструктор псевдонима позволяет создавать std::shared_ptr на объект, который является частью другого объекта, при этом конструктор псевдонима обеспечивает сохранность родительского объекта.

```
template<typename Other, typename Target>
shared_ptr(shared_ptr<Other> const& other, Target* p);
```

```
void bar(){
    auto px = std::make_shared<X>();
    std::shared_ptr<Y> py(px, &px->y);
    store_for_later(py);
} // our X object is kept alive
```

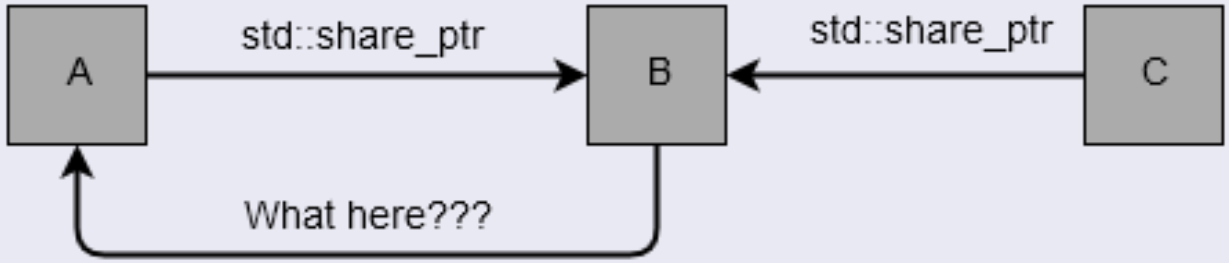
Данный метод может быть полезным в классах, которые используют идиому указателя на реализацию. Также конструктор псевдонима можно применять при необходимости передавать указатели на узлы дерева, при этом требуется само дерево.

Указатели shared_ptr обеспечивают автоматическое освобождение ресурсов, связанных с объектами, которые более не нужны. Однако при определенных обстоятельствах такое поведение является невозможным или нежелательным

- Если два объекста ссылаются друг на друга с помощью shared_ptr и вы хотите освобобить объекты и связзанные с ними ресурсы при условии, что на них больше никто не ссылается, указатель shared_ptr не освободит данные, потому что счетчик ссылок будет равен 1. В этой ситуации можно использовать обычные указатели, но тогда придется взять на себя управление ресурсами
- Другой пример - ситуация, в которой вы хотите раздельно использовать объект, но не владеть им. Таким образом возникает ситуация, в которой время жизни ссылки на объект превышает время жизни самого объекта.

Для таких ситуациях нам на помощь приходит класс **std::weak_ptr**

Поясняющий пример



- Обычный указатель. При таком подходе, если уничтожается A, а c продолжает указывать на B, B будет содержать указатель на A, который становится висящим. B не в состоянии этого определить, а потому B может непреднамеренно этот указатель разыменовать. В результате получается неопределенное поведение.
- Указатель std::shared_ptr. В этом случае A и B содержат указатели std::shared_ptr один на другой. Получающийся цикл std::shared_ptr предохраняет и A, и B от уничтожения.
- Указатель std::weak_ptr. Это позволяет избежать обеих описанных выше проблем. Если уничтожается A, указатель в в становится висящим, но B в состоянии это обнаружить.

std::weak_ptr

Класс std::weak_ptr требует создания совместно используемого указателя. Как только последний совместно используемый указатель, владеющий объектом, потеряет владение, слабый указатель автоматически станет пустым.

Помимо конструктора по умолчанию и копирующего конструктора, класс std::weak_ptr содержит только конструктор, получающий аргумент типа std::shared_ptr.

Нельзя использовать операторы * и -> для доступа к объекты, на который ссылается указатель std::weak_ptr.

Чтобы получить доступ, следует создать соответствующий std::shared_ptr.

weak_ptr to shared_ptr

Существует два способа получить std::shared_ptr из std::weak_ptr

- Использовать функцию std::weak_ptr::lock, которая возвращает std::shared_ptr в случае если std::weak_ptr не просрочен, иначе nullptr (в некоторых компиляторах будет сгенерировано исключение)
- Использовать конструктор std::shared_ptr, который принимает в качестве параметра std::weak_ptr. Если std::weak_ptr просрочен, генерируется исключение

weak_ptr::expired()

Если std::weak_ptr ссылается на уничтоженный объект (счетчик ссылок у shared_ptr равен 0), такой указатель называют висячим или просроченным

Чтобы узнать является ли висячим объект std::weak_ptr, можно использовать функцию std::weak_ptr::expired().

```
1 using namespace std;
2 auto spw = make_shared<Widget>();
3 weak_ptr<Widget> wpw(spw);
4 cout << boolalpha << wpw.expired() << endl; // cout: false
5 cout << boolalpha << wpw.use_count() == 0 << endl; // cout: false
6 spw.reset();
7 cout << boolalpha << wpw.expired() << endl; // cout: true
8 cout << boolalpha << wpw.use_count() == 0 << endl; // cout: true
```

Использование std::weak_ptr::expired() предпочтительнее, чем сравнение std::weak_ptr::use_count() на ноль

The end

Самостоятельное изучение

- boost::scoped_ptr
- boost::intrusive_ptr
- nothrow

Список литературы

- Мейерс. Эффективный и современный C++. 42 рекомендации по использованию C++ 11 и C++ 14
- Джосаттис. Стандартная библиотека C++. Справочное руководство
- http://archive.kalnitsky.org/2011/11/02/smart-pointers-in-cpp11/
- https://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/