Лабораторная работа

Умные указатели и семантика перемещения

1. Умные указатели и семантика перемещения
   1. Зачем нужны умные указатели

Рассмотрим функцию, в которой динамически выделяется переменная:

**void myFunction()**

**{**

**Item \*ptr = new Item; // Item-ом может быть структура или класс**

**delete ptr;**

**}**

Хотя код, приведенный выше, кажется довольно простым, можно очень легко забыть в конце освободить память, выделенную ptr. Даже если вы не забудете это сделать, существует множество причин, по которым ptr не будет удален. Это может произойти из-за досрочного возврата return или из-за генерации исключения:

**void myFunction()**

**{**

**Item \*ptr = new Item;**

**int a;**

**std::cout << "Enter an integer: ";**

**std::cin >> a;**

**if (a == 0)**

**throw 0; // ptr не удаляется!**

**delete ptr;**

**}**

Одна из лучших особенностей классов — это деструкторы, которые автоматически выполняются при выходе объекта класса из области видимости. При выделении памяти в конструкторе класса, вы можете быть уверены, что эта память будет освобождена в деструкторе при уничтожении объекта класса (независимо от того, выйдет ли он из области видимости, будет ли явно удален и т.д.).

* 1. Пример умного указателя

Например, рассмотрим класс, единственными задачами которого является хранение и «управление» переданным ему указателем, а затем корректное освобождение памяти при выходе объекта класса из области видимости. До того момента, пока объекты этого класса создаются как локальные переменные, мы можем гарантировать, что, как только они выйдут из области видимости (независимо от того, когда или как), переданный указатель будет уничтожен.

**#include <iostream>**

**template<class T>**

**class Auto\_ptr1**

**{**

**T\* m\_ptr;**

**public:**

**// Получаем указатель через конструктор**

**Auto\_ptr1(T\* ptr=nullptr)**

**:m\_ptr(ptr)**

**{**

**}**

**// Деструктор позаботится об удалении указателя**

**~Auto\_ptr1()**

**{**

**delete m\_ptr;**

**}**

**// Выполняем перегрузку оператора разыменования и оператора ->, чтобы иметь возможность использовать Auto\_ptr1 как m\_ptr**

**T& operator\*() const { return \*m\_ptr; }**

**T\* operator->() const { return m\_ptr; }**

**};**

**// Класс для проверки работоспособности вышеприведенного кода**

**class Item**

**{**

**public:**

**Item() { std::cout << "Item acquired\n"; }**

**~Item() { std::cout << "Item destroyed\n"; }**

**};**

**int main()**

**{**

**Auto\_ptr1<Item> item(new Item); // динамическое выделение памяти**

**// Также обратите внимание на то, что Item-у в угловых скобках не требуется символ \*, поскольку это предоставляется шаблоном класса**

**return 0;**

**} // item выходит из области видимости здесь и уничтожает выделенный Item вместо нас**

Сначала мы динамически выделяем объект класса Item и передаем его в качестве параметра нашему шаблону класса Auto\_ptr1. С этого момента объект item класса Auto\_ptr1 владеет выделенным объектом класса Item (Auto\_ptr1 имеет композиционную связь с m\_ptr). Поскольку item объявлен в качестве локальной переменной и имеет область видимости блока, он выйдет из области видимости после завершения выполнения блока, в котором находится, и будет уничтожен. А поскольку это объект класса, то при его уничтожении будет вызван деструктор Auto\_ptr1. Этот деструктор и обеспечит удаление указателя Item, который он хранит.

До тех пор, пока объект класса Auto\_ptr1 определен как локальная переменная, Item гарантированно будет уничтожен в конце блока, в котором он объявлен, независимо от того, как этот блок завершит свое выполнение (досрочно или нет).

* 1. Критический недостаток

Класс Auto\_ptr1, приведенный выше, имеет критическую ошибку.

**int main()**

**{**

**Auto\_ptr1<Item> item1(new Item);**

**Auto\_ptr1<Item> item2(item1); // в качестве альтернативы вы можете не инициализировать item2 значением item1, а просто выполнить присваивание item2 = item1**

**return 0;**

**}**

Поскольку мы не предоставили конструктор копирования или свой оператор присваивания (перегрузку оператора присваивания), то язык C++ предоставил их самостоятельно. И то, что он предоставил, выполняет поверхностное копирование. Поэтому, когда мы инициализируем item2 значением item1, оба объекта класса Auto\_ptr1 указывают на один и тот же Item. Когда item2 выходит из области видимости, он удаляет Item, оставляя item1 с висячим указателем.

* 1. Семантика перемещения

А что, если бы наш конструктор копирования и оператор присваивания не копировали указатель (семантика копирования), а передавали владение указателем из источника в объект назначения? Это основная идея семантики перемещения. Семантика перемещения означает, что класс, вместо копирования, передает право собственности на объект.

**#include <iostream>**

**template<class T>**

**class Auto\_ptr2**

**{**

**T\* m\_ptr;**

**public:**

**Auto\_ptr2(T\* ptr=nullptr)**

**:m\_ptr(ptr)**

**{**

**}**

**~Auto\_ptr2()**

**{**

**delete m\_ptr;**

**}**

**// Конструктор копирования, который реализовывает семантику перемещения**

**Auto\_ptr2(Auto\_ptr2& a) // примечание: Ссылка не является константной**

**{**

**m\_ptr = a.m\_ptr; // перемещаем наш глупый указатель от источника к нашему локальному объекту**

**a.m\_ptr = nullptr; // подтверждаем, что источник больше не владеет указателем**

**}**

**// Оператор присваивания, который реализовывает семантику перемещения**

**Auto\_ptr2& operator=(Auto\_ptr2& a) // примечание: Ссылка не является константной**

**{**

**if (&a == this)**

**return \*this;**

**delete m\_ptr; // подтверждаем, что удалили любой указатель, который наш локальный объект имел до этого**

**m\_ptr = a.m\_ptr; // затем перемещаем наш глупый указатель из источника к нашему локальному объекту**

**a.m\_ptr = nullptr; // подтверждаем, что источник больше не владеет указателем**

**return \*this;**

**}**

**T& operator\*() const { return \*m\_ptr; }**

**T\* operator->() const { return m\_ptr; }**

**bool isNull() const { return m\_ptr == nullptr; }**

**};**

**class Item**

**{**

**public:**

**Item() { std::cout << "Item acquired\n"; }**

**~Item() { std::cout << "Item destroyed\n"; }**

**};**

**int main()**

**{**

**Auto\_ptr2<Item> item1(new Item);**

**Auto\_ptr2<Item> item2; // начнем с nullptr**

**std::cout << "item1 is " << (item1.isNull() ? "null\n" : "not null\n");**

**std::cout << "item2 is " << (item2.isNull() ? "null\n" : "not null\n");**

**item2 = item1; // item2 теперь является "владельцем" значения item1, объекту item1 присваивается null**

**std::cout << "Ownership transferred\n";**

**std::cout << "item1 is " << (item1.isNull() ? "null\n" : "not null\n");**

**std::cout << "item2 is " << (item2.isNull() ? "null\n" : "not null\n");**

**return 0;**

**}**

Обратите внимание, перегруженный operator= передает право собственности на m\_ptr от item1 к item2. Следовательно, у нас не выполняется дублирования указателей, и всё аккуратно очищается (удаляется).

* 1. auto\_ptr и почему его лучше не использовать

std::auto\_ptr, представленный в C++98, был первой попыткой в языке C++ сделать стандартизированный умный указатель. В std::auto\_ptr решили реализовать семантику перемещения точно так же, как это сделано в классе Auto\_ptr2.

Однако, std::auto\_ptr (как и наш класс Auto\_ptr2) имеет ряд проблем, которые делают его использование опасным.

Во-первых, поскольку std::auto\_ptr реализовывает семантику перемещения через конструктор копирования и оператор присваивания, то передача std::auto\_ptr в функцию по значению приведет к тому, что ваш Item будет перемещен в параметр функции и, следовательно, будет уничтожен в конце функции, когда параметры этой функции выйдут из области видимости (в нашем классе Auto\_ptr2 передача выполняется по ссылке). Затем, когда вы попытаетесь получить доступ к аргументу std::auto\_ptr из caller-а (не осознавая, что он был передан и удален), вы внезапно выполните разыменование нулевого указателя.

Во-вторых, std::auto\_ptr всегда удаляет свое содержимое, используя оператор delete, который не работает с массивами. Это означает, что std::auto\_ptr не будет правильно работать с динамическими массивами, поскольку использует неправильный тип удаления. Хуже того, std::auto\_ptr не помешает вам передать ему динамический массив, который затем будет неправильно обработан, что приведет к утечке памяти.

Наконец, std::auto\_ptr не очень хорошо работает со многими другими классами из Стандартной библиотеки С++ (особенно с контейнерными классами и классами алгоритмов). Это происходит из-за того, что классы Стандартной библиотеки С++ предполагают, что, когда они копируют элемент, они фактически выполняют копирование, а не перемещение.

1. Умный указатель unique\_ptr
   1. unique\_ptr

Умный указатель std::unique\_ptr является заменой std::auto\_ptr в C++11. Вы должны использовать именно его для управления любым динамически выделенным объектом/ресурсом, но с условием, что std::unique\_ptr полностью владеет переданным ему объектом, а не делится «владением» еще с другими классами. Умный указатель std::unique\_ptr находится в заголовочном файле memory.

Рассмотрим простой пример использования std::unique\_ptr:

**#include <iostream>**

**#include <memory> // для std::unique\_ptr**

**class Item**

**{**

**public:**

**Item() { std::cout << "Item acquired\n"; }**

**~Item() { std::cout << "Item destroyed\n"; }**

**};**

**int main()**

**{**

**// Выделяем объект класса Item и передаем право собственности на него std::unique\_ptr**

**std::unique\_ptr<Item> item(new Item);**

**return 0;**

**} // item выходит из области видимости здесь, соответственно, Item уничтожается также здесь**

Когда std::unique\_ptr выходит из области видимости, он удаляет Item, которым владеет.

В отличие от std::auto\_ptr, std::unique\_ptr корректно реализовывает семантику перемещения.

Поскольку std::unique\_ptr разработан с учетом семантики перемещения, то семантика копирования по умолчанию отключена. Если вы хотите передать содержимое, управляемое std::unique\_ptr, то вы должны использовать семантику перемещения. В программе, приведенной выше, мы передаем содержимое std::unique\_ptr с помощью функции std::move() (которая конвертирует item1 в r-value, являющееся триггером для выполнения семантики перемещения вместо семантики копирования).

* 1. Доступ к объекту, который хранит умный указатель

Умный указатель std::unique\_ptr имеет перегруженные операторы \* и ->, которые используются для доступа к хранимым объектам. Оператор \* возвращает ссылку на управляемый ресурс, а оператор -> возвращает указатель.

Умный указатель std::unique\_ptr не всегда может управлять объектом: либо потому, что объект был создан пустым (с использованием конструктора по умолчанию, или в объект передан в качестве параметра nullptr), либо потому, что ресурс, которым он управлял, был перемещен в другой std::unique\_ptr. Поэтому, прежде чем использовать какой-либо из этих операторов, вы должны проверить, действительно ли std::unique\_ptr управляет ресурсом. К счастью, это легко сделать: std::unique\_ptr имеет неявное преобразование в тип bool, возвращая true, если std::unique\_ptr владеет ресурсом. Например:

**#include <iostream>**

**#include <memory> // для std::unique\_ptr**

**class Item**

**{**

**public:**

**Item() { std::cout << "Item acquired\n"; }**

**~Item() { std::cout << "Item destroyed\n"; }**

**friend std::ostream& operator<<(std::ostream& out, const Item &item)**

**{**

**out << "I am an Item!\n";**

**return out;**

**}**

**};**

**int main()**

**{**

**std::unique\_ptr<Item> item(new Item);**

**if (item) // используем неявное преобразование item в тип bool, чтобы убедиться, что item владеет Item-ом**

**std::cout << \*item; // выводим Item, которым владеет item**

**return 0;**

**}**

* 1. unique\_ptr и динамические массивы

В отличие от std::auto\_ptr, std::unique\_ptr достаточно умен, чтобы знать, когда использовать единичный оператор delete, а когда форму оператора delete для массива, поэтому std::unique\_ptr можно использовать как с единичными объектами, так и с динамическими массивами.

Однако использование std::vector почти всегда является лучшим выбором, чем использование std::unique\_ptr с динамическим массивом.

* 1. make\_unique()

В C++14 добавили новую функцию — std::make\_unique(). Это шаблон функции, который создает объект типа шаблона и инициализирует его аргументами, переданными в функцию.

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

* 1. Возврат unique\_ptr из функции

Умный указатель std::unique\_ptr можно возвращать из функции по значению:

**std::unique\_ptr<Item> createItem()**

**{**

**return std::make\_unique<Item>();**

**}**

**int main()**

**{**

**std::unique\_ptr<Item> ptr = createItem();**

**// Делаем что-либо**

**return 0;**

**}**

* 1. Передача unique\_ptr в функцию

Если вы хотите, чтобы функция стала владельцем содержимого умного указателя, то передавать std::unique\_ptr в функцию нужно по значению. Обратите внимание, поскольку семантика копирования отключена, то вам придется использовать std::move() для фактической передачи ресурса в функцию:

**#include <iostream>**

**#include <memory> // для std::unique\_ptr**

**class Item**

**{**

**public:**

**Item() { std::cout << "Item acquired\n"; }**

**~Item() { std::cout << "Item destroyed\n"; }**

**friend std::ostream& operator<<(std::ostream& out, const Item &item)**

**{**

**out << "I am an Item!\n";**

**return out;**

**}**

**};**

**void takeOwnership(std::unique\_ptr<Item> item)**

**{**

**if (item)**

**std::cout << \*item;**

**} // Item уничтожается здесь**

**int main()**

**{**

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

**// takeOwnership(ptr); // это не скомпилируется. Мы должны использовать семантику перемещения**

**takeOwnership(std::move(ptr)); // используем семантику перемещения**

**std::cout << "Ending program\n";**

**return 0;**

**}**

Обратите внимание, в данном случае право собственности на Item было передано в takeOwnership(), поэтому Item уничтожается в конце takeOwnership(), а не в конце main().

Однако в большинстве случаев вам не нужно будет, чтобы функция владела ресурсом. Хотя вы можете передать std::unique\_ptr по ссылке (что позволит функции использовать объект без передачи ей права собственности на этот объект), вы должны делать это только тогда, когда caller может изменить передаваемый объект.

Вместо этого лучше передавать сам объект по указателю или по ссылке (в зависимости от того, является ли null допустимым аргументом). Это позволит функции оставаться в стороне от управления объектом. Чтобы получить необработанный указатель на объект из std::unique\_ptr, вы можете использовать метод get()

* 1. unique\_ptr и классы

Вы можете использовать std::unique\_ptr в качестве члена композиции вашего класса. Таким образом, вам не нужно будет беспокоиться о том, удалит ли деструктор вашего класса ресурс std::unique\_ptr, так как std::unique\_ptr будет автоматически уничтожен при уничтожении объекта класса. Тем не менее, если объект вашего класса выделяется динамически, то сам ресурс std::unique\_ptr подвергается риску неправильного удаления, и в таком случае даже умный указатель не поможет.

1. Умный указатель std::shared\_ptr
   1. shared\_ptr

Вы можете иметь несколько умных указателей std::shared\_ptr, указывающих на один и тот же ресурс. Умный указатель std::shared\_ptr отслеживает количество владельцев у каждого полученного ресурса. До тех пор, пока хотя бы один std::shared\_ptr владеет ресурсом, этот ресурс не будет уничтожен, даже если удалить все остальные std::shared\_ptr (которые также владеют этим ресурсом). Как только последний std::shared\_ptr, владеющий ресурсом, выйдет из области видимости (или ему передадут другой ресурс для управления), тогда ресурс будет уничтожен.

Как и std::unique\_ptr, std::shared\_ptr находится в заголовочном файле memory.

**#include <iostream>**

**#include <memory> // для std::shared\_ptr**

**class Item**

**{**

**public:**

**Item() { std::cout << "Item acquired\n"; }**

**~Item() { std::cout << "Item destroyed\n"; }**

**};**

**int main()**

**{**

**// Выделяем Item и передаем его в std::shared\_ptr**

**Item \*item = new Item;**

**std::shared\_ptr<Item> ptr1(item);**

**{**

**std::shared\_ptr<Item> ptr2(ptr1); // используем копирующую инициализацию для создания второго std::shared\_ptr из ptr1, указывающего на тот же Item**

**std::cout << "Killing one shared pointer\n";**

**} // ptr2 выходит из области видимости здесь, но больше ничего не происходит**

**std::cout << "Killing another shared pointer\n";**

**return 0;**

**} // ptr1 выходит из области видимости здесь, и выделенный Item уничтожается также здесь**

* 1. make\_shared()

Как функцию std::make\_unique() можно использовать для создания std::unique\_ptr в C++14, так и функцию std::make\_shared() можно (и нужно) использовать для создания std::shared\_ptr. Функцию std::make\_shared() добавили в C++11.

* 1. Детали реализации

В отличие от std::unique\_ptr, который использует внутри один указатель, std::shared\_ptr использует внутри два указателя. Один указывает на передаваемый ресурс, а второй — на «блок управления» — динамически выделенный объект, который отслеживает кучу разных вещей, включая и то, сколько умных указателей std::shared\_ptr одновременно указывают на каждый полученный ресурс. При создании std::shared\_ptr с помощью конструктора std::shared\_ptr, память для полученного ресурса и блока управления (который также создает конструктор) выделяется отдельно. Однако в std::make\_shared() это оптимизировано в единое выделение памяти, что, соответственно, повышает производительность.

Это также объясняет то, почему независимое создание двух std::shared\_ptr, указывающих на один и тот же ресурс, приводит к проблемам. Каждый std::shared\_ptr имеет один указатель, указывающий на полученный ресурс. Однако каждый std::shared\_ptr еще и независимо выделяет свой собственный блок управления, который сообщает указателю, что он является единственным «владельцем» полученного ресурса (даже если это не так). Таким образом, когда данный std::shared\_ptr выходит из области видимости, он уничтожает ресурс, которым владеет, не осознавая того, что могут быть еще другие умные указатели std::shared\_ptr, которые владеют этим же ресурсом.

Однако, когда std::shared\_ptr клонируется с использованием семантики копирования, данные в блоке управления соответствующим образом обновляются и говорят о том, что появился еще один std::shared\_ptr, указывающий на полученный ресурс.

* 1. Создание shared\_ptr из unique\_ptr

Умный указатель std::unique\_ptr может быть конвертирован в умный указатель std::shared\_ptr через специальный конструктор std::shared\_ptr, который принимает std::unique\_ptr в качестве r-value. Таким образом, содержимое std::unique\_ptr перемещается в std::shared\_ptr.

Однако std::shared\_ptr нельзя безопасно конвертировать в std::unique\_ptr. Поэтому, если вы хотите создать функцию, которая будет возвращать умный указатель, вам лучше возвращать std::unique\_ptr и затем присваивать его std::shared\_ptr, когда это будет уместно.

* 1. shared\_ptr и массивы

В C++14 и в более ранних версиях С++ std::shared\_ptr не имеет поддержки управления динамическими массивами и, соответственно, не должен использоваться с ними. Начиная с C++17, в std::shared\_ptr добавили поддержку динамических массивов. Однако в C++17 «забыли» добавить эту поддержку в std::make\_shared(), поэтому данную функцию не следует использовать для создания std::shared\_ptr, указывающего на динамические массивы.

1. Умный указатель std::weak\_ptr
   1. Пересечение умных указателей

Например, рассмотрим случай, когда два умных указателя типа std::shared\_ptr владеют двумя разными объектами и «пересекаются» между собой:

**#include <iostream>**

**#include <memory> // для std::shared\_ptr**

**#include <string>**

**class Human**

**{**

**std::string m\_name;**

**std::shared\_ptr<Human> m\_partner; // изначально пустой**

**public:**

**Human(const std::string &name): m\_name(name)**

**{**

**std::cout << m\_name << " created\n";**

**}**

**~Human()**

**{**

**std::cout << m\_name << " destroyed\n";**

**}**

**friend bool partnerUp(std::shared\_ptr<Human> &h1, std::shared\_ptr<Human> &h2)**

**{**

**if (!h1 || !h2)**

**return false;**

**h1->m\_partner = h2;**

**h2->m\_partner = h1;**

**std::cout << h1->m\_name << " is now partnered with " << h2->m\_name << "\n";**

**return true;**

**}**

**};**

**int main()**

**{**

**auto anton = std::make\_shared<Human>("Anton"); // создание умного указателя с объектом Anton класса Human**

**auto ivan = std::make\_shared<Human>("Ivan"); // создание умного указателя с объектом Ivan класса Human**

**partnerUp(anton, ivan); // Anton указывает на Ivan-а, а Ivan указывает на Anton-а**

**return 0;**

**}**

Здесь мы динамически выделяем два объекта (Anton и Ivan) класса Human и, используя std::make\_shared, передаем их в два отдельно созданных умных указателя типа std::shared\_ptr. Затем «связываем» их с помощью дружественной функции partnerUp().

После вызова функции partnerUp() у нас образуется 4 умных указателя типа std::shared\_ptr:

* Два умных указателя указывают на объект Ivan: ivan (из функции main()) и m\_partner (из класса Human) объекта Anton.
* Два умных указателя указывают на объект Anton: anton и m\_partner объекта Ivan.

В конце функции partnerUp() умный указатель ivan выходит из области видимости первым. Когда это происходит, он проверяет, есть ли другие умные указатели, которые владеют объектом Ivan класса Human. Есть — m\_partner объекта Anton. Из-за этого умный указатель не уничтожает Ivan-а (если он это сделает, то m\_partner объекта Anton останется висячим указателем). Таким образом у нас остается один умный указатель, владеющий Ivan-ом (m\_partner объекта Anton) и два умных указателя, владеющие Anton-ом (anton и m\_partner объекта Ivan).

Затем умный указатель anton выходит из области видимости, и происходит то же самое. anton проверяет, есть ли другие умные указатели, которые также владеют объектом Anton класса Human. Есть — m\_partner объекта Ivan, поэтому объект Anton не уничтожается. Таким образом, остаются два умных указателя:

* m\_partner объекта Ivan, который указывает на Anton-а;
* m\_partner объекта Anton, который указывает на Ivan-а.

Затем программа завершает свое выполнение, и ни объект Anton, ни объект Ivan не уничтожаются. По сути, Anton не дает уничтожить Ivan-а, а Ivan не дает уничтожить Anton-а.

Это тот случай, когда умные указатели типа std::shared\_ptr формируют циклическую зависимость.

* 1. Циклическая зависимость

Циклическая зависимость (или «циклические ссылки») — это серия «ссылок», где текущий объект ссылается на следующий, а последний объект ссылается на первый. Эти «ссылки» не обязательно должны быть обычными ссылками в языке C++, они могут быть указателями, уникальными ID или любыми другими средствами идентификации конкретных объектов.

В контексте std::shared\_ptr этими «ссылками» являются указатели.

Это именно то, что мы видим выше: Anton указывает на Ivan-а, а Ivan указывает на Anton-а. Аналогично, A указывает на B, B указывает на C, а C указывает на A. Практическая ценность такой циклической зависимости в том, что текущий объект не дает уничтожить следующий объект.

* 1. weak\_ptr

Умный указатель std::weak\_ptr был разработан для решения проблемы «циклической зависимости», описанной выше. std::weak\_ptr является наблюдателем — он может наблюдать и получать доступ к тому же объекту, на который указывает std::shared\_ptr (или другой std::weak\_ptr), но не считаться владельцем этого объекта. Помните, когда std::shared\_ptr выходит из области видимости, он проверяет, есть ли другие владельцы std::shared\_ptr. std::weak\_ptr владельцем не считается.

Давайте перепишем предыдущую программу, но уже с использованием std::weak\_ptr:

**#include <iostream>**

**#include <memory> // для std::shared\_ptr и std::weak\_ptr**

**#include <string>**

**class Human**

**{**

**std::string m\_name;**

**std::weak\_ptr<Human> m\_partner; // обратите внимание, здесь std::weak\_ptr**

**public:**

**Human(const std::string &name): m\_name(name)**

**{**

**std::cout << m\_name << " created\n";**

**}**

**~Human()**

**{**

**std::cout << m\_name << " destroyed\n";**

**}**

**friend bool partnerUp(std::shared\_ptr<Human> &h1, std::shared\_ptr<Human> &h2)**

**{**

**if (!h1 || !h2)**

**return false;**

**h1->m\_partner = h2;**

**h2->m\_partner = h1;**

**std::cout << h1->m\_name << " is now partnered with " << h2->m\_name << "\n";**

**return true;**

**}**

**};**

**int main()**

**{**

**auto anton = std::make\_shared<Human>("Anton");**

**auto ivan = std::make\_shared<Human>("Ivan");**

**partnerUp(anton, ivan);**

**return 0;**

**}**

* 1. Использование

Недостатком умного указателя std::weak\_ptr является то, что его нельзя использовать напрямую (нет оператора ->). Чтобы использовать std::weak\_ptr, вы сначала должны конвертировать его в std::shared\_ptr (с помощью метода lock()), а затем уже использовать std::shared\_ptr.

**#include <iostream>**

**#include <memory> // для std::shared\_ptr и std::weak\_ptr**

**#include <string>**

**class Human**

**{**

**std::string m\_name;**

**std::weak\_ptr<Human> m\_partner; // обратите внимание, здесь std::weak\_ptr**

**public:**

**Human(const std::string &name) : m\_name(name)**

**{**

**std::cout << m\_name << " created\n";**

**}**

**~Human()**

**{**

**std::cout << m\_name << " destroyed\n";**

**}**

**friend bool partnerUp(std::shared\_ptr<Human> &h1, std::shared\_ptr<Human> &h2)**

**{**

**if (!h1 || !h2)**

**return false;**

**h1->m\_partner = h2;**

**h2->m\_partner = h1;**

**std::cout << h1->m\_name << " is now partnered with " << h2->m\_name << "\n";**

**return true;**

**}**

**const std::shared\_ptr<Human> getPartner() const { return m\_partner.lock(); } // используем метод lock() для конвертации std::weak\_ptr в std::shared\_ptr**

**const std::string& getName() const { return m\_name; }**

**};**

**int main()**

**{**

**auto anton = std::make\_shared<Human>("Anton");**

**auto ivan = std::make\_shared<Human>("Ivan");**

**partnerUp(anton, ivan);**

**auto partner = ivan->getPartner(); // передаем partner-у содержимое умного указателя, которым владеет ivan**

**std::cout << ivan->getName() << "'s partner is: " << partner->getName() << '\n';**

**return 0;**

**}**

1. Задания

Задание 1

Выберите вариант в соответствии с вашим номером в списке группы. Для реализации связей между элементами списка/дерева выберите наиболее подходящий из умных указателей (unique\_ptr, shared\_ptr или weak\_ptr).

|  |  |
| --- | --- |
| 1, 8, 15, 22 | Разработать класс стека с операциями вставки и извлечения элемента (push и pop) |
| 2, 9, 16, 23 | Разработать класс очереди с операциями вставки и извлечения элемента (push\_back и pop\_front). |
| 3, 10, 17, 24 | Разработать класс двунаправленной очереди с операциями вставки и извлечения элемента из начала или конца (push\_front, push\_back, pop\_front, pop\_back). |
| 4, 11, 18, 25 | Разработать класс сортированного бинарного дерева. Реализовать операции вставки и удаления элемента. |
| 5, 12, 19, 26 | Разработать класс сортированного бинарного дерева. Реализовать операции вставки и поиска элемента. |
| 6, 13, 20, 27 | Разработать класс хэш-таблицы с закрытой адресацией. Реализовать операции вставки и удаления элемента. Используйте односвязный список для хранения элементов с одинаковым значением хэша. |
| 7, 14, 21, 28 | Разработать класс хэш-таблицы с закрытой адресацией. Реализовать операции вставки и поиска элемента. Используйте односвязный список для хранения элементов с одинаковым значением хэша. |

Задание 2

Согласно варианту реализовать класс, наполнить его полями и методами. Использовать в реализации умные указатели. Протестировать в случаях, где вероятна утечка памяти (досрочный выход с помощью return или генерации исключения, циклическая зависимость указателей).

|  |  |
| --- | --- |
| **Вариант** | **Задание** |
| 1 | Класс Студент |
| 2 | Класс Поезд |
| 3 | Класс Домашняя библиотека |
| 4 | Класс Покупатель |
| 5 | Класс Дата |
| 6 | Класс Работник |
| 7 | Класс Магазин |
| 8 | Класс Ракета-носитель |
| 9 | Класс Космический аппарат |