Контрольные вопросы

9 ноября 2022 г.

1 Что такое шаблон и какие разновидности шаблонов существуют?

Шаблон – это функция или класс (на самом деле, не только – см. разновидности шаблонов далее), написанные для одного или нескольких типов данных, которые пока ещё не известны. При использовании шаблона в качестве аргументов ему явно или неявно передаются конкретные типы данных. Разновидности шаблонов:

- Шаблоны функций – параметризованные функции; иными словами, шаблон функции представляет собой целое семейство функций. Он очень похож на обычную функцию, но отличается тем, что некоторые элементы этой функции остаются неопределенными и являются параметризованными.

- Шаблоны классов – параметризованные классы; иными словами, это шаблон для создания экземпляров классов (точнее классов, структур, объединений, включая т.н. шаблоны агрегатных классов – классов/структур без пользовательских, явных или унаследованных конструкторов, без закрытых или защищенных нестатических членов-данных, без виртуальных функций и без виртуальных, закрытых или защищенных базовых классов). Яркий пример подобных шаблонов – классы контейнеров STL, которые используются для управления элементами определенного типа.

- Шаблоны псевдонимов (начиная с C++11) – параметризованные объявления псевдонимов (возможны только с использованием ключевого слова using). Этот шаблон предоставляет удобное имя для семейства типов. Шаблоны псевдонимов особенно полезны для определения сокращений для типов, являющихся членами шаблонов классов.

- Шаблоны переменных (начиная с C++14) – параметризованные переменные const или constexpr или параметризованные статические данные-члены классов/структур. Эти шаблоны могут понадобиться если есть необходимость использовать параметризованную переменную, например, в шаблоне функции. Особенно часто используются для математических констант.

2 Каким образом осуществляется двухэтапная трансляция шаблона?

Попытка инстанцирования шаблона для типа, который не поддерживает все используемые в шаблоне операции, приведёт к ошибке времени компиляции. Это происходит из-за процесса двухэтапной трансляции шаблона:

- Во время определения (definition time) код шаблона проверяется на корректность без инстанцирования, с игнорированием параметров шаблонов. Этот процесс включает:
  - выявление таких синтаксических ошибок, как отсутствующие точки с запятой;
  - выявление применения неизвестных имен (имен типов, функций и т.д.), которые не зависят от параметров шаблона;
  - выполнение проверок статических утверждений, не зависящих от параметров шаблонов.
- Во время инстанцирования код шаблона вновь проверяется на корректность. Таким образом, все части, которые зависят от параметров шаблонов, подвергаются двойной проверке.

Проверка имен, выполняемая дважды, называется двухэтапным (двухфазным) поиском (two-phase, lookup).

Отметим важную проблему, возникающую из-за двухэтапной трансляции при практической работе с шаблонами: когда шаблон функции используется способом, запускающим его инстанцирование, компилятору (в определенный момент) потребуется определение этого шаблона. Т.е. теперь объявление и определение шаблона не могут быть разделены. Это приводит к отходу от обычной практики, различающей компиляцию и компоновку для нешаблонных функций, когда объявления функции достаточно для компиляции кода с ее использованием.

3 Что предпочитает компилятор при перегрузке шаблона функции?

Шаблоны не компилируются в единую сущность, способную обработать любой тип данных. Вместо этого из шаблона генерируются различные объекты для каждого типа, для которого применяется шаблон. Таким образом, возможна перегрузка шаблона функции, и нешаблонная функция может сосуществовать с шаблоном функции с тем же именем (который может быть инстанцирован с тем же типом). При прочих равных факторах, в процессе разрешения перегрузки компилятор предпочитает нешаблонные функции тем, которые генерируются из шаблона. В остальном, при перегрузке шаблонов следует учитывать следующие аспекты:

- Если шаблон может генерировать функцию с лучшим соответствием, то компилятор выберет шаблон.
- При явном указании пустого списка аргументов шаблона разрешить вызов может только шаблон, а параметры шаблона должны быть выведены из аргументов вызова.
- Если при при вызове функции указаны параметры, которые могут быть неявно приведены к параметрам в объявлении/определении функции, то, поскольку автоматическое преобразование типов не рассматривается при выводе параметров шаблона, будет использована нормальная функция (если шаблон не может быть использован).
- При перегрузке шаблонов функций возможно возникновение ошибки неоднозначности, поэтому следует убедиться, что любому вызову соответствует только один из шаблонов.

4 В чём заключается особенность инстанцирования шаблонов классов?

Шаблон класса обычно выполняет несколько операций над аргументами шаблона, для которых он инстанцируется (в том числе конструирование и уничтожение). Это может привести к впечатлению, что аргументы шаблона должны предоставлять все операции, необходимые для всех функций-членов шаблона класса. Но это не так: аргументы шаблона обязаны предоставлять только те операции, которые необходимы (а не те, которые могут потребоваться). Таким образом, при инстанцировании шаблонов классов инстанцирование происходит только для вызываемых функций (членов) шаблона. Для шаблонов классов функции-члены инстанцируются только при их использовании. Такой подход позволяет сэкономить время, память и использовать шаблоны классов только частично.

Бывает так, что некоторая функция шаблона класса невыполнима для определённых типов данных (например, не для всех типов прописан operator«). Тогда при использовании такой функции будет происходить compile-time error.

Friend функции для шаблонов классов могут являюьтся не шаблонами функции, а нормальными функциями, инстанцируемыми при необходимости с использованием шаблонов классов. Такие функции называются шаблонизированными сущностями (templated entity). Такая реализация friend функций является наиболее простой.

Если шаблон класса имеет статические члены, то они инстанцируются однократно для каждого типа, для которого используется шаблон класса.

5 Когда необходимы полная или частичная специализации шаблона?

Специализированные шаблоны классов позволяют оптимизировать реализации для конкретных типов или корректировать неверное поведение определенных типов при инстанцировании шаблона класса. Однако при специализации шаблона класса необходимо специализировать все его функции-члены. Хотя можно специализировать и отдельную функцию-член шаблона класса, но если сделать это, то потом нельзя будет специализировать целый шаблон класса, которому принадлежит специализированный член.

Шаблоны классов могут быть специализированы частично (с одним или несколькими параметрами, причём возможно неоднозначное объявление экземпляра шаблонного класса). Можно определить частные реализации для определенных условий, но при этом некоторые параметры шаблона все ещё будут оставаться задаваемыми пользователем. Частичная специализация может оказаться полезной, например, когда у шаблона класса есть несколько параметров, но в силу некоторых причин, второй из них (при фиксации первого) определяет поведение некоторых функций-членов, которые мы и можем изменить для данной пары параметров.

Важно отметить, что специализация может предоставлять иной интерфейс для пользователя (обычно общедоступный интерфейс стараются сохранять). Кроме того, полностью специализированная версия имеет приоритет над частично специализированной версией.

Итак, полная или частичная специализации шаблона используются, когда:

- Есть необходимость оптимизировать/скорректировать реализацию кода для некоторого типа данных:
  - Оптимизация возможна, например, если некоторый тип данных (например, bool) занимает значительно меньше памяти, чем другие. Тем самым возможно сокращение неиспользуемого пространства.
  - Корректирование может быть очень важно, чтобы избежать некоторых серьёзных ошибок. Например, могут потребоваться специализированный конструктор и деструктор при работе с типами данных, связанными с динамической памятью (например, с указателями – может оказаться что произойдёт shallow copy при инициализации указателя указателем в "общем"конструкторе, поэтому удобным решением является использование частичной специализации для указателей всевозможных типов или полной специализации в некоторых других случаях).

- (Если речь идёт о специализации шаблонов функций) нет возможности использовать перегрузку шаблонов функций (см. один из ответов [2]) или перегрузка шаблонов функций является слишком громоздкой (например, если требуется отдельно рассматривать указатели на типы и сами типы данных). Также в [2] упоминается, что у перегрузок не всё хорошо с областями видимости: "If the function is specified as being in one scope, but the overload is in another scope, then a call to the function's qualified name will miss the overload"(правда, я не очень понимаю, почему).

6 Для какого типа специализация стандартного шаблона класса std::vector имеет смысл с точки зрения оптимизации потребления памяти и почему?*

Специализация стандартного шаблона класса std::vector имеет смысл для типа bool. Способ, которым std::vector<bool> эффективно использует память (а также оптимизация std::vector, в целом) определяется реализацией. Одна потенциальная оптимизация представляет собой объединение векторных элементов таким образом, чтобы каждый элемент занимал один бит, а не размер sizeof(bool) байт (поскольку в таком случае память будет использована гораздо эффективнее). (В действительности, можно перечислить, какие изменения включает в себя специализация std::vector<bool>). Эти реализации обеспечивают необычный интерфейс для рассматриваемой специализации и отдают предпочтение оптимизации памяти, а не обработке (которая может соответствовать или не соответствовать потребностям пользователя). В любом случае невозможно напрямую создать экземпляр неспециализированного шаблона вектора для логического значения.

Т.о., std::vector<bool> хранит биты, а не булевские значения, и поэтому не может вернуть ссылки bool& по оператору индексации или разыменованию итератора. И в данном случае невозможно взять адрес бита внутри байта, поэтому, например, operator[] не может вернуть ссылку bool&, но вместо этого возвращает прокси-объект, который позволяет манипулировать конкретным рассматриваемым битом.

Кроме того, в [3] приводятся аргументы, доказывающие что динамический массив битов – очень хорошая структура данных при оптимизации алгоритмов, которые могут обрабатывать последовательность битов за раз (несмотря на необычный интерфейс). В этом случае оптимизируются не только пространства, но и скорости. Если эти детали не брать во внимание, то оптимизация пространства приводит к значительному уменьшению скорости работы программы. Т.о., массив битов может быть очень быстрой структурой данных для работы различных алгоритмов (например, алгоритма поиска).

Литература

[1] https://www.learncpp.com/

[2] https://stackoverflow.com/questions/3692510/c-what-is-the-purpose-of-function-template-specialization-when-to-use-it

[3] https://isocpp.org/blog/2012/11/on-vectorbool

[4] https://stackoverflow.com/questions/17794569/why-isnt-vectorbool-a-stl-container

[5] https://cplusplus.com/reference/vector/vector-bool/

[6] https://en.cppreference.com/w/cpp/container/vector_bool