Программирование на C++

Лекция 5 Шаблоны классов. Часть 2.

Концепты

Давайте вспомним последний слайд предыдущей лекции. Напомним, что мы рассматривали частичное использование шаблонов и внутри класса стек определили функцию-член printOn, которая по сути была оберткой над operator<<, который определен только для некоторых стандартных типов. Тем не менее, мы могли использовать этот шаблон класса для типов, для которых данный оператор не определен, до тех пор, конечно, пока не попытаемся вызвать эту функцию. В этом случае возникнет ошибка времени компиляции, так как компилятор не сможет инстанцировать вызов operator<< для этого типа.

Возникает вопрос, как узнать, какие операции нам необходимы чтобы шаблон мог быть инстанцирован?

601-800
9

370
13

Концепты

Термин «концепт» (concept) часто используется для обозначения множества ограничений, которые многократно требуются в библиотеке шаблонов. Например, стандартная библиотека C++ опирается на такие ограничения, как *итератор с произвольным доступом* и *наличие конструктора по умолчанию.*

По стандарту C++ 17 концепты могут быть более или менее выражены только в документации (например, с помощью комментариев в коде). Это может доставить серьезные проблемы, так как нарушение ограничений приведет к «жутким» сообщениям об ошибках.

Концепты

В течение многих лет производились попытки ввести определения и проверки концептов в качестве возможностей самого языка, однако до стандарта C++ 17 и включая его такой подход так и не был стандартизован. Как обстоит дело в настоящее время прочитайте по ссылке на предыдущем слайде (если вы этого еще не сделали☺)

601-800
9

370
13

Концепты

Начиная с C++ 11 существует возможность выполнения по крайней мере некоторых проверок основных ограничений с использованием ключевого слова *static_assert* и некоторых предопределенных свойств типов:

```
template<typename T>
class C {
    static_assert(std::is_default_constructible<T>::value, "class C requires"
        "default-constructed members");
    ...
};
```

Однако для проверки, например, того, что объекты типа T предоставляют определенную функцию-член ли что их можно сравнивать с помощью оператора <, требуется куда более сложный код. Пока нам достаточно просто знать, что возможность таких проверок существует и необходима.

Не имей сто рублей, а имей сто друзей. Ключевое слово friend

Вернемся к нашему примеру шаблонного класса Stack<>. Вместо печати содержимого с помощью функции printOn() лучше реализовать operator<< для стека. Однако operator<< должен быть реализован как свободная функция. Это означает, что operator<< для класса Stack<> не является шаблоном функции, а обычной функцией инстанцируемой при необходимости с использованием шаблона класса (так называемая шаблонизированная сущность).

Однако при попытках объявить дружественную функцию и определить ее позже все оказывается более сложным.

Не имей сто рублей, а имей сто друзей. Ключевое слово friend

Вернемся к нашему примеру шаблонного класса Stack<>. Вместо печати содержимого с помощью функции printOn() лучше реализовать operator<< для стека. Однако operator<< должен быть реализован как свободная функция. Это означает, что operator<< для класса Stack<> не является шаблоном функции, а обычной функцией инстанцируемой при необходимости с использованием шаблона класса (так называемая шаблонизированная сущность).

Однако при попытках объявить дружественную функцию и определить ее позже все оказывается более сложным.

```
friend std::ostream& operator<<(std::ostream& strm, Stack<T> const& st);
```

Такое объявление внутри класса Stack приведет к ошибке

Не имей сто рублей, а имей сто друзей. Ключевое слово friend

Фактически у нас есть два варианта:

1. Мы можем неявно объявить новый шаблон функции, который должен использовать отличный от параметра класса параметр шаблона, например U:

```
template<typename U>
friend std::ostream& operator<<(std::ostream& strm, Stack<U> const&);
```

Не имей сто рублей, а имей сто друзей. Ключевое слово friend

2. Мы можем предварительно объявить оператор вывода для Stack<T> как шаблон, что означает, что мы предварительно должны объявить Stack<T>:

```
template<typename T>
class Stack;
template<typename T>
std::ostream& operator<<(std::ostream&, Stack<T> const&);
```

Затем мы можем объявить эту функцию другом (friend):

```
template<typename T>
class Stack {
    friend std::ostream& operator<<<T>(std::ostream& strm, Stack<T> const&);
```

Специализация шаблонов классов

Шаблон класса можно специализировать для конкретных аргументов шаблона. Так же, как и в случае перегрузки шаблонов функций 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.

601-800
9

370
13

Специализация шаблонов классов

Чтобы специализировать шаблон класса, следует объявить класс с предваряющей конструкцией template<> и указать типы, для которых специализируется шаблон класса. Типы используются в качестве аргументов шаблона и должны задаваться непосредственно после имени класса:

```
template<>
class Stack<std::string> {
    ...
};
```

Для таких специализаций любая функция-член должна определяться как обычная функция с заменой каждого включения T специализированным типом, например:

```
void Stack<std::string>::push(std::string const& elem) {

    //реализация
}
```

Специализация шаблонов классов

Ниже приведен полностью завершенный пример специализации Stack<> для строк:

```
template<>
class Stack<std::string> {
private:
    std::deque<std::string> elems; // элементы
public:
    void push(std::string const&); // внесение элемента в стек
    void pop(); // удаление элемента из стека
    std::string const& top() const; // возврат последнего элемента
    bool empty() const // проверка на пустоту
    {
        return(elems.empty());
    }
};

void Stack<std::string>::push(std::string const& elem) {
    elems.push_back(elem); // добавление копии переданного элемента
}

void Stack<std::string>::pop() {
    assert(!elems.empty());
    elems.pop_back(); // удаление последнего элемента
}

std::string const& Stack<std::string>::top() const {
    assert(!elems.empty());
    return(elems.back()); // возврат последнего элемента
}
```

В данном примере специализация использует семантику ссылок для передачи строкового аргумента функции push(), которая именно для данного типа имеет больше смысла.
Кроме того, для хранения элементов мы использовали deque, чтобы продемонстрировать, что реализация специализации может значительно отличаться от реализации первичного шаблона

Частичная специализация шаблонов классов

Когда мы с вами обсуждали специализацию шаблонов функций, то сказали, что частичная специализация есть только у шаблонов классов. Давайте рассмотрим, как это работает. Мы можем определить частные реализации для определенных условий, при этом некоторые параметры шаблона все еще остаются задаваемые пользователем. Например, можно определить отдельную специализацию Stack<> для указателей:

```
// Частичная специализация Stack<> для указателей
template<typename T>
class Stack <T*>
{
private:
    std::vector<T*>elems; // элементы
public:
    void push(T*); // добавление элемента в стек
    T* pop(); // удаление элемента из стека
    T* top() const; // Возврат последнего элемента
    bool empty() const {
        return(elems.empty());
    }
};
```

Реализация функций-членов класса на следующем слайде

Частичная специализация шаблонов классов

```cpp
template<typename T>
void Stack<T*>::push(T* elem) {
    elems.push_back(elem); // добавление копии переданного элемента
}

template<typename T>
T* Stack<T*>::pop() {
    assert(!elems.empty());
    T* p = elems.back();
    elems.pop_back(); // удаление последнего элемента и его возврат (в отличие от общего случая)
    return(p);
}

template<typename T>
T* Stack<T*>::top() const {
    assert(!elems.empty());
    T* p = elems.back();
    return(p); // возврат копии последнего элемента
}
```

С помощью
template<typename T>
class Stack<T*>{};
Мы определяем шаблон класса, параметризованный T, но специализированный для указателей (Stack<T*>)

Частичная специализация шаблонов классов

Еще раз обратите внимание на то, что специализация может предоставлять (несколько) иной интерфейс. Например, в приведенном коде функция pop() возвращает хранимый указатель, так что пользователь шаблона класса может вызвать delete для удаляемого значения, если оно было создано с помощью оператора new:

```
#include <iostream>
#include<string>
#include "stack1.hpp"

int main()
{
    Stack<int*> intPtrStack;
    intPtrStack.push(new int{ 42 });
    std::cout << *intPtrStack.top() << '\n';
    delete intPtrStack.pop();
```

Шаблоны классов. Часть 2. Конспект

Сделайте краткий письменный конспект лекции, отвечая на следующие вопросы:

1. В чем заключается основная идея концепта?
2. Какая ошибка возникает в случае объявления оператора operator<< как friend внутри класса Stack<> без предварительного объявления вне класса и без использования другого параметра шаблона. Приведите пример, как это можно исправить. Можете обратиться сюда, здесь неплохое обсуждение данного вопроса ☺
3. Что такое специализация шаблона класса? Приведите пример.
4. Что такое частичная специализация шаблона класса. Приведите пример.

Шаблоны классов. Задание 1

Реализуйте частичную специализацию своего шаблонного класса Очередь (Queue), которую вы делали к прошлому занятия, для указателей.

601-800
9

370
13

Шаблоны классов. Задание 2

Добавьте в изначальный шаблон Stack<> operator<<. Напишите специализацию Stack<> для пар (std::pair), элементы такого стека должны выводиться с помощью operator <<.

601-800
9

370
13

Спасибо за внимание