Уникальные указатели

Уникальный указатель имеет передаваемое исключительное право собственности на один динамический объект. Можно перемещать уникальные указатели, что делает их переносимыми. Они также имеют исключительную собственность, поэтому их нельзя скопировать. У stdlib есть `unique_ptr`, доступный в заголовке `<memory>`.

ПРИМЕЧАНИЕ

Boost не предоставляет уникальный указатель.

Создание

Функция `std::unique_ptr` принимает один параметр шаблона, соответствующий указанному типу, как в `std::unique_ptr<int>` для типа «уникальный указатель на `int`».

Как и в случае с ограниченным указателем, уникальный указатель имеет конструктор по умолчанию, который инициализирует уникальный указатель как пустой. Он также предоставляет конструктор, принимающий обычный указатель, который становится владельцем динамического объекта, на который указывает указатель. Один способ создания — создать динамический объект через `new` и передать результат в конструктор, например так:

```
std::unique_ptr<int> my_ptr{ new int{ 808 } };
```

Другой метод — использовать функцию `std::make_unique`. Функция `make_unique` — это шаблон, который принимает все аргументы и передает их соответствующему параметру конструктора шаблона. Это избавляет от необходимости использования `new`. Используя `std::make_unique`, можно переписать предыдущую инициализацию объекта следующим образом:

```
auto my_ptr = make_unique<int>(808);
```

Функция `make_unique` была создана, чтобы избежать некоторых утечек памяти, которые имели место, когда использовался `new` с предыдущими версиями C++. Однако в последней версии C++ эти утечки памяти больше не происходят. Использование конструктора в основном зависит от ваших предпочтений.

Поддерживаемые операции

Функция `std::unique_ptr` поддерживает все операции, которые поддерживает `boost::scoped_ptr`. Например, можно использовать следующий псевдоним типа в качестве замены для `ScopedOathbreaker` в листингах 11.1–11.7:

```
using UniqueOathbreakers = std::unique_ptr<DeadMenOfDunharrow>;
```

Одно из основных отличий между уникальными и ограниченными указателями состоит в том, что можно перемещать уникальные указатели, потому что они *переносимы*.

Переносимое и исключительное владение

Уникальные указатели не только могут быть переданы, но они имеют исключительное право собственности (их *нельзя* копировать). В листинге 11.10 показано, как можно использовать семантику переноса для `unique_ptr`.

Листинг 11.10. std::unique_ptr поддерживает семантику переноса для передачи владения

```
TEST_CASE("UniquePtr can be used in move") {
  auto aragorn = std::make_unique<DeadMenOfDunharrow>(); ❶
  SECTION("construction") {
    auto son_of_arathorn{ std::move(aragorn) }; ❷
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❸
  }
  SECTION("assignment") {
    auto son_of_arathorn = std::make_unique<DeadMenOfDunharrow>(); ❹
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2); ❺
    son_of_arathorn = std::move(aragorn); ❻
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❼
  }
}
```

Этот листинг создает `unique_ptr` с именем `aragorn` ❶, который используется в двух отдельных тестах.

В первом тесте `aragorn` перемещается с помощью `std::move` в конструктор переноса `son_of_arathorn` ❷. Поскольку `aragorn` передает право собственности на его `DeadMenOfDunharrow` в `son_of_arathorn`, объект `oaths_to_fulfill` все еще имеет значение 1 ❸.

Второй тест создает `son_of_arathorn` через `make_unique` ❹, который увеличивает значение `oaths_to_fulfill` до 2 ❺. Затем используется оператор присваивания переноса для перемещения `aragorn` в `son_of_arathorn` ❻. Опять же `aragorn` передает право собственности в `son_of_arathorn`. Поскольку `son_of_arathorn` может одновременно владеть только одним динамическим объектом, оператор присваивания переноса уничтожает принадлежащий в настоящее время объект перед очисткой динамического объекта `aragorn`. Это приводит к уменьшению `oaths_to_fulfill` до 1 ❼.

Уникальные массивы

В отличие от `boost::scoped_ptr`, `std::unique_ptr` имеет встроенную поддержку динамических массивов. Просто используется тип массива в качестве параметра шаблона в типе уникального указателя, как в `std::unique_ptr<int []>`.

Очень важно, чтобы `std::unique_ptr<T>` не был инициализирован динамическим массивом `T[]`. Это приведет к неопределенному поведению, потому что будет вызываться `delete` массива (а не `delete[]`). Компилятор не может уберечь от этого, потому что оператор `new[]` возвращает указатель, который неотличим от вида, возвращаемого оператором `new`.

Как и `scoped_array`, тип `unique_ptr` для массива предлагает `operator[]` для доступа к элементам. Листинг 11.11 демонстрирует эту концепцию.

Листинг 11.11. std::unique_ptr для типа массива поддерживает operator[]

```
TEST_CASE("UniquePtr to array supports operator[]") {
  std::unique_ptr<int[]❶> squares{
    new int[5]{ 1, 4, 9, 16, 25 } ❷
  };
  squares[0] = 1; ❸
  REQUIRE(squares[0] == 1); ❹
  REQUIRE(squares[1] == 4);
  REQUIRE(squares[2] == 9);
}
```

Параметр шаблона `int[]` ❶ указывает `std::unique_ptr`, что ему принадлежит динамический массив. Вы передаете вновь созданный динамический массив ❷, а затем используете `operator[]` для установки первого элемента ❸; затем используется `operator[]` для извлечения элементов ❹.

Удалители

У `std::unique_ptr` есть второй, необязательный, параметр шаблона, называемый его типом удалителя. *Удалитель* уникального указателя — это то, что вызывается, когда уникальный указатель должен уничтожить свой собственный объект.

Экземпляр `unique_ptr` содержит следующие параметры шаблона:

```
std::unique_ptr<T, Deleter=std::default_delete<T>>
```

Двумя параметрами шаблона являются `T`, тип принадлежащего динамического объекта, и `Deleter`, тип объекта, ответственного за освобождение принадлежащего объекта. По умолчанию `Deleter` — это `std::default_delete<T>`, который вызывает `delete` или `delete[]` для динамического объекта.

Чтобы написать пользовательский удалитель, все, что нужно, — это функциональный объект, который вызывается с помощью `T*`. (Уникальный указатель будет игнорировать возвращаемое значение удалителя.) Этот удалитель передается как второй параметр конструктору уникального указателя, что показано в листинге 11.12.

Тип принадлежащего объекта — `int` ❷, поэтому объявляется объект функции `my_deleter`, который принимает `int*` ❶. `decltype` используется, чтобы установить параметр шаблона удалителя ❸.

Листинг 11.12. Передача пользовательского удалителя в уникальный указатель

```
#include <cstdio>

auto my_deleter = [](int* x) { ❶
  printf("Deleting an int at %p.", x);
  delete x;
};
std::unique_ptr<int❷, decltype(my_deleter)❸> my_up{
  new int,
  my_deleter
};
```

Пользовательские удалители и системное программирование

Пользовательский удалитель используется, когда `delete` не обеспечивает требуемого поведения для освобождения ресурса. В некоторых разработках никогда не понадобится пользовательский удалитель. В других, таких как системное программирование, они могут оказаться весьма полезными. Рассмотрим простой пример, где происходит управление файлом с помощью низкоуровневых API-интерфейсов `fopen`, `fprintf` и `fclose` в заголовке `<cstdio>`.

Функция `fopen` открывает файл и имеет следующую сигнатуру:

```
FILE*❶ fopen(const char *имя-файла❷, const char *режим❸);
```

В случае успеха `fopen` возвращает ненулевой `FILE*` ❶. В случае неудачи `fopen` возвращает `nullptr` и устанавливает статическую переменную `int errno` равной коду ошибки, например отказ в доступе (`EACCES = 13`) или отсутствие такого файла (`ENOENT = 2`).

ПРИМЕЧАНИЕ

В заголовке `errno.h` приведен список всех состояний ошибок и их соответствующих значений `int`.

Дескриптор файла `FILE*` — это ссылка на файл, которым управляет операционная система. *Дескриптор* — это непрозрачная абстрактная ссылка на некоторый ресурс в операционной системе. Функция `fopen` принимает два аргумента: `имя-файла` ❷ — это путь к файлу, который нужно открыть, а `режим` ❸ — это один из шести параметров, показанных в таблице 11.2.

Файл должен быть закрыт вручную с помощью `fclose` после завершения его использования. Неспособность закрыть дескрипторы файлов является распространенным источником утечек ресурсов, например:

```
int fclose(FILE* file);
```

Таблица 11.2. Опции для всех шести режимов `fopen`

Строка	Операции	Файл существует:	Файл не существует:	Примечания
`r`	Чтение		Провал `fopen`	
`w`	Запись	Перезаписать	Создать файл	Если файл существует, все содержимое отбрасывается
`a`	Добавление		Создать файл	Запись всегда в конец файла
`r+`	Чтение/запись		Провал `fopen`	
`w+`	Чтение/запись	Перезаписать	Создать файл	Если файл существует, все содержимое отбрасывается
`a+`	Чтение/запись		Создать файл	Запись всегда в конец файла

Для записи в файл можно использовать функцию `fprintf`, которая похожа на `printf`, но выводит результат в файл вместо консоли. Функция `fprintf` имеет идентичное использование с `printf`, за исключением того, что в качестве первого аргумента перед строкой формата предоставляется дескриптор файла:

```
int❶ fprintf(FILE* file❷, const char* format_string❸, ...❹);
```

В случае успеха `fprintf` возвращает количество символов ❶, записанных в открытый файл ❷. Строка `format_string` такая же, как строка формата для `printf` ❸, так же, как и вариативные аргументы ❹.

Можно использовать `std::unique_ptr` для `FILE`. Очевидно, что не нужно вызывать `delete` для дескриптора файла `FILE*` в момент закрытия файла. Вместо этого нужно закрыть файл с помощью `fclose`. Поскольку `fclose` является функционально-подобным объектом, принимающим `FILE*`, он является подходящим удалителем.

Программа в листинге 11.13 записывает строку `HELLO DAVE` в файл `HAL9000` и использует уникальный указатель для управления ресурсами открытого файла.

Листинг 11.13. Программа, использующая std::unique_ptr и пользовательский удалитель для управления дескриптором файла

```
#include <cstdio>
#include <memory>

using FileGuard = std::unique_ptr<FILE, int(*)(FILE*)>; ❶

void say_hello(FileGuard file❷) {
  fprintf(file.get(), "HELLO DAVE"); ❸
}

int main() {
  auto file = fopen("HAL9000", "w"); ❹
  if (!file) return errno; ❺
  FileGuard file_guard{ file, fclose }; ❻
```

```
    // Открытие файла
    say_hello(std::move(file_guard)); ❼
    // Закрытие файла
    return 0;
}
```

Этот листинг создает псевдоним типа `FileGuard` ❶ для краткости. (Обратите внимание, что тип удалителя соответствует типу `fclose`.) Далее следует функция `say_hello`, которая принимает `FileGuard` по значению ❷. В пределах `say_hello` используется `fprintf` для вывода `HELLO DAVE` в `file` ❸. Поскольку время жизни файла привязано к `say_hello`, файл закрывается после возврата `say_hello`. В `main` файл `HAL9000` открывается в режиме `w`, который создаст или перезапишет файл, и сохраняется дескриптор обычного файла `FILE*` в `file` ❹. Вы проверяете, имеет ли файл значение `nullptr`, что указывает на ошибку, и возвращаете результат с ошибкой `errno`, если `HAL9000` не может быть открыт ❺. `FileGuard` создается при передаче файла дескриптора и пользовательского удалителя `fclose` ❻. На этом этапе файл открыт, и благодаря его собственному удалителю `file_guard` автоматически управляет временем жизни файла.

Чтобы вызвать `say_hello`, нужно передать право собственности на эту функцию (потому что она принимает `FileGuard` по значению) ❼. Вспомните из «Категорий значений», что такие переменные, как `file_guard`, являются `l`-значениями. Это означает, что нужно переместить ее в `say_hello` с помощью `std::move`, который записывает `HELLODAVE` в файл. Если опустить `std::move`, компилятор попытается скопировать его в `say_hello`. Поскольку `unique_ptr` имеет конструктор удаленных копий, это приведет к ошибке компилятора.

Когда `say_hello` возвращает результат, его аргумент `FileGuard` уничтожается и пользовательский удалитель вызывает `fclose` для дескриптора файла. По сути, утечка дескриптора файла невозможна. Он связан со временем существования `FileGuard`.

Неполный список поддерживаемых операций

Таблица 11.3 перечисляет все поддерживаемые операции `std::unique_ptr`. В этой таблице `ptr` означает обычный указатель, `u_ptr` — уникальный указатель, а `del` — удалитель.

Таблица 11.3. Все поддерживаемые операции `std::unique_ptr`

Операция	Примечания
`unique_ptr{ }` или `unique_ptr{ nullptr }`	Создает пустой уникальный указатель с удалителем `std::default_delete<...>`
`unique_ptr{ ptr }`	Создает уникальный указатель, владеющий динамическим объектом, на который указывает `ptr`. Использует удалитель `std::default _delete<...>`

продолжение ⇩

Таблица 11.3. *(продолжение)*

Операция	Примечания
`unique_ptr{ ptr, del }`	Создает уникальный указатель, владеющий динамическим объектом, на который указывает `ptr`. Использует `del` в качестве удалителя
`unique_ptr{ move(u_ptr) }`	Создает уникальный указатель, владеющий динамическим объектом, на который указывает уникальный указатель `u_ptr`. Переносит владение из `u_ptr` во вновь созданный уникальный указатель. Также перемещает удалитель `u_ptr`
`~unique_ptr()`	Вызывает удалитель принадлежащего объекта, если полный
`u_ptr1 = move(u_ptr2)`	Передает право собственности на принадлежащий объект и удалитель из `u_ptr2` в `u_ptr1`. Уничтожает принадлежащий объект, если полный
`u_ptr1.swap(u_ptr2)`	Обменивается собственными объектами и удалителями между `u_ptr1` и `u_ptr2`
`swap(u_ptr1, u_ptr2)`	Свободная функция, идентичная методу `swap`
`u_ptr.reset()`	Если полный, вызывает удалитель для объекта, принадлежащего `u_ptr`
`u_ptr.reset(ptr)`	Удаляет принадлежащий объект; затем вступает во владение `ptr`
`ptr = u_ptr.release()`	Возвращает обычный указатель `ptr`; `u_ptr` становится пустым. Удалитель не вызывается
`ptr = u_ptr.get()`	Возвращает обычный указатель `ptr`; `u_ptr` сохраняет право владения
`*u_ptr`	Оператор разыменования на принадлежащем объекте
`u_ptr->`	Оператор разыменования члена на принадлежащем объекте
`u_ptr[index]`	Ссылка на элемент по индексу (только для массивов)
`bool{ u_ptr }`	Преобразование `bool`: `true`, если полный, `false`, если пустой

`u_ptr1 == u_ptr2`
`u_ptr1 != u_ptr2`
`u_ptr1 > u_ptr2`
`u_ptr1 >= u_ptr2`
`u_ptr1 < u_ptr2`
`u_ptr1 <= u_ptr2`
	Операторы сравнения; эквивалентны вычислению операторов сравнения на обычных указателях

`u_ptr.get_deleter()`	Возвращает ссылку на удалитель