

Контрольные вопросы

19 октября 2022 г.

1 Перечислите все специальные функции-члены класса, включая перемещающие операции.

Специальные функции-члены класса:

- конструктор по умолчанию;
- пользовательские конструкторы;
- деструктор;
- copy assignment оператор;
- copy конструктор;
- move assignment оператор;
- move конструктор.

2 Приведите примеры операторов, которые можно, нельзя и не рекомендуется перегружать.

Нельзя перегрузить операторы:

- conditional (`?:`);
- `sizeof`;
- scope (`::`);
- member selector (`.`);
- member pointer selector (`.*`);
- `typeid`;
- casting operators.

Все остальные операторы могут быть перегружены, например:

- plus operator (`+`);
- binary minus operator (`-`);
- multiplication operator (`*`);
- division operator (`/`);
- многие другие...

Наконец, не рекомендуется перегружать операторы:

- logical 'and' operator (`&`);
- logical 'or' operator (`||`);
- comma operator (`,`).

3 О каких преобразованиях стоит помнить при проектировании операторов?

При проектировании операторов следует помнить о:

- перегруженном приведении типов (с помощью перегрузки оператора приведения типов);
- о неявном приведении типов (с помощью перегруженных пользовательских конструкторов, поэтому при использовании этого способа необходимо наличие конструктора без ключевого слова `explicit`).

Это связано с тем, что компилятор работает по следующему принципу: если какой-либо из операндов оператора является пользовательским типом данных, компилятор проверяет, есть ли у данного типа соответствующая перегруженная операторная функция, которую он может вызвать, но, если он не может найти её, он попытается преобразовать один или несколько операндов пользовательского типа в фундаментальные типы данных, чтобы он мог использовать соответствующий встроенный оператор (через перегруженное приведение типов либо с помощью неявного преобразования типов). Если это не удастся, то это приведет к ошибке компиляции. Кроме того, может возникнуть неоднозначность выбора между неявным преобразованием с помощью конструктора или оператора приведения типов.

4 Опишите классификацию выражений на основе перемещаемости и идентифицируемости.

Идентифицируемость (identity) – свойство выражения, заключающееся в наличии какого-либо параметра, по которому можно понять, ссылаются ли два выражения на одну и ту же сущность или нет. Перемещаемость (mobility) – свойство выражения, заключающееся в возможности поддержания семантики перемещения.

Обладающие идентичностью выражения обобщены под термином glvalue (generalized values), перемещаемые выражения называются rvalue. Комбинации двух этих свойств определили 3 основные категории выражений:

- lvalue – идентифицируемое неперемещаемое выражение;
- xvalue – идентифицируемое перемещаемое выражение;
- prvalue – неидентифицируемое перемещаемое выражение.

На самом деле, в Стандарте C++17 появилось понятие избегание копирования (copy elision) – формализация ситуаций, когда компилятор может и должен избегать копирования и перемещения объектов. В связи с этим, prvalue не обязательно могут быть перемещены.

5 Зачем нужны rvalue ссылки?

rvalue ссылки позволяют:

- продлить срок жизни объекта, которым они инициализированы до срока жизни rvalue ссылки (подобно lvalue ссылкам на константные объекты);
- неконстантные rvalue ссылки позволяют изменять rvalue.

Правда, эти свойства используются редко. Гораздо чаще rvalue ссылки используются в качестве аргументов функции. Это бывает особенно полезно, при перегрузки функций, когда имеет смысл разграничить реализации функций для lvalue и rvalue ссылок. Это свойство используется для реализации семантики перемещения. Поэтому rvalue ссылки дают возможность сделать код семантически более правильным, привнести в него дополнительную скорость за счёт семантики перемещения, а также предоставляют надежное средство для передачи параметров во внутренние функции без большого количества перегруженных функций.

6 Почему семантика перемещения лучше копирования?

Семантика перемещения позволяет повторно использовать объекты, время жизни которых приближается к концу, с помощью операций перемещения, что позволяет в некоторых ситуациях избежать дорогостоящих операций копирования (особенно "глубокого" копирования). Благодаря операциям перемещения вместе с самим объектом могут быть перемещены связанные с ним сущности, что может быть более удобно (и даже более безопасно) чем их копирование. Кроме того, с помощью семантики перемещений можно реализовать объекты, которые не могут копироваться (или их копирование нежелательно).

Но не всё однозначно. Так, С. Мейерс пишет: "Перемещающие операции не всегда дешевле копирования, а когда и дешевле, то не всегда настолько, как вы думаете; кроме того, они не всегда вызываются в контексте, где перемещение является корректным. Конструкция `type &&` не всегда представляет rvalue-ссылку".

7 Что делает функция `std::move` и когда нет необходимости явно её вызывать?

Функция `std::move` выполняет безусловное приведение своего аргумента к `rvalue`. Функция `std::move` ничего не перемещает (более того она даже не гарантирует, что приведенный этой функцией объект будет иметь право быть перемещенным).

Функцию `std::move` нет необходимости явно вызывать, если она:

- предотвращает выполнение компилятором оптимизации `copy/move elision`;
- избыточна, поскольку уже используется неявно.

Первый случай реализуется в результате оптимизации именованного возвращаемого значения (NRVO). Это возможно, если, например, возвращаемая функцией переменная является `non-volatile` (значение не может меняться извне и компилятор не будет оптимизировать эту переменную) локальным объектом некоторого класса и не является параметром функции. Тогда компилятор может создать объект непосредственно в месте его конечного назначения (ячейке стека вызова). Использование `std::move` в данном случае будет конфликтовать с реализацией NRVO, требующего возврата имени в функции, в то время как `std::move` возвращает ссылку.

Второй случай реализуется в результате оптимизации возвращаемого значения (RVO) при некоторых условиях, когда невозможно выполнение оптимизации `copy/move elision` компилятором. Это возможно, если, например, возвращаемая функцией переменная является объектом некоторого класса и при этом – ещё и аргументом данной функции. В таком случае C++ гарантирует использование операции перемещения по умолчанию и выполнение `two-stage overload resolution` как будто объект являлся `rvalue`. Аргумент функции, будучи `lvalue`, становится `xvalue`, так как он находится "на грани уничтожения" (он действительно исчезнет после выхода из области видимости функции). В результате при возвращении функцией значения неявно используется `std::move`, и она возвращает `rvalue`-объект.

8 Кем выполняется непосредственная работа по перемещению?

Непосредственная работа по перемещению выполняется специальными перемещающими операциями, имплементируемыми специальными функциями-членами класса: `move assignment` оператором (`=`) и `move` конструктором. Первый работает непосредственно с объектами класса, а второй служит для инициализации.

`Move` конструктор и `move assignment` оператор вызываются, когда эти функции определены в классе, а аргументом для конструктора или присваивания является `rvalue`, часто являющееся литералом или временным значением.

В большинстве случаев `move` конструктор перемещения и `move assignment` оператор не создаются по умолчанию, если в классе нет чего-либо из следующих специальных функций-членов: `copy` конструктора, `copy assignment` оператора, `move assignment` оператора или деструктора.

9 Когда может потребоваться пользовательская реализация специальных функций-членов класса?

Использование shallow (memberwise) copy (поверхностные копии), реализуемое с помощью copy конструктора по умолчанию copy assignment оператора по умолчанию, может привести к UB. Так, shallow copy указателя может привести к проблеме, когда объект класса и его копия имеют разные указатели, ссылающиеся на одну и ту же сущность, что потенциально может привести к висячему указателю. От этой проблемы можно избавиться с помощью deep copy (глубокого копирования), благодаря которому разные указатели будут ссылаться на разные копии. Тогда и необходима пользовательская реализация специальных функций-членов. В целом, стоит помнить что:

- copy конструктор по умолчанию и copy assignment операторы по умолчанию делают shallow copy, что хорошо для классов, не содержащих динамически выделяемых переменных;
- классы с динамически выделяемыми переменными должны иметь copy конструктор и copy assignment оператор присваивания, которые выполняют deep copy;
- классы из стандартной библиотеки более предпочтительны, чем собственное управление памятью (там многое уже реализовано).

10 Для чего нужны ключевые слова `default` и `delete` в объявлении специальных функций-членов класса?

Каждый класс может явно выбирать, какие из специальных функций-членов существуют с их определением по умолчанию, а какие удаляются с помощью ключевых слов `default` и `delete` соответственно.

Ключевое слово `default` в объявлении специальных функций членов-классов необходимо для создания специальной функции-члена (`default/move/copy`) по умолчанию без аргументов. При этом соответствующие пользовательские конструкторы не должны иметь все аргументы по умолчанию (это касается `default` конструктора). Кроме того, `default` повышает читабельность кода.

Ключевое слово `delete` в объявлении специальных функций членов-классов необходимо для запрета на использование данной специальной функции-члена компилятором. Любое использование `deleted` функции (например, при явном преобразовании типов) приведёт к ошибке компиляции.

Литература

- [1] <https://www.learncpp.com/>
- [2] <https://scrutator.me/post/2011/08/02/rvalue-refs.aspx>
- [3] <https://habr.com/ru/post/441742/>
- [4] Мейерс, Скотт. Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14.: Пер. с англ. – М. : ООО "ИЛ. Вильямс 2016. – 304 с.: ил. – Парал. тит. англ.
- [5] <https://cplusplus.com/>