Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ВЛАСТИВОСТІ ТИПІВ. ГЕНЕРУВАННЯ СПИСКІВ ТИПІВ

**Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121**

Керівник курсової роботи
к.н., доц. Бублик В.В.
_____
(підпис)
“____” _____ 2020 р.

Виконав студент
Семенюк Х.Р.
“____” _____ 2020 р.

Київ 2020

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем

ЗАТВЕРДЖУЮ
Зав.кафедри мультимедійних систем,
доцент, к.н.
_____О. П. Жежерун
(підпис)
„____”_____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Семенюк Христині Романівні факультету інформатики 4-го курсу
ТЕМА Властивості типів. Генерування списків типів
Вихідні дані:
-
-
Зміст ТЧ до курсової роботи:
Індивідуальне завдання
Вступ
1 Метапрограмування
2 Шаблонне метапрограмування
3 Списки типів
4 Методи маніпулювання списками типів
5 Приклади різних аспектів застосування списків типів
6 Приклад застосування довільних списків типів
Висновки
Список літератури

Дата видачі „___” _____ 2020 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Властивості типів. Генерування списків типів

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	грудень 2019 – січень 2020	
2.	Огляд технічної літератури за темою роботи	січень – лютий 2020	
3.	Створення практичної частини роботи	лютий 2020	
4.	Написання текстової частини роботи	березень – квітень 2020	
5.	Надання роботи керівнику для перевірки	квітень 2020	
6.	Коригування роботи за результатами перевірки	квітень 2020	
7.	Подання роботи на кафедру для перевірки на плагіат	19.04.2020	
8.	Здача курсової роботи	19.04.2020	

Студент Семенюк Х.Р.

Керівник Бублик В.В.

“_____”_____

Зміст

Анотація

Роботу присвячено вдосконаленню реалізації списків типів, розглянутої у книзі Александреску, а також методів маніпулювання ними шляхом застосування варіативних шаблонів. Розглянуто інструменти та підходи шаблонного метапрограмування, поняття та різні аспекти застосування списків типів, варіативних шаблонів, виокремлено переваги використання варіативних шаблонів у реалізації списків типів. Реалізовано списки типів з використанням варіативних шаблонів, методи маніпулювання ними та приклад їхнього застосування – варіативний шаблон для виведення значень з довільного списку типів.

Вступ

Іноді виникає потреба написати шаблон функції чи класу, який повинен бути конкретизованим лише для певного набору типів. Якщо просто оголосити і визначити шаблон функції у заголовному файлі, його потенційно можна буде конкретизувати для будь-якого типу, а не лише для певного набору типів. З іншого боку, якщо явно конкретизувати шаблон для кожного окремого типу, вручну спеціалізувати шаблони для всіх типів із набору, доведеться виправляти, якщо набір типів зміниться, відбудеться дублювання і роздування коду, збільшиться ймовірність виникнення помилок, які важче знайти.

Уникнути всіх цих недоліків дозволяє така структура даних, як списки типів. Вони дають можливість писати один і той же код для великої кількості різних типів і доволі легко маніпулювати цим набором типів. Списки типів є центральною структурою для шаблонного метапрограмування – техніки програмування, у якій шаблони використовуються компілятором для генерування тимчасового вихідного коду, який поєднується компілятором з рештою вихідного коду і тоді компілюється. Розрахунки у метапрограмуванні відбуваються на етапі компіляції, код маніпулює типами, а не даними.

У третій главі книги Андрія Александреску «Сучасне проектування на C++», яка називається «Списки типів», розглянуто реалізацію списків типів із бібліотеки Loki. У цій реалізації використано команди препроцесора, адже без них синтаксис створення списків типів дуже незручний.

Однак з часу написання книги Александреску в C++ виникли нові можливості, зокрема, варіативні шаблони. Використавши їх, можна вдосконалити реалізацію, розглянуту в книзі.

За мету цієї роботи було поставлено вдосконалення реалізації списків типів, розглянутої у книзі Александреску, а також методів маніпулювання ними шляхом застосування варіативних шаблонів.

Щоб досягти поставленої мети, потрібно розв’язати такі завдання:

а) оглянути і описати поняття метапрограмування;

б) розглянути і описати особливості, інструменти та підходи шаблонного метапрограмування;
в) проаналізувати й описати реалізацію списків типів з книги Александреску, поняття варіативних шаблонів, реалізувати й описати списки типів з використанням варіативних шаблонів, виокремити переваги використання варіативних шаблонів у реалізації списків типів;
г) розглянути і реалізувати алгоритми методів маніпулювання списками типів, проаналізувати підходи метапрограмування, які застосовують у роботі зі списками типів;
д) оглянути приклади різних аспектів застосування списків типів;
е) реалізувати й описати приклад застосування списків типів з використанням варіативних шаблонів (у цій роботі – варіативний шаблон для виведення значень з довільного списку типів за умови, що кожен з типів допускає оператор <<).

Об’єктом дослідження цієї роботи є списки типів.

Робота складається з шести розділів.

Перший розділ присвячено загальному огляду поняття метапрограмування.

У другому розділі оглянуто інструменти та підходи шаблонного метапрограмування на основі реалізації type traits та if-else statements етапу компіляції.

У третьому розділі розглянуто реалізацію списків типів з книги Александреску, поняття варіативних шаблонів, реалізацію списків типів з використанням варіативних шаблонів, переваги використання варіативних шаблонів у реалізації списків типів.

Четвертий розділ присвячено огляду алгоритмів методів маніпулювання списками типів.

У п’ятому розділі оглянуто приклади різних аспектів застосування списків типів з використанням варіативних шаблонів, зокрема, іншу реалізацію списку типів з індексованим доступом до елементів та реалізацію поділу списку типів залежно від предиката, у якій використано дещо інші інструменти метапрограмування.

Шостий розділ присвячено огляду реалізації варіативного шаблона для виведення значень з довільного списку типів за умови, що кожен з типів допускає оператор <<.

Створено програмний продукт: списки типів з використанням варіативних шаблонів; методи маніпулювання списками типів; приклад застосування – варіативний шаблон для виведення значень з довільного списку типів за умови, що кожен з типів допускає оператор <<.

1 Метапрограмування

Метапрограмування – це техніка програмування, в якій комп'ютерні програми мають можливість розглядати інші програми як свої дані. Це означає, що деякі програми (їх називають метапрограмами) можуть читати, генерувати, аналізувати або перетворювати інші програми і навіть модифікувати себе під час роботи. У деяких випадках це дозволяє програмістам мінімізувати кількість рядків коду для вираження рішення, що, у свою чергу, скорочує час розробки. Це також надає програмам більшої гнучкості ефективно впоратися з новими ситуаціями без перекомпіляції.

Метапрограмування може використовуватися для переміщення обчислень з часу виконання на час компіляції, для генерування коду за допомогою обчислень часу компіляції та для включення коду, що може самомодифікуватись. Мова, якою написана метапрограма, називається метамовою. Мова маніпульованих програм називається цільовою (attribute-oriented) мовою програмування.

Один з підходів метапрограмування – генерація коду. При цьому підході код програми не пишеться вручну, а створюється автоматично програмою-

генератором на основі іншої програми. Такий підхід набуває сенсу, якщо при програмуванні виробляються різні додаткові правила (більш високорівневі парадигми, виконання вимог зовнішніх бібліотек, стереотипні методи реалізації певних функцій та ін.). При цьому частина коду або даних втрачає змістовний сенс і стає лише механічним виконанням правил. Коли ця частина стає значною, виникає думка задавати вручну лише змістовну частину, а решту додавати автоматично. Це і робить генератор.

Розрізняють два принципово різні види генерації коду:

а) генератор є фізично окремою бінарною програмою, необов'язково написаною цільовою мовою.

б) цільова мова є одночасно мовою реалізації генератора, так що метапрограма становить з цільовою програмою єдине ціле.

Здатність мови програмування до власного метамовлення називається рефлексією або рефлексивністю. Рефлексія – цінна для полегшення метапрограмування особливість мови.

Отже, метапрограмування – це техніка програмування, в якій програми (їх називають метапрограми) розглядають інші програми як свої дані, тобто можуть читати, генерувати, аналізувати або перетворювати їх і навіть модифікувати себе під час роботи. Метапрограмування використовується для переміщення обчислень з часу виконання на час компіляції.

2 Шаблонне метапрограмування

Шаблонне метапрограмування – це техніка метапрограмування, у якій шаблони використовуються компілятором для генерування тимчасового вихідного коду, який поєднується компілятором з рештою вихідного коду і тоді компілюється. Розрахунки відбуваються на етапі компіляції, код маніпулює типами, а не даними.

Одним з основних інструментів шаблонного метапрограмування є шаблони класів (або структур (struct)), які у випадку використання їх для

розрахунків на етапі компіляції називаються метафункціями. Параметрами таких метафункцій є параметри шаблонів класів, а значення метафункцій зберігаються у статичних (static) змінних, визначеннях типів (typedef) класу чи псевдонімах типів.

При підстановці значень параметрів у шаблон класу отримується клас. При цьому для різних наборів значень параметрів компілятор створює різні класи. Крім того, для деяких конкретних значень параметрів можна створювати особливі класи, які мають іншу реалізацію, ніж загальний шаблон класу. Такі особливі класи називаються спеціалізаціями класів. Спеціалізації класів можуть бути частковими (коли конкретні значення вказуються для частини параметрів) або повними (коли конкретні значення вказуються для всіх параметрів). Під повним шаблоном класу мається на увазі сукупність основного визначення шаблону класу і всіх його спеціалізацій. Зокрема, спеціалізації класів використовуються для організації рекурсивних циклів. У рекурсивних функціях важливо вчасно завершити рекурсію, в метапрограмування для цього використовуються спеціалізації класів для граничних значень параметрів шаблонів.

Шаблонне метапрограмування є функціональним в тому сенсі, що всі метазмінні, типи і псевдоніми типів є незмінними (const), відсутній оператор присвоєння значень змінним, змінні можливі тільки в якості параметрів метафункцій, єдиний можливий спосіб організації циклів – рекурсія.

Щоб мати можливість писати гнучкий метакод, потрібно реалізувати деякі притаманні часу виконання ідеї так, щоб вони виконувались під час компіляції, наприклад, if, for та while. Можна детальніше розглянути підходи шаблонного метапрограмування на основі реалізації if.

2.1 Реалізація if у шаблонному метапрограмуванні (тернарний оператор)

```
template <bool B, typename T, typename F>
struct if_c {
  using type = F;
};
template <typename T, typename F>
```

```
struct if_c<true, T, F> {
  using type = T;
};
```

Реалізовано метафункцію як struct. Аргументи передаються їй як параметри шаблона і значення повертається шляхом визначення члена функції псевдоніма типу.

Загальне (неспеціалізоване) визначення метафункції має три параметри шаблона: булеве (bool); тип, який потрібно повернути, якщо bool є істинним; тип, який потрібно повернути, якщо bool не є істинним. Часткове визначення шаблону відповідає випадку, коли перший параметр дорівнює true, у ньому внутрішньому типу (метазмінній, псевдоніму типу) присвоюється тип, який є другим параметром метафункції. Загальне визначення відповідає всім іншим випадкам (тобто одному випадку, коли перший параметр дорівнює false), у ньому внутрішньому типу присвоюється тип, який є третім параметром метафункції.

Компілятор повинен використовувати повну або часткову спеціалізацію метафункції, якщо вона існує, а якщо ні, то він намагається інстанціювати та використати загальне визначення. Таким чином, якщо перший параметр шаблону є істинним, тоді використовується спеціалізація, інакше використовується загальне визначення.

Це аналог тернарного оператора (ternary operator) у метапрограмуванні.

2.2 Реалізація if у шаблонному метапрограмуванні (if-else statements)

У деяких випадках потрібно відгалузити компіляцію залежно від умови або просто уникнути компіляції певної гілки, щоб зберегти час компіляції невеликим. Потрібна можливість виконувати метафункцію при виконанні певної умови. Для цього інша реалізація if підійде краще, ніж тернарний оператор.

```
template <bool B, typename T = void>
struct enable_if {};
template <typename T>
struct enable_if<true, T> {
  using type = T;
};
```

Метафункція приймає два шаблонні параметри, bool і тип, який за замовченням дорівнює void. Шаблонний параметр, як і аргумент функції, може мати лише одне значення за замовченням для всіх спеціалізацій.

Загальне визначення метафункції обробляє випадок, коли параметр дорівнює false, воно порожнє. Спеціалізація для випадку true присвоює внутрішньому типу тип, який є параметром метафункції. Таким чином метафункція виконує певні дії лише у випадку true.

2.3 Властивості типів (type traits)

Можна реалізувати структуру, яка визначає, чи є тип певним типом, наприклад map чи unordered_map. Підхід такий самий, як у розглянутих реалізаціях if, він відіграє центральну роль у шаблонному метапрограмуванні.

```
template <typename T>
struct is_std_map : std::false_type {};

template <typename K, typename V, typename Comp, typename A>
struct is_std_map<std::map<K, V, Comp, A>> : std::true_type {};

template <typename T>
struct is_std_unordered_map : std::false_type {};

template <typename K, typename V, typename Hash, typename KeyEqual,
typename A>
struct is_std_unordered_map<std::unordered_map<K, V, Hash,
KeyEqual, A>>
    : std::true_type {};
```

Загальне визначення шаблонної структури is_std_map успадковується від std :: false_type і тому має змінну-член з назвою value, значення якої дорівнює false. Спеціалізоване визначення структури is_std_map відповідає типу map і успадковується від std :: true _type, має змінну-член value, що дорівнює true. Компілятор використовує спеціалізоване визначення, якщо воно задоволене, так само, як у розглянутих реалізаціях if. Таким чином, якщо переданий структурі параметр є типом map, внутрішнє значення value буде істинним, у іншому випадку хибним.

Реалізація структури is_std_unordered_map аналогічна.

2.4 Властивості типів (type traits) з використанням оператора enable_if

Можна реалізувати структуру, яка визначає, чи є тип одним із двох конкретних типів, наприклад map і unordered_map.

Загальне визначення приймає тип, який потрібно перевірити, як параметр і успадковується від false_type. Для спеціалізації is_map можна використати enable_if, тому потрібно додати другий шаблонний параметр, який буде використаний для перевірки enable_if і матиме значення void за замовчуванням. Значення за замовчуванням потрібне для того, щоб користувачеві структури не потрібно було вказувати другий параметр шаблону.

```
template <typename T, typename = void>
struct is_map : std::false_type {};
template <typename T>
struct is_map<T, typename std::enable_if<is_std_map<T>::value or
                  is_std_unordered_map<T>::value>::type>
    : std::true_type {};
```

Спеціалізація структури приймає один параметр шаблону, тип T, який потрібно перевірити. Другий параметр містить розглянутий вище оператор enable_if, який у цьому випадку приймає результат логічного «або» між результатом is_std_map <T>::value та is_std_unordered_map <T>::value. Якщо тип T є map або unordered_map, тоді виконується спеціалізація enable_if, яка має член метазмінну тип. У цьому випадку компілятор може скомпілювати код і повинен віддати перевагу спеціалізації над загальним визначенням. Якщо результат логічного «або» хибний, то використовується загальне визначення enable_if, і компілятор не може скомпілювати код, оскільки загальний enable_if не має члена метазмінної типу. В результаті компілятор ігнорує спеціалізацію is_map і повертається до загального визначення. Нарешті, ключове слово typename перед std::enable_if у другому параметрі потрібне, щоб повідомити компілятор, що тип є метазмінною (типом), а не змінною зі значенням.

Отже, шаблонне метапрограмування – це техніка метапрограмування, у якій шаблони використовуються компілятором для генерування тимчасового

вихідного коду, розрахунки відбуваються на етапі компіляції, код маніпулює типами. Основний інструмент – метафункції – шаблони класів; параметри шаблонів класів є параметрами метафункцій; значення метафункцій зберігаються у статичних змінних, визначеннях типів класу чи псевдонімах типів. При підстановці значень параметрів у шаблон класу отримується клас. Спеціалізації класів: часткові – особливі реалізації класів для конкретних значень частини параметрів; повні – для конкретних значень усіх параметрів. Шаблонне метапрограмування є функціональним: метазмінні, типи і псевдоніми типів є незмінними, відсутній оператор присвоєння значень змінним, змінні можливі тільки в якості параметрів метафункцій, єдиний можливий спосіб організації циклів – рекурсія, для її організації використовуються спеціалізації класів.

3 Списки типів

Іноді виникає потреба написати шаблон функції чи класу, який повинен бути конкретизованим лише для певного набору типів.

Якщо просто оголосити і визначити шаблон функції у заголовному файлі, його потенційно можна буде конкретизувати для будь-якого типу. Це не є вирішенням проблеми.

З іншого боку, якщо шаблон має бути конкретизований тільки для кількох певних типів, його визначення можна перенести у вихідний файл і явно конкретизувати для кожного окремого типу. У цьому випадку можлива ситуація, коли у проекті є багато різних шаблонів функцій і кожен з них має бути конкретизований для одного і того ж набору типів. Тоді доведеться вручну спеціалізувати шаблони для всіх типів із набору і виправляти, якщо набір типів зміниться. Це надто довго, призводить до дублювання і роздування коду, збільшує ймовірність виникнення помилок, які важче знайти. Тому цей підхід також не підходить.

Списки типів є вирішенням проблеми, вони дають можливість писати один і той же код для великої кількості різних типів і доволі легко маніпулювати цим

набором типів. Для шаблонного метапрограмування списки типів є центральною структурою даних.

3.1 Списки типів у книзі Александреску

У третій главі книги Андрія Александреску «Сучасне проектування на C++», яка називається «Списки типів», розглянуто реалізацію списків типів із бібліотеки Loki. Виглядає вона так:

```
template <class T, class U>
struct Typelist {
  typedef T Head;
  typedef U Tail;
};
namespace TL
{
...алгоритми роботи зі списками типів...
}
```

Клас Typelist містить два типи, доступ до яких забезпечується внутрішніми іменами Head і Tail. Перший тип Head є будь-яким типом, другий тип Tail є класом Typelist, тобто також списком типів. Тут використана особлива властивість шаблона, яка полягає в тому, що шаблонний параметр може мати будь-який тип, у тому числі бути конкретизацією свого власного шаблона. Оскільки клас Typelist має два параметри, його завжди можна розширити, замінивши один із параметрів (прийнято другий) класом Typelist, і так багато разів. Таким чином, список, який містить три типи, виглядає так:

```
typedef Typelist<char, Typelist<signed char, unsigned char> >
  CharList;
```

Списки типів не містять жодних значень, їхні тіла пусті, не мають жодного стану і не визначають ніяких функціональних можливостей. Списки типів не призначені для створення об’єктів. Їхнє єдине призначення – надавати інформацію про типи. Тому будь-яка обробка списків типів можлива лише на етапі компіляції, а не під час виконання програми.

Для опису списків, які не містять типів або містять тільки один, необхідний клас NullType. Прийнято домовленість, що кожен список типів повинен закінчуватись класом NullType, який служить маркером кінця списку. Визначення класу NullType:

```
class NullType();
```

Таким чином, клас Typelist, що складається з одного елемента, виглядає так:

```
typedef Typelist<int, NullType> OneTypeOnly;
```

Тепер клас Typelist, що складається з трьох елементів, має такий вигляд:

```
typedef Typelist<char, Typelist<signed char, Typelist<unsigned
char, NullType> > > AllCharTypes;
```

Списки типів у такому вигляді не дуже зручно використовувати, особливо створювати, тому у файлі Typelist.h із бібліотеки Loki визначено макроси, які перетворюють рекурсію на просте перерахування. Макроси виглядають так:

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3) >
...
#define TYPELIST_50(...)...
```

Макроси – це препроцесорні "функції", тобто лексеми, створені за допомогою директиви #define, які приймають параметри подібно до функцій. Після директиви #define вказується ім'я макросу, за ним в дужках (без пробілів) параметри, відокремлені комами, і після цих дужок і пробілу визначення макросу. Макроси обробляються препроцесором – програмним інструментом, який змінює код програми для подальшої компіляції та збірки. Макрос можна умовно назвати функцією обробки і заміни програмного коду: препроцесор замінює макроси макровизначеннями.

Макроси визначено для списків типів, які містять п’ятдесят або менше елементів, але користувач бібліотеки при необхідності може визначити макроси для більшої кількості елементів завдяки тому, що кожен макрос використовує

попередній. Зручніше визначення списку типів з використанням макросів має такий вигляд:

```
typedef TYPELIST_3(signed char, short int, long int)
  SignedIntegrals;
```

3.2 Варіативні шаблони

Книга Александреску написана порівняно давно, з тих пір в C++ виникли нові можливості, зокрема, варіативні шаблони (variadic templates) – шаблони функції або класу зі змінною кількістю аргументів. Вони, як і списки типів, є структурою даних шаблонного метапрограмування. Варіативні шаблони приймають у якості своїх аргументів так званий пакет параметрів (parameter pack) – нуль або більше типів. Варіативні шаблони підтримуються мовою C++ (починаючи з стандарту C++11). Синтаксис їхнього оголошення у загальному випадку такий:

```
template<typename... Values> class tuple;
```

Цей клас прийматиме нуль або більше параметрів. Якщо потрібно заборонити використання шаблона без параметрів, то використовують наступний підхід, що вимагає використання як мінімум одного параметра:

```
template<typename First, typename... Rest> class tuple;
```

Приклад функції з варіативним шаблоном:

```
template<typename... Params> //Оголошення
void printf(Params... parameters); //Використання
```

Виклик printf(1, 2.3, "abcd") конкретизується в printf<int, double, const char*>(1, 2.3, "abcd").

Оператор еліпсису (...) має дві ролі. Коли він знаходиться зліва від назви параметра, він оголошує пакет параметрів. Використовуючи пакет параметрів, користувач прив’язує до параметрів варіативного шаблону нуль або більше

аргументів. Пакети параметрів можна також використовувати для нетипових параметрів. З іншого боку, коли оператор еліпсису знаходиться праворуч від назви параметра, він розпаковує параметри пакету в окремі аргументи. При цьому розпаковується пакет в залежності від положення еліпсису, тобто розкривається вираз, який безпосередньо прилягає до еліпсису. Наприклад:

```
template<typename T>
T bar(T t) {/*...*/}

template<typename... Args>
void foo(Args... args)
{
  /*...*/
}

template<typename... Args>
void foo2(Args... args)
{
  foo(bar(args)...);
}
```

У цьому прикладі у функції foo2, оскільки еліпсис стоїть після виклику bar(), спочатку для кожного значення з пакета параметрів args викличеться функція bar(), і у функцію foo() в якості аргументів потраплять значення, повернуті bar().

Таким чином, використання оператора еліпсису в коді призводить до повторення всього виразу, який передує еліпсису, для кожного наступного аргументу, розпакованого з пакету аргументів.

Ось ще приклади, які демонструють цю властивість застосування оператора еліпсису:

```
(const args&..) // -> (const T1& arg1, const T2& arg2, ...)
((f(args) + g(args))...) // -> (f(arg1) + g(arg1), f(arg2) + g(arg2), ...)
(f(args...) + g(args...)) // -> (f(arg1, arg2,...) + g(arg1, arg2, ...))
(std::make_tuple(std::forward<Args>(args)...)) // ->
(std::make_tuple(std::forward<T1>(arg1), std::forward<T2>(arg2), ...))
```

Кількість параметрів у пакеті можна отримати за допомогою оператора sizeof:

```
template<typename... Args>
void foo(Args... args)
{
  std::cout << sizeof...(args) << std::endl;
}

foo(1, 2.3); // 2

```

Оскільки застосування варіативних шаблонів належить до шаблонного метапрограмування, для перебору параметрів використовується рекурсія, а для її завершення – спеціалізації функції для граничних значень параметрів шаблонів, тобто для випадку, коли пакет аргументів args... порожній.

3.3 Списки типів з використанням варіативних шаблонів

Списки типів можна реалізувати з використанням варіативних шаблонів:

```
template<typename H, typename... T>
struct TypeList
{
      using Head = H;
      using Tail = TypeList<T...>;
};

template<typename H>
struct TypeList<H, EmptyList>
{
      using Head = H;
      using Tail = EmptyList;
};

template<>
struct TypeList<EmptyList> {};

...алгоритми роботи зі списками типів...
```

Перша спеціалізація класу TypeList загальна, має два параметри шаблону, перший звичайний, другий – пакет параметрів; у ній внутрішній тип Head відповідає голові списку, Tail – хвосту списку, він є класом TypeList. Друга спеціалізація відповідає списку, який складається з одного типу, у ній внутрішній тип Head є цим типом, Tail є структурою EmptyList. Третя

спеціалізація є повною і відповідає класу TypeList<EmptyList>, тобто порожньому списку.

Кожен список типів повинен закінчуватись класом EmptyList, який служить маркером кінця списку. Визначення класу EmptyList:

```
struct EmptyList {};
```

Таким чином, список, який містить чотири типи, виглядає так:

```
using typeList = TypeList<int, double, char, float,
EmptyList>;
```

Списки типів у такому вигляді зручно використовувати, адже типи задаються простим перерахуванням. Отже, визначати макроси не потрібно, на відміну від реалізації списків типів із бібліотеки Loki, яка розглянута в третій главі книги Андрія Александреску «Сучасне проектування на C++».

3.4 Переваги використання варіативних шаблонів у реалізації списків типів

Отже, було розглянуто дві реалізації списків типів: реалізація із бібліотеки Loki, яка проаналізована у третій главі книги Александреску, і реалізація з використанням варіативних шаблонів. Їхня найбільша відмінність полягає у тому, що у першій реалізації використовуються макроси, бо вони там необхідні, щоб синтаксис створення списку типів був зручним для користувача. Внаслідок цього в обох реалізаціях створення списку типів виглядає по-різному.

У першій реалізації:

```
using typeList = TYPELIST_4(int, double, char, float);
```

У другій реалізації:

```
using typeList = TypeList<int, double, char, float,
EmptyList>;
```

Переваги другого варіанта при реалізації полягають у тому, що не потрібно загромаджувати заголовний файл визначенням макросів, витрачати час на їхнє визначення, немає можливості помилитись у визначенні макросів.

Переваги другого варіанта при використанні полягають у тому, що користувачеві не потрібно турбуватись про використання макросу, відповідного кількості елементів у списку (TYPELIST_4 для чотирьох елементів і т. д.), змінювати макрос при зміні кількості елементів. Також користувач може створювати списки типів з великою кількістю елементів, наприклад, більше п’ятдесяти, і для цього йому не доведеться визначати самостійно макроси, як при використанні першого варіанта. Окрім того, у другому варіанті типи зі списку перераховуються у кутових дужках, які вказують користувачеві на те, що це саме параметри шаблона, типи; тоді як у першому варіанті типи перераховуються в круглих дужках, які зазвичай використовують для параметрів функцій і класів. Отож, можна вважати, що у другого варіанта логічніший синтаксис створення списку типів і сприймати це за перевагу.

Отже, списки типів дають можливість писати один і той же код для різних типів і маніпулювати цим набором типів. Списки типів не містять значень, не призначені для створення об’єктів, а лише для надання інформації про типи. Реалізація: клас Typelist містить два внутрішні типи: Head – будь-який тип, Tail – клас Typelist. Реалізація списків типів із бібліотеки Loki, яка розглянута у книзі Александреску, для зручного синтаксису створення списків типів потребує макросів, які перетворюють рекурсію на просте перерахування типів. Макроси визначено для списків типів, які містять п’ятдесят або менше елементів, але користувач бібліотеки може визначити макроси для більшої кількості елементів завдяки тому, що кожен макрос використовує попередній. Варіативні шаблони – шаблони функції або класу зі змінною кількістю аргументів, приймають у якості аргументів пакет параметрів – нуль або більше типів. Застосування списків типів і варіативних шаблонів належать до шаблонного метапрограмування, тому всі обчислення відбуваються на етапі компіляції, а не під час виконання програми. Якщо реалізувати списки типів з використанням варіативних шаблонів, то при

створенні списку типи задаються простим перерахуванням, отже, визначати макроси не потрібно – це найбільша відмінність від реалізації з книги Александреску. Переваги використання варіативних шаблонів у реалізації списків типів:

а) не потрібно загромаджувати заголовний файл визначенням макросів, витрачати час на їхнє визначення, немає можливості помилитись у визначенні макросів;
б) користувачеві не потрібно турбуватись про використання макросу, відповідного кількості елементів у списку (TYPELIST_4 для чотирьох елементів і т. д.), змінювати макрос при зміні кількості елементів;
в) користувач може створювати списки типів з великою кількістю елементів;
г) при створенні списку типи перераховуються не в круглих, а у кутових дужках, які вказують користувачеві на те, що це саме параметри шаблона (логічніший синтаксис).

4 Методи маніпулювання списками типів

4.1 Обчислення довжини списку

Структура Length

Вхідні дані: список типів TList

Результат: внутрішня статична константа value

Реалізація структури Length використовує часткову шаблонну спеціалізацію, що дозволяє розділяти порожній і непорожній список типів. Перша спеціалізація структури є повною і відповідає класу TypeList<EmptyList>, у ній величина value визначається як нуль, бо довжина порожнього списку типів дорівнює нулю. Друга спеціалізація, часткова, відповідає будь-якому класу TypeList<Head, Tail...>. У другій спеціалізації обчислення проводяться рекурсивно: величина value визначається як один (з урахуванням голови списку)

плюс довжина хвоста списку. Коли хвіст списку виявляється класом TypeList<EmptyList>, відбувається співпадіння з першим визначенням і рекурсія зупиняється.

4.2 Індексований доступ до елементів списку

Структура TypeAt

Вхідні дані: індекс i, список типів TList

Результат: внутрішній тип Result

Якщо список не пустий та індекс i дорівнює нулю, то внутрішній тип Result – це голова списку.

Інакше, якщо список не пустий та індекс i не дорівнює нулю, то клас Result отримується шляхом застосування алгоритму TypeAt до хвоста списку та індексу i-1.

Інакше відбувається вихід за межі допустимого діапазону зміни індексу, тобто у списку немає елемента з індексом i, породжується повідомлення про помилку на етапі компіляції.

4.3 Пошук елемента списку

Структура IndexOf

Вхідні дані: тип T, список типів TList

Результат: внутрішня статична константа value

Якщо список TList є порожнім, то значення value дорівнює мінус один.

Інакше, якщо в голові списку знаходиться тип T, то значення value дорівнює нулю.

Інакше вирахувати результат алгоритму IndexOf, застосовуючи його до типу T і хвоста списку TList, присвоїти його змінній temp.

Якщо значення temp дорівнює мінус один, то значення value дорівнює мінус один.

Інакше значення value дорівнює один плюс temp.

4.4 Додавання елемента на початок списку

Структура AppendFront

Вхідні дані: тип T, список типів TList

Результат: внутрішній тип Result

Якщо тип T є типом EmptyList і список типів TList є порожнім, то клас Result є порожнім списком типів.

Інакше, якщо список типів TList є порожнім, то клас Result є списком типів, що складається з єдиного типу T.

Інакше, якщо тип T є типом EmptyList, то клас Result є списком типів TList.

Інакше клас Result є списком типів, голова якого є типом T, а хвіст списком TList.

4.5 Додавання елемента в кінець списку

Структура Append

Вхідні дані: тип або список типів T, список типів TList

Результат: внутрішній тип Result

Якщо тип T є типом EmptyList і список типів TList є порожнім, то клас Result є порожнім списком типів.

Інакше, якщо параметр T є окремим типом (не списком типів) і список типів TList є порожнім, то клас Result є списком типів, що складається з єдиного типу T.

Інакше, якщо параметр T є списком типів і список типів TList є порожнім, то клас Result є списком типів T.

Інакше клас Result є списком типів, голова якого є головою списку TList, а хвіст результатом застосування алгоритму Append до хвоста списку TList з параметром T.

4.6 Видалення першого входження елемента зі списку

Структура Erase

Вхідні дані: тип T, список типів TList

Результат: внутрішній тип Result

Якщо список TList є порожнім, то клас Result є порожнім списком типів.

Інакше, якщо тип T співпадає з типом, який є головою списку TList, то клас Result є хвостом списку TList.

Інакше клас Result є списком типів, голова якого є головою списку TList, а хвіст результатом застосування алгоритму Erase до хвоста списку TList з параметром T.

4.7 Видалення всіх входжень елемента зі списку

Структура EraseAll

Вхідні дані: тип T, список типів TList

Результат: внутрішній тип Result

Якщо список TList є порожнім, то клас Result є порожнім списком типів.

Інакше, якщо тип T співпадає з типом, який є головою списку TList, то клас Result є результатом застосування алгоритму EraseAll до хвоста списку TList з параметром T.

Інакше клас Result є списком типів, голова якого є головою списку TList, а хвіст результатом застосування алгоритму EraseAll до хвоста списку TList з параметром T.

4.8 Видалення дублікатів зі списку

Структура NoDublicates

Вхідні дані: список типів TList

Результат: внутрішній тип Result

Якщо список TList є порожнім, то клас Result є порожнім списком типів.

Інакше застосувати алгоритм NoDublicates до хвоста списку TList, отримавши список L1;

застосувати алгоритм Erase до списку L1 з параметром голова списку TList, отримавши список L2;

клас Result є списком типів, голова якого є головою списку TList, а хвіст списком L2.

4.9 Заміна першого входження елемента списку

Структура Replace

Вхідні дані: тип T (підлягає заміні), тип U (заміна), список типів TList

Результат: внутрішній тип Result

Якщо список TList є порожнім, то клас Result є порожнім списком типів.

Інакше, якщо тип T співпадає з типом, який є головою списку TList, то клас Result є списком типів, голова якого є типом U, а хвіст хвостом списку TList.

Інакше клас Result є списком типів, голова якого є головою списку TList, а хвіст результатом застосування алгоритму Replace до хвоста списку TList з параметрами T і U.

4.10 Заміна всіх входжень елемента списку

Структура ReplaceAll

Вхідні дані: тип T (підлягає заміні), тип U (заміна), список типів TList

Результат: внутрішній тип Result

Якщо список TList є порожнім, то клас Result є порожнім списком типів.

Інакше, якщо тип T співпадає з типом, який є головою списку TList, то клас Result є списком типів, голова якого є типом U, а хвіст результатом застосування алгоритму ReplaceAll до хвоста списку TList з параметрами T і U.

Інакше клас Result є списком типів, голова якого є головою списку TList, а хвіст результатом застосування алгоритму ReplaceAll до хвоста списку TList з параметрами T і U.

4.11 Визначення найглибше вкладеного типу, похідного від заданого

Структура MostDelived

Вхідні дані: тип T, список типів TList

Результат: внутрішній тип Result

Якщо список TList порожній, то клас Result є типом T.

Інакше застосувати алгоритм MostDelived до хвоста списку TList і типу T, отримати тип Candidate.

Якщо тип, що є головою списку TList, є похідним від типу Candidate, то Result є головою списку TList.

Інакше Result є типом Candidate.

У реалізації цього алгоритму використовується макрос SUPERSUBCLASS, який приймає у якості параметрів два типи і під час компіляції повертає true, якщо другий тип є похідним від першого, і false у іншому випадку.

Також використовується шаблонний клас Select, який приймає булеву константу і два типи і повертає перший тип, якщо константа дорівнює true, і другий тип, якщо константа дорівнює false.

4.12 Переміщення найглибше вкладених похідних типів у голову списку

Структура DerivedToFront

Вхідні дані: список типів TList

Результат: внутрішній тип Result

Якщо список TList є порожнім, то клас Result є порожнім списком типів.

Інакше знайти (застосувавши алгоритм MostDelived) у хвості списку TList клас найнижчого рівня, похідний від класу, який є головою списку TList, зберегти його у тимчасовій змінній TheMostDerived.

Замінити клас TheMostDerived у хвості списку TList класом, який є головою списку TList, отримавши у результаті список L.

Застосувати алгоритм DerivedToFront до списку L, отримати у результаті список L1.

Result є списком типів, головою якого є TheMostDerived, а хвостом список L1.

Отже, реалізовано методи маніпулювання списками типів:

а) обчислення довжини списку;
б) індексований доступ до елементів списку;
в) пошук елемента списку;
г) додавання елемента списку;
д) видалення елемента списку;
е) заміна елемента списку;
ж) видалення дублікатів зі списку;
з) визначення найглибше вкладеного типу, похідного від заданого;
и) переміщення найглибше вкладених похідних типів у голову списку.

У реалізованих методах маніпулювання списками типів використано часткову і повну спеціалізацію шаблонів для організації рекурсії, параметри шаблонів класів є параметрами метафункцій, значення метафункцій зберігаються у статичних змінних, визначеннях типів класу чи псевдонімах типів.

5 Приклади різних аспектів застосування списків типів

5.1 Список типів з індексованим доступом до елементів (інша реалізація)

Можна реалізувати список типів з використанням варіативного шаблона і з метафункцією звернення до елемента за індексом іншим способом, без внутрішніх типів Head і Tail. Тоді реалізація списку типів виглядатиме так:

```
template <class... Types>
class type_list {};
```

Реалізація класу extract, який містить структуру extract_impl – метафункцію звернення до елемента за індексом:

```
template <std::size_t idx, class... Types>
class extract
```

```
{
    static_assert( idx < sizeof...( Types ), "index out of bounds" );

    template <std::size_t i, std::size_t n, class... Rest>
    struct extract_impl;

    template <std::size_t i, std::size_t n, class T, class... Rest>
    struct extract_impl<i, n, T, Rest...>
    {
        using type = typename extract_impl<i + 1, n, Rest...>::type;
    };

    template <std::size_t n, class T, class... Rest>
    struct extract_impl<n, n, T, Rest...>
    {
        using type = T;
    };

public:
    using type = typename extract_impl<0, idx, Types...>::type;
};
```

Клас extract

Вхідні дані: індекс idx, пакет параметрів Types

Результат: внутрішній тип type

Спочатку, на етапі компіляції, як і всі інші обчислення, за допомогою static_assert відбувається перевірка, чи індекс менший за кількість параметрів у пакеті (яка отримується за допомогою оператора sizeof); викликається помилка компіляції при невиконанні цієї умови. Далі йде реалізація структури extract_impl і у розділі public внутрішньому типу type присвоюється результат виконання цієї структури з параметрами нуль, idx, Types.

Структура extract_impl

Вхідні дані: індекс i, індекс n, пакет параметрів Rest (у якому є принаймні перший параметр – тип T)

Результат: внутрішній тип type

Загальне визначення структури рекурсивно викликає цю ж структуру для індексу i, збільшеного на одиницю, індексу n, пакета параметрів без першого Rest. Спеціалізація структури відповідає випадку, коли індекс i дорівнює індексу

n, у ній внутрішній тип type визначається як тип T, що є першим у пакеті параметрів.

При виклику структури з класу extract індекс і змінюється від нуля до n.

Клас extract приймає пакет параметрів. Для зручності можна реалізувати структуру-обгортку для нього, яка прийматиме у якості параметрів індекс idx і клас TypeList і присвоюватиме своєму внутрішньому типу type результат виконання класу extract для idx і пакета параметрів класу TypeList. Реалізація цієї структури виглядає таким чином:

```
template <std::size_t idx, class TypeList>
struct type_list_extract;

template <std::size_t idx, template <class...> class TypeList,
class... Types>
struct type_list_extract<idx, TypeList<Types...>>
{
    using type = typename extract<idx, Types...>::type;
};
```

Використання структури:

```
int main()
{
  using list_t = type_list<char, bool, void>;

  static_assert(std::is_same<char,
  typename type_list_extract_t<0, list_t>>:: type, "!");

  static_assert(std::is_same<bool,
  typename type_list_extract_t<1, list_t>>:: type, "!");

  static_assert(std::is_same<void,
  typename type_list_extract_t<2, list_t>>:: type, "!");

  //type_list_extract_t<3, list_t>; // static_assert fails: index
out of bounds
}
```

5.2 Поділ списку типів залежно від предиката

Іноді може бути потрібно поділити список типів залежно від предиката, тобто у один список типів додати типи, для яких предикат повертає true, у інший

список типи, для яких предикат повертає false. Наприклад, можна розглянути предикат std::is_floating_point, який перевіряє, чи є тип типом числа з плаваючою точкою: float, double, long double.

Реалізація списку типів для цього прикладу:

```
template <typename... T>
struct TypeList {};
```

Метафункція, яка ділить список типів залежно від предиката:

```
template<bool, template<typename> class, class... Vs>
auto FilterImpl(TypeList<>, TypeList<Vs...> v ) { return v; }

template
<bool Include, template<typename> class P, class T, class... Ts, class... Vs>
auto FilterImpl(TypeList<T,Ts...>, TypeList<Vs...>) { return FilterImpl<Include,P>(
  TypeList<Ts...>{},
  std::conditional_t<Include == P<T>::value, TypeList<T,Vs...>, TypeList<Vs...>>{}
  ); }

template<template<typename> class PREDICATE, typename TYPELIST>
struct SplitTypeList
{
  using predicate_is_true_typeList_type = decltype(FilterImpl<true,PREDICATE>(
TYPELIST{}, TypeList<>{}));
  using predicate_is_false_typeList_type = decltype(FilterImpl<false,PREDICATE>(
TYPELIST{}, TypeList<>{}));
};
```

Варто зазначити, що у цій реалізації використано механізм повернення значень метафункцій, який відрізняється від розглянутого раніше. Значення метафункцій не зберігається, як внутрішній оголошений псевдонім типу, і звернення до нього не відбувається через оператор «::», як у розглянутих раніше прикладах. Значення функції FilterImpl повертається через оператор return; тип значення, яке повертає функція, визначається автоматично завдяки ключовому слову auto перед іменем функції у її визначенні. Звернення до повернутого значення функції (у структурі SplitTypeList) відбувається через ключове слово decltype, яке дозволяє отримати тип значення.

Функція FilterImpl

Вхідні дані: булеве значення Include, шаблонний клас P (предикат), два списки типів

Результат: список типів v

Перша спеціалізація функції відповідає випадку, коли перший список типів з параметрів порожній, повертає другий список типів з параметрів. Друга спеціалізація відповідає випадку, коли перший список типів з параметрів містить хоча б один тип T. Вона рекурсивно викликає функцію FilterImpl для значень Include, P, хвіст першого списку типів і список типів, який є результатом виконання утиліти std::conditional_t. Утиліта приймає булеве значення Include == P<T>::value, якщо воно істинне, повертає другий список типів з доданим у голову списку типом T, якщо хибне – просто другий список типів. Тобто, якщо Include істинне, у список-результат додаються типи, для яких предикат повертає true, якщо Include хибне – типи, для яких предикат повертає false.

Структура SplitTypeList

Вхідні дані: шаблонний клас PREDICATE (предикат), список типів TYPELIST

Результат: внутрішні типи predicate_is_true_typeList_type, predicate_is_false_typeList_type

У структурі псевдонім predicate_is_true_typeList_type присвоюється результату виклику функції FilterImpl з параметрами true, PREDICATE, TYPELIST, порожній список типів; псевдонім predicate_is_false_typeList_type присвоюється результату виклику функції FilterImpl з параметрами false, PREDICATE, TYPELIST, порожній список типів. Таким чином predicate_is_true_typeList_type – це список типів, для яких предикат повертає true, а predicate_is_false_typeList_type – це список типів, для яких предикат повертає false.

Використання метафункції:

```
int main()
{
  using typeList = TypeList<int, double, float, A, int>;

  using splited_typeList = SplitTypeList<std::is_floating_point,
typeList>;

```

```
  using float_typeList =
splited_typeList::predicate_is_true_typeList_type;
  using other_typeList =
splited_typeList::predicate_is_false_typeList_type;

  static_assert(std::is_same< TypeList<double, float>,
float_typeList>:: type, "!");

  static_assert(std::is_same< TypeList<int, main()::A, int>,
other_typeList>:: type, "!");
}
```

Отже, розглянуто приклади різних аспектів застосування списків типів. По-перше, проаналізовано ще одну реалізацію списку типів з індексованим доступом до елементів: з використанням варіативного шаблона, без внутрішніх типів Head і Tail. У реалізації використано часткову і повну спеціалізацію шаблонів для організації рекурсії, параметри шаблонів класів є параметрами метафункцій, значення метафункцій зберігаються у статичних змінних, визначеннях типів класу чи псевдонімах типів. По-друге, оглянуто поділ списку типів залежно від предиката. У реалізації також використана рекурсія, проте значення метафункцій повертається через оператор return; тип значення, яке повертає функція, визначається автоматично завдяки ключовому слову auto; звернення до повернутого значення функції відбувається через ключове слово decltype.

6 Приклад застосування довільних списків типів

У якості прикладу застосування реалізовано варіативний шаблон для виведення значень з довільного списку типів за умови, що кожен з типів допускає оператор <<.

Реалізовано параметризований клас Printer, який приймає у якості параметра довільний список типів. Клас містить статичний метод print, який приймає значення певного типу і виводить його або виводить повідомлення про відсутність типу прийнятого значення у списку типів.

Реалізація класу Printer використовує часткову шаблонну спеціалізацію, що дозволяє розділяти порожній і непорожній список типів. Друга спеціалізація

є повною і відповідає класу TypeList<EmptyList>, у ній метод print виводить повідомлення про відсутність типу прийнятого значення у списку типів. Перша спеціалізація, часткова, відповідає будь-якому класу TypeList<Head, Tail...>. У першій спеціалізації міститься:

а) повна спеціалізація методу print, яка відповідає голові списку типів і виводить значення;
б) часткова спеціалізація методу print, яка рекурсивно викликає метод print для класу Printer з хвостом списку типів у якості параметра.

Отже, розглянуто реалізацію створеного програмного продукту – варіативного шаблона для виведення значень з довільного списку типів за умови, що кожен з типів допускає оператор <<. У реалізації використано часткову і повну спеціалізацію шаблонів для організації рекурсії, параметри шаблонів класів є параметрами метафункцій, значення метафункцій зберігаються у статичних змінних, визначеннях типів класу чи псевдонімах типів.

Висновки

Метапрограмування – це техніка програмування, в якій програми (їх називають метапрограми) розглядають інші програми як свої дані, тобто можуть читати, генерувати, аналізувати або перетворювати їх і навіть модифікувати себе під час роботи. Метапрограмування використовується для переміщення обчислень з часу виконання на час компіляції.

Шаблонне метапрограмування – це техніка метапрограмування, у якій шаблони використовуються компілятором для генерування тимчасового вихідного коду, розрахунки відбуваються на етапі компіляції, код маніпулює типами. Основний інструмент – метафункції – шаблони класів; параметри шаблонів класів є параметрами метафункцій; значення метафункцій зберігаються у статичних змінних, визначеннях типів класу чи псевдонімах типів. При підстановці значень параметрів у шаблон класу отримується клас. Спеціалізації класів: часткові – особливі реалізації класів для конкретних значень частини параметрів; повні – для конкретних значень усіх параметрів. Шаблонне метапрограмування є функціональним: метазмінні, типи і псевдоніми типів є незмінними, відсутній оператор присвоєння значень змінним, змінні можливі тільки в якості параметрів метафункцій, єдиний можливий спосіб організації циклів – рекурсія, для її організації використовуються спеціалізації класів.

Списки типів дають можливість писати один і той же код для різних типів і маніпулювати цим набором типів. Списки типів не містять значень, не призначені для створення об’єктів, а лише для надання інформації про типи. Реалізація: клас Typelist містить два внутрішні типи: Head – будь-який тип, Tail – клас Typelist. Реалізація списків типів із бібліотеки Loki, яка розглянута у книзі Александреску, для зручного синтаксису створення списків типів потребує макросів, які перетворюють рекурсію на просте перерахування типів. Макроси визначено для списків типів, які містять п’ятдесят або менше елементів, але користувач бібліотеки може визначити макроси для більшої кількості елементів

завдяки тому, що кожен макрос використовує попередній. Варіативні шаблони – шаблони функції або класу зі змінною кількістю аргументів, приймають у якості аргументів пакет параметрів – нуль або більше типів. Застосування списків типів і варіативних шаблонів належать до шаблонного метапрограмування, тому всі обчислення відбуваються на етапі компіляції, а не під час виконання програми. Якщо реалізувати списки типів з використанням варіативних шаблонів, то при створенні списку типи задаються простим перерахуванням, отже, визначати макроси не потрібно – це найбільша відмінність від реалізації з книги Александреску. Переваги використання варіативних шаблонів у реалізації списків типів:

а) не потрібно загромаджувати заголовний файл визначенням макросів, витрачати час на їхнє визначення, немає можливості помилитись у визначенні макросів;
б) користувачеві не потрібно турбуватись про використання макросу, відповідного кількості елементів у списку (TYPELIST_4 для чотирьох елементів і т. д.), змінювати макрос при зміні кількості елементів;
в) користувач може створювати списки типів з великою кількістю елементів;
г) при створенні списку типи перераховуються не в круглих, а у кутових дужках, які вказують користувачеві на те, що це саме параметри шаблона (логічніший синтаксис).

Реалізовано списки типів з використанням варіативних шаблонів та методи маніпулювання ними:

а) обчислення довжини списку;
б) індексований доступ до елементів списку;
в) пошук елемента списку;
г) додавання елемента списку;

д) видалення елемента списку;

е) заміна елемента списку;

ж) видалення дублікатів зі списку;

з) визначення найглибше вкладеного типу, похідного від заданого;

и) переміщення найглибше вкладених похідних типів у голову списку.

У реалізованих методах маніпулювання списками типів використано часткову і повну спеціалізацію шаблонів для організації рекурсії, параметри шаблонів класів є параметрами метафункцій, значення метафункцій зберігаються у статичних змінних, визначеннях типів класу чи псевдонімах типів.

Розглянуто приклади різних аспектів застосування списків типів. По-перше, проаналізовано ще одну реалізацію списку типів з індексованим доступом до елементів: з використанням варіативного шаблона, без внутрішніх типів Head і Tail. У реалізації використано часткову і повну спеціалізацію шаблонів для організації рекурсії, параметри шаблонів класів є параметрами метафункцій, значення метафункцій зберігаються у статичних змінних, визначеннях типів класу чи псевдонімах типів. По-друге, оглянуто поділ списку типів залежно від предиката. У реалізації також використана рекурсія, проте значення метафункцій повертається через оператор return; тип значення, яке повертає функція, визначається автоматично завдяки ключовому слову auto; звернення до повернутого значення функції відбувається через ключове слово decltype.

Створено програмний продукт – варіативний шаблон для виведення значень з довільного списку типів за умови, що кожен з типів допускає оператор <<. У його реалізації використано часткову і повну спеціалізацію шаблонів для організації рекурсії, параметри шаблонів класів є параметрами метафункцій, значення метафункцій зберігаються у статичних змінних, визначеннях типів класу чи псевдонімах типів.

Перспективи покращення досягнутих результатів:

а) реалізувати списки типів з використанням варіативних шаблонів без внутрішніх типів Head і Tail та методи маніпулювання ними;

б) реалізувати поділ списку типів залежно від предиката;

в) реалізувати генерацію класу на основі списку типів (розподілених і лінійних ієрархій).

Список літератури

1. Metaprogramming [Електронний ресурс] : Матеріал з Вікіпедії — вільної енциклопедії : Версія 948480681, збережена о 07:38 UTC 1 квітня 2020, Режим доступу: https://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=948480681

2. Метапрограммирование [Електронний ресурс] : Матеріал з Вікіпедії — вільної енциклопедії : Версія 101201617, збережена о 10:43 UTC 23 липня 2019, Режим доступу: https://ru.wikipedia.org/?oldid=101201617

3. Nilsdeppe.com [Електронний ресурс] : [Інтернет-портал]. – Режим доступу: https://nilsdeppe.com/posts/tmpl-part1. – Назва з екрану.

4. Краснов М.М. Метапрограммирование шаблонов C++ в задачах математической физики. М.: ИПМ им.М.В.Келдыша, 2017. 84 с.

doi:10.20948/mono-2017-krasnov

URL: https://www.keldysh.ru/e-biblio/krasnov

5. Habr.com [Електронний ресурс] : [Інтернет-портал]. – Режим доступу: https://habr.com/ru/post/235831/. – Назва з екрану.

6. www.codenet.ru [Електронний ресурс] : [Інтернет-портал]. – Режим доступу: http://www.codenet.ru/progr/cpp/Macros.php – Назва з екрану.

7. Variadic template [Електронний ресурс] : Матеріал з Вікіпедії — вільної енциклопедії : Версія 27610162, збережена о 12:17 UTC 3 квітня 2020, Режим доступу: https://uk.wikipedia.org/w/index.php?title=Variadic_template&oldid=27610162

8. Habr.com [Електронний ресурс] : [Інтернет-портал]. – Режим доступу: https://habr.com/ru/post/228031/ – Назва з екрану.

9. Александреску, Андрей Современное проектирование на C++. Серия C++ In-Depth, т. 3.: Пер. с англ. – М. : Издательский дом «Вильямс», 2002. – 336 с. : ил. – Парал. тит. англ.

10. Codereview.stackexchange.com [Електронний ресурс] : [Інтернет-портал]. – Режим доступу:

https://codereview.stackexchange.com/questions/127925/typelist-with-extractor – Назва з екрану.

11. Stackoverflow.com [Електронний ресурс] : [Інтернет-портал]. – Режим доступу: https://stackoverflow.com/questions/46852136/most-elegant-way-to-split-a-c-typelist – Назва з екрану.