Метапрограмування C++.11-17

Метапрограмування - це техніка програмування, в якій комп'ютерні програми мають можливість розглядати інші програми як свої дані.
Це означає, що програма може бути розроблена для читання, генерування, аналізу або перетворення інших програм 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 дозволяє програмам більшої гнучкості для ефективного виконання з новими ситуаціями без перекомпіляції. Метапрограмування можна використовувати для переміщення обчислень з часу виконання на час компіляції, для генерування коду за допомогою обчислень часу компіляції та для включення коду, що самовиправляється. Мова, якою написана метапрограма, називається *метамовою*. Мова маніпулюючих програм називається *атрибутивно орієнтованою* мовою програмування. Здатність мови програмування до власного метамовлення називається відображенням або "рефлексивністю".

Метапрограмування шаблонів *(template metaprogramming - TMP)* - це процес написання заснованих на шаблонах програм на C ++, що виконуються під час компіляції. По суті, шаблонна метапрограмма - це програма, написана на C ++, яка виповнюється всередині компілятора C + +. Коли TMP-програма завершує виконання, її результат - фрагменти коду на C ++, конкретизовані з шаблонів, - компілюється як зазвичай.

Шаблони - це особливість мови програмування C ++, яка дозволяє функціям та класам працювати з загальними типами. Це дозволяє функції або класу працювати над багатьма різними типами даних, не переписуючись для кожного. Шаблони були введені в мову програмування C ++ як засіб вираження параметризованих типів. Прикладом параметризованого типу є список, для якого неоптимано реалізовувати окремі версії для кожного типу збережених елементів. Замість цього стоїть задача передбачити єдину реалізацію списку (шаблон), яка використовує "заповнювач" типу елементів *(параметр шаблону)*, за допомогою якого компілятор може генерувати різні класи списку *(екземпляри шаблонів)*.

Існує три **типи шаблонів**:

1) шаблони функцій,
2) шаблони класів,
3) шаблони змінних (починаючи з C ++ 14).

Шаблони функцій

Шаблон функції поводиться як функція, за винятком того, що шаблон може мати аргументи багатьох різних типів. Іншими словами, шаблон функції являє собою сімейство функцій.

Формат для оголошення шаблонів функцій з параметрами типу:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

Обидва вирази мають однакове значення і поводяться абсолютно однаково. (//Остання форма була введена для уникнення плутанини, оскільки параметр типу не повинен бути класом. (Це також може бути базовий тип, наприклад, int або double.)

Це єдине визначення функції працює з багатьма типами даних. Зокрема, він працює з усіма типами даних, для яких визначено більше (ніж оператор). Використання шаблону функції економить місце у файлі вихідного коду, крім обмеження змін в одному описі функції та полегшує зчитування коду.

Приклад використання шаблону функції:

```
#include <iostream>

int main()
{
    // This will call max<int> by implicit argument deduction.
    std::cout << max(3, 7) << std::endl;

    // This will call max<double> by implicit argument deduction.
    std::cout << max(3.0, 7.0) << std::endl;

    // This depends on the compiler. Some compilers handle this by defining a
template
    // function like double max <double> ( double a, double b);, while in
some compilers
    // we need to explicitly cast it, like std::cout << max<double>(3,7.0);
    std::cout << max(3, 7.0) << std::endl;
    std::cout << max<double>(3, 7.0) << std::endl;
    return 0;
}
```

У перших двох випадках компілятор автоматично виводить аргумент шаблону `Type` аргументу як `int` , так і `double`,відповідно. У третьому випадку автоматичне виведення max (3, 7.0) буде невдалим, оскільки тип параметрів повинен загалом точно

відповідати аргументам шаблону. Тому ми явно створюємо подвійну версію з max <double> ().

Шаблони класів

Шаблон класу забезпечує специфікацію для генерації класів на основі параметрів. Шаблони класів зазвичай використовуються для реалізації контейнерів. Шаблон класу створюється миттєво, передаючи йому заданий набір типів як аргументи шаблону. Стандартна бібліотека C ++ містить багато шаблонів класів, зокрема контейнери, адаптовані з бібліотеки стандартних шаблонів, такі як `vector`.

Шаблони змінних

У версіях до C ++.14, шаблони могли бути шаблонами лише функцій, класів або псевдонімів типів. Від C ++ 14 з'явилася можливість створювати шаблони змінних. Шаблони можуть також використовуватися для змінних. Наприклад:

```
template<typename T> constexpr T pi = T(3.14159265358979323846264338383L);
```

Сьогодні C ++ - шаблони використовуються абсолютно несподіваним для їх творців чином. Шаблонне програмування включає такі методики, як узагальнене програмування, обчислення під час компіляції, бібліотеки шаблонних виразів, шаблонне метапрограмування і цей список можна продовжувати.
В принципі, розрахунок простих чисел, ґрунтуються на тому факті, що конкретизація шаблонів є рекурсивним процесом. Коли компілятор конкретизує шаблон, він може виявити, що для конкретизації цього шаблону, можливо, буде потрібно конкретизація іншого шаблону. Потім він конкретизує інший шаблон, якому може знадобитися конкретизація ще одного шаблону і так далі. Багато з методик шаблонного метапрограмування використовують рекурсивну природу процесу конкретизації шаблонів для виконання рекурсивних обчислень.

Приклад обчислень під час компіляції - факторіал

Підрахуємо факторіал під час етапу компіляції. Факторіал n (математична нотація якого n!) Є продуктом всього ряду від 1 до n, факторіал 0 дорівнює 1. Зазвичай, без використання шаблонів, ви повинні підрахувати факторіал за допомогою рекурсивних викликів функцій.

```
int factorial (int n)
{ return (n==0) ? 1: n*factorial(n-1); }
```

Ця функція викликає себе рекурсивно до тих пір, поки n не зменшиться до 0. Рекурсивні виклики функції є дорогими. Компілятор, ймовірно, відмовиться від вбудованих рекурсивних викликів, і в підсумку все завершиться цілою низкою викликів функцій. Замість цього ми можемо вчинити так - робити обчислення під час компіляції. Для цього необхідно замінити рекурсивний виклик функції рекурсивної конкретизацією шаблону. Замість реалізації функції з ім'ям factorial ми реалізуємо клас з ім'ям factorial. Або точніше, це буде шаблонний клас з ім'ям factorial.

```
template <int n>
struct factorial {
  enum { ret = factorial<n-1>::ret * n };
};
```

Цей шаблон не має ні даних, ні функцій-членів, він просто визначає безіменний перелічувальний тип з єдиним значенням. Для того щоб обчислити значення цього перерахування, компілятор повинен конкретизувати ще одну версію шаблону `factorial`, а саме версію для n-1, і це те, з чого починається рекурсивна конкретизація примірників шаблонів.

Слід зауважити, що аргумент шаблону `factorial` - незвичайний: це не тип в чистому вигляді, а константне значення типу `int` . Як правило, шаблони мають аргументи на кшталт, шаблонного `<class T> class X{...}`, де `T` є "міткою" конкретного типу. Ця "мітка" пізніше буде замінена конкретним типом, коли з шаблону буде генеруватися клас, наприклад, X `<int>`. У вище наведеному прикладі ми маємо шаблонний аргумент у вигляді константного значення типу `int`. Це означає, що цей шаблон повинен конкретизуватися константним значенням типу і`int`. За допомогою цього шаблону користувач має можливість обчислити факторіал n наступним чином:

```
cout << factorial<4>::ret << endl;
```

Компілятор буде рекурсивно конкретизувати `factorial <4>`, `factorial<3>`. При використанні методики шаблонів кінець рекурсії визначається за допомогою шаблону спеціалізації. У вище наведеному прикладі спеціалізація повинна визначатися для випадку n = 0:

```
template <>
struct factorial<0> {
  enum { ret = 1 };
};
```

Не складно помітити, з фрагмента коду вище, тут рекурсія буде припинятися, тому що значення перерахування `ret` більше не залежить від подальшої конкретизації шаблону `factorial`. У прикладі ми надається спеціальна реалізація шаблону `factorial` для випадку, коли шаблонний аргумент дорівнює 0. І ця спеціалізація буде завершувати рекурсію.

Отже, вираз `factorial <4>::ret` всього лише зведеться до числа 24 (яке є результатом 4!). На стадії виконання в виконуваному файлі не можна виявити ніяких обчислень. Замість цього в усіх місцях вихідного коду, де зустрічається `factorial <4>::ret` буде константне значення 24.

Ще одним прикладом підрахунків під час компіляції є квадратний корінь.

Шаблон `root`:

```
template <size_t N, size_t Low=1, size_t Upp=N>
struct Root {
  static const size_t ret =
                  Root<N,(down?Low:mean+1),(down?mean:Upp)>::ret;
  static const size_t mean = (Low+Upp)/2;
  static const bool   down = ((mean*mean)>=N);
};
```

Обчислення на етапі компіляції часто є рекурсивними процесами, які використовують переваги рекурсивної природи процесу конкретизації шаблонів. Те, що було функцією, що виконувало обчислення на етапі виконання, тепер є шаблоном, що виконує обчислення на етапі компіляції. Те, що було аргументом функції під час виконання обчислень, є "нетиповим" шаблонним аргументом під час компіляції. Те, що було поверненням значенням функції під час виконання обчислень, тепер є константою, вкладеної в шаблон класу під час компіляції. І умова закінчення рекурсії на етапі виконання тепер є шаблонною спеціалізацією на етапі компіляції.

Шаблони виразів

До сих пір ми виробляли обчислення значень на етапі компіляції. Тепер розрахуємо більш складні вирази під час компіляції. На першому кроці, реалізуємо версію розрахунку скалярного добутку векторів на етапі компіляції. Скалярний добуток двох

векторів дорівнює сумі добутків відповідних елементів. Мета полягає в тому, щоб створити шаблони виразів для розрахунку скалярного добутку векторів довільної розмірності. Скалярний добуток - є доволі примітивним прикладом використання шаблонів, але методи, застосовані для обрахунку скалярного добутку векторів, були узагальнені для арифметичних операцій над багатовимірними матрицями. Природно, виграш в продуктивності часу виконання завдяки обчислень на етапі компіляції буде значно істотнішим для операцій над матрицями, ніж для скалярного добутку векторів. Методи, що застосовуються в обох випадках, засновані на одних і тих самих принципах.

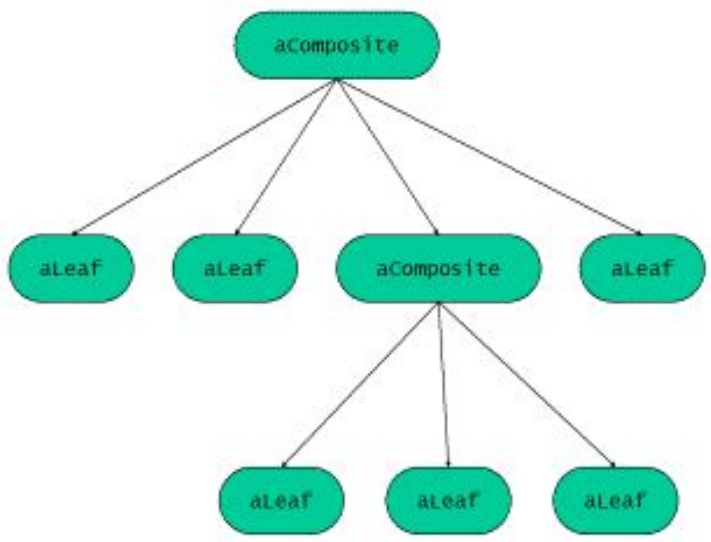
Тож, скалярний добуток векторів можна розглядати як окремий випадок паттерна Composite (компоновщик).

Паттерн Composite дає можливість представити співвідношення "частина-ціле", за допомогою якого користувач отримує можливість ігнорувати відмінності між окремими об'єктами та композицією об'єктів. Ключовими елементами патерну є `leaf` (лист, примітивний об'єкт) і `composite` (композиція, компонований об’єкт).

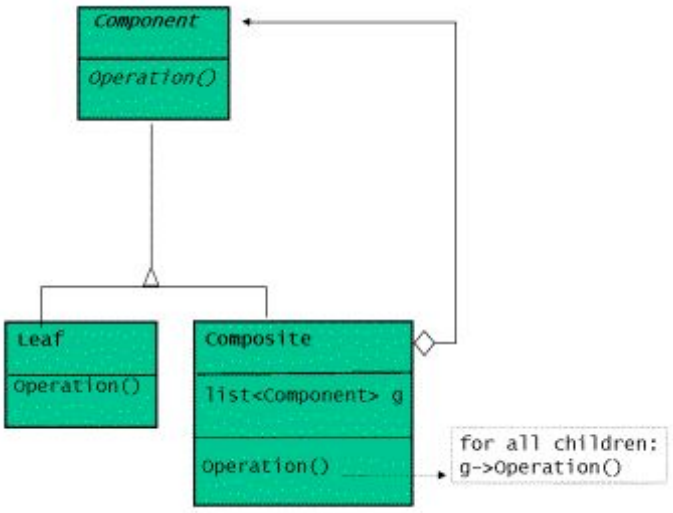
`leaf` визначає поведінку примітивних об'єктів в композиції.

`composite` визначає поведінку компонентів, що складаються з листків.

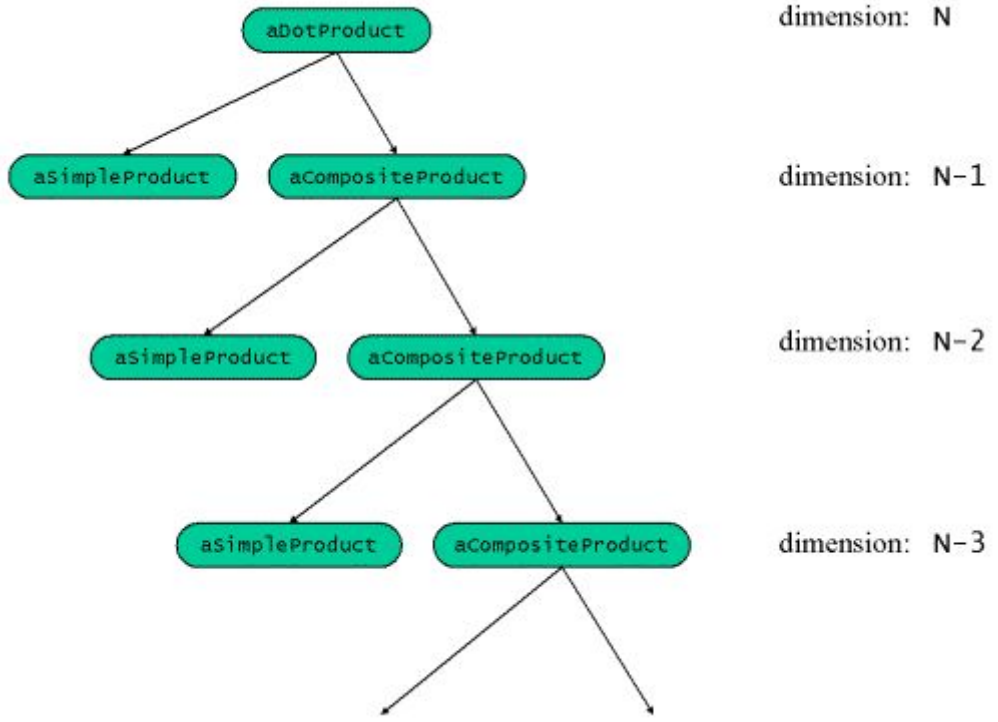
Прикладами `composite` є синтаксичні дерева, агрегати, рекурсивні структури і рекурсивні алгоритми. Ось приклад типової структури `composite:`



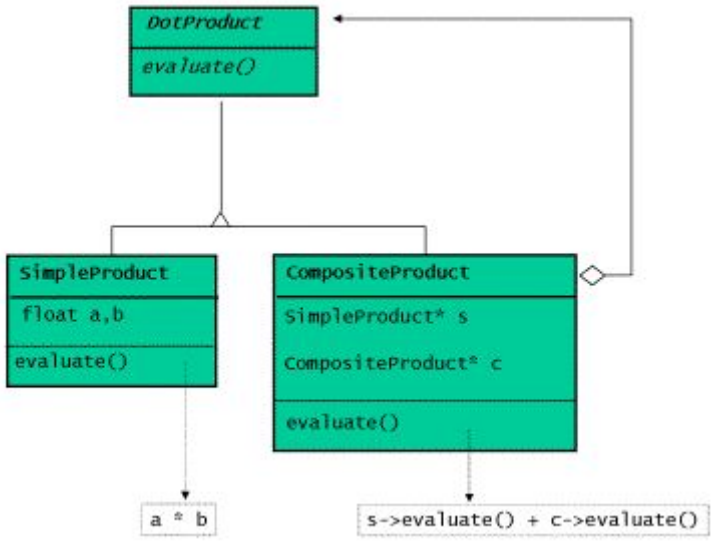
Також, існує об'єктно-орієнтовану реалізація паттерна `composite` з абстрактним базовим класом, який визначає операцію, загальну для leaf і composite і два похідних класу, які представляють `leaf` і `composite` відповідно.



Скалярний добуток векторів може розглядатися як окремий випадок паттерна `composite`. Скалярний добуток можна розділити на `leaf` (а саме скалярний добуток двох векторів розмірності 1) і `composite` (а саме скалярний добуток двох векторів розмірності N-1).



Очевидно, це вироджена форма композиту, оскільки кожен складений об'єкт містить рівно з один листок та один композит. Використовуючи запропоновану об'єктно-орієнтовану реалізацію, можна представити скалярний добуток за допомогою базового і двох похідних класів, як показано на малюнку нижче:



Реалізація прямолінійна і показана в лістингу 1-3. У лістингу 4 показана допоміжна функція, яка полегшує використання цих класів, а лістинг 5 в підсумку демонструє, яким чином можна розрахувати скалярний добуток двох векторів.

Лістинг 1: Базовий клас

```
template <class T>
class DotProduct  {
   public:
      virtual ~DotProduct () {}
      virtual T eval() = 0;
};
```

Лістинг 2: `composite`

```
template <class T>
class CompositeDotProduct  : public DotProduct<T> {
public:
  CompositeDotProduct (T* a, T* b, size_t dim)
  :s(new SimpleDotProduct<T>(a,b))
  ,c((dim==1)?0:new CompositeDotProduct<T>(a+1,b+1,dim-1))
  {}
  virtual ~CompositeDotProduct () { delete c; delete s; }
  virtual T eval()
  { return (s->eval() + ((c)?c->eval():0)); }
protected:
  SimpleDotProduct<T>* s;
  CompositeDotProduct<T> * c;
};
```

Лістинг 3: `leaf`

```
template <class T>
class SimpleDotProduct : public DotProduct<T> {
   public:
      SimpleDotProduct (T* a, T* b) : v1(a), v2(b) {}
      virtual T eval() { return (*v1)*(*v2); }
   private:
      T* v1; T* v2;
};
```

Лістинг 4: Допоміжна функція

```
template <class T> T dot(T* a, T* b, size_t dim)
{ return (dim==1)
         ? SimpleDotProduct<T>(a,b).eval()
         : CompositeDotProduct<T>(a,b,dim).eval();
}
```

Лістинг 5: Використання реалізації скалярного добутку

```
int a[4] = {1,100,0,-1};
int b[4] = {2,2,2,2};
cout << dot(a,b,4);
```

Вищенаведений спосіб не найефективніший спосіб обчислення скалярного добутку двох векторів. Ми можемо істотно спростити реалізацію, усуваючи уявлення складеного і примітивного об'єктів як членів даних в похідних класах. Замість передачі векторів у складений і примітивний об'єкти, під час конструювання та їх збереження для подальшого аналізу, можна передавати інформацію безпосередньо в обчислювальну функцію.

Конструктор і обчислювальна функція

```
SimpleDotProduct<T>::SimpleDotProduct (T* a, T* b) :v1(a), v2(b) {}
virtual T SimpleDotProduct<T>::eval() { return (*v1)*(*v2); }
```

буде замінена функцією, яка приймає аргументи:

```
T SimpleDotProduct::eval(T* a, T* b, size_t dim) { return (*a)*(*b); }
```

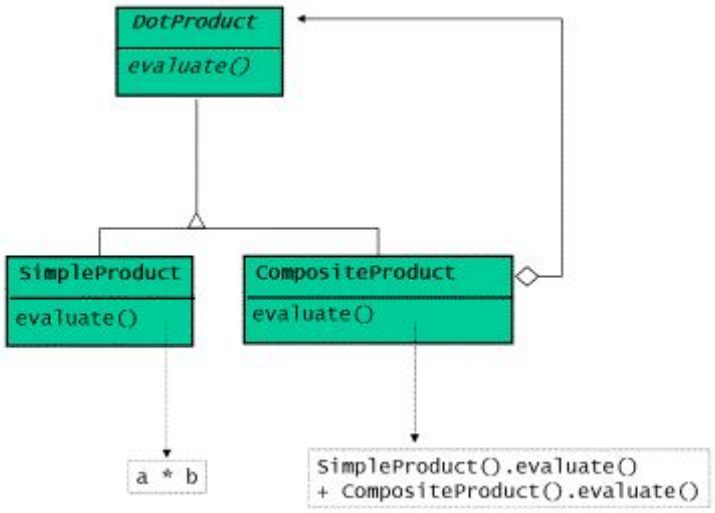
Спрощена реалізація похідного класу показана в лістингу 6; базовий клас залишиться незмінним, як показано в лістингу 1, допоміжна функція повинна бути скоригована.

Лістинг 4: Спрощена об’єктно - орієнтована реалізація скалярного добутку векторів

```
template <class T>
class CompositeDotProduct  : public DotProduct <T> {
public:
  virtual T eval(T* a, T* b, size_t dim)
  { return  SimpleDotProduct<T>().eval(a,b,dim)
    + ((dim==1) ? 0
      : CompositeDotProduct<T>().eval(a+1,b+1,dim-1));
  }
};

template <class T>
class SimpleDotProduct  : public DotProduct <T> {
public:
  virtual T eval(T* a, T* b, size_t dim)
  { return (*a)*(*b); }
};
```

Діаграма класів для спрощеної реалізації матиме наступний вигляд:



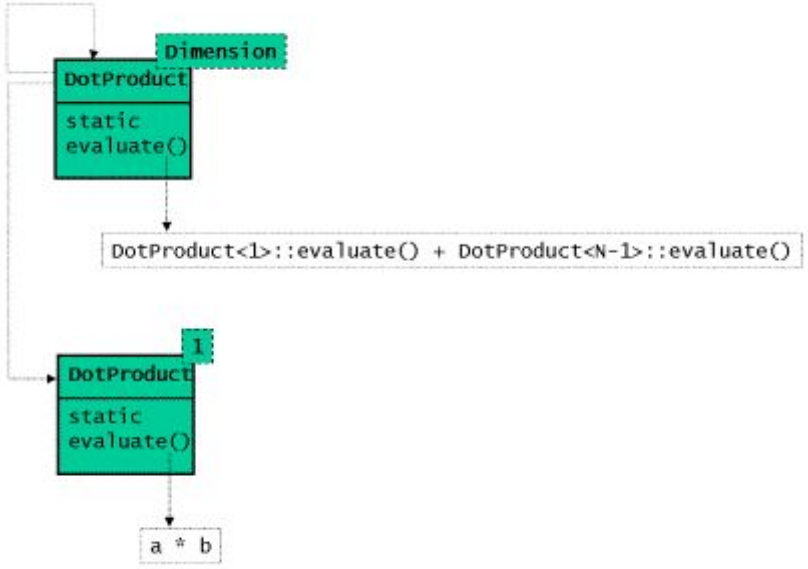
Версія скалярного добутку із підрахунком під час компіляції

Тепер скористаємося об'єктно-орієнтованим підходом, і переведемо його в рішення, яке працює під час компіляції. Два класи, що представляють `leaf` і `composite` мають загальний базовий клас, який визначає їх операції. Це добре відома об'єктно-орієнтована техніка: спільності виражається за допомогою загального базового класу. У шаблонному програмуванні спільності виражаються в термінах іменування і спільності імен. Те, що є віртуальною функцією в об'єктно-орієнтованому підході стане простою невіртуальної функцією, яка має певну назву. Два похідних класу, більше не будуть успадковувати від загального базового класу. Замість цього вони будуть автономними класами, які мають функції-члени з тими ж іменами і сумісними сигнатурами. Тобто, виключиться базовий клас.

Далі, ми реалізуємо `composite`, як шаблон класу, який використовує структурну інформацію в якості аргументів шаблону. Ця структурна інформація представляє розмірність векторів. Пам'ятайте, що ми робили для розрахунку факторіала і квадратного кореня: аргумент функції реалізації під час виконання став аргументом шаблону реалізації під час компіляції. Ми будемо робити щось подібне і тут: в об'єктно-орієнтованому підході розмірність векторів приймає в якості аргументу для функції; ми будемо передавати її в якості аргументу шаблону в реалізації на стадії компіляції. Таким чином, розмірність векторів стане "нетиповим" аргументом шаблону класу `composite`.

`leaf` буде реалізований у вигляді спеціалізації шаблону класу `composite` для розмірності N = 1. Як і раніше ми замінимо рекурсию стадії виконання рекурсією стадії

компіляції: ми замінимо рекурсивний виклик віртуальної обчислювальної функції рекурсивної шаблонної конкретизацією статичної обчислювальної функції. Ось діаграма реалізації скалярного твори векторів на етапі компіляції:



Лістинг 7: Реалізація скалярного добутку, що підраховується на етапі компіляції

```
template <size_t N, class T>
class DotProduct  {
public:
  static T eval(T* a, T* b)
  { return  DotProduct<1,T>::eval(a,b)
          + DotProduct<N-1,T>::eval(a+1,b+1);
  }
};
template <class T>
class DotProduct<1,T> {
public:
  static T eval(T* a, T* b)
  { return (*a)*(*b); }
};
```

Лістинг 8: Використання реалізації скалярного добутку

```
template <size_t N, class T>
inline T dot(T* a, T* b)
{ return DotProduct<N,T>::eval(a,b); }
int a[4] = {1,100,0,-1};
int b[4] = {2,2,2,2};
cout << dot<4>(a,b);
```

Слід звернути увагу на різницю між виразами `dot(a, b, 4)` в реалізації обчислень на етапі виконання і `dot<4> (a,b)`) в реалізації обчислень на етапі компіляції:

`dot(a, b, 4)` зводиться до обчислення `CompositeDotProduct <size_t> ().eval (a,b.4)`, що призводить до наступних рекурсивним викликам на стадії виконання:

```
SimpleDotProduct<size_t>().eval(a,b,1)
CompositeDotProduct<size_t>().eval(a+1,b+1,3)
SimpleDotProduct<size_t>().eval(a+1,b+1,1)
CompositeDotProduct<size_t>().eval(a+2,b+2,2)
SimpleDotProduct<size_t>().eval(a+2,b+2,1)
CompositeDotProduct<size_t>().eval(a+3,b+3,1)
SimpleDotProduct<size_t>().eval(a+3,b+3,1)
```

що призводить в результаті до 7 викликам віртуальних функцій.

dot <4> (a, b) з-якому другому боку, зводиться до обчислення DotProduct <4, size_t> :: eval (a, b), що запускає конкретизацію шаблонів, яка послідовно розгортається таким чином:

```
DotProduct<4,size_t>::eval(a,b)
```

призводить до

```
DotProduct<1,size_t>::eval(a,b) + DotProduct<3,size_t>::eval(a+1,b+1)
```

що призводить до підрахунку

```
(*a)*(*b) + DotProduct<1,size_t>::eval(a+1,b+1) +
  DotProduct<2,size_t>::eval(a+2,b+2)
```

що призводить до підрахунку

```
(*a)*(*b) + (*a+1)*(*b+1) + DotProduct<1,size_t>::eval(a+2,b+2) +
  DotProduct<1,size_t>::eval(a+3,b+3)
```

що призводить до підрахунку

```
(*a)*(*b) + (*a+1)*(*b+1) + (*a+2)*(*b+2) + (*a+3)*(*b+3)
```

Видимим в виконуваному коді буде тільки результуюче вираз (* a) * (* b) + (* a + 1) * (* b + 1) + (* a + 2) * (* b + 2) + (* a + 3) * (* b + 3); рекурсивна конкретизація шаблонів вже була виконана під час компіляції.

Очевидно, шаблонний підхід до вирішення проблеми є більш ефективним з точки зору продуктивності на стадії виконання за рахунок збільшення часу компіляції. Рекурсивна шаблонна конкретизація займає багато часу і не дивно, що необхідний час компіляції зростає на кілька порядків для шаблонного рішення в порівнянні з об'єктно-орієнтованим рішенням, яке компілюється швидко, натомість його робота значно уповільнюється.

Ще одне обмеження шаблонного рішення полягає в тому, що розмірність векторів повинна бути відома під час компіляції, тому що розмірність є аргументом шаблону DotProduct. На практиці це не здається таким вже великим обмеженням, так як розмірність векторів часто відома заздалегідь, а в багатьох додатках, ми просто знаємо, що будемо виконувати арифметичні операції з 3-мірними векторами, наприклад, такі, що представляють точки в просторі.

Реалізація скалярного добутку векторів може видатися не надто корисною у прикладному плані, адже така ж висока продуктивність досягається і розгортанням вираження скалярного добутку вручну. Але методи, продемонстровані тут для реалізації скалярного добутку, можуть бути узагальнені на арифметичні операції над багатовимірними матрицями. Якщо взяти такий вислів, як a * b + c, де a, b і c є матрицями 10x20. Розгорнутати його вручну дуже трудомістки процес, в той час, як компілятор може робити це автоматично і надійно.

Ще одним потужним узагальненням є інший спосіб застосування шаблонів: в кінцевому підсумку ми хочемо розраховувати вираз неодноразово, задаючи різні значення для кожного обчислення. За допомогою шаблонів ці розрахунки будуть виконані під час компіляції, що тягне нульові витрати на етапі виконання. Оцінка виразів відіграє важливу роль у обчисленні інтегралів. Наприклад, інтеграл

$$\int_1^5 \frac{x}{1+x}\,dx$$

апроксимується обчисленням виразу x / (1 + x) для n рівновіддалених точок в інтервалі [1.0,5.0]. Функція, що обчислює інтеграли для довільних арифметичних виразів, може

виглядати так само, якщо тільки нам вдасться реалізувати шаблони виразів, які можуть використовуватися багаторазово для різних значень:

```
template <class ExprT>
double integrate (ExprT e,double from,double to,size_t n)
{  double sum=0, step=(to-from)/n;
   for (double i=from+step/2; i<to; i+=step)
       sum+=e.eval(i);
   return step*sum;
}
```

ExprT в даному прикладі, так чи інакше, представляє вираз, таке як x / (1 + x).

Давайте просунемося ще один на один крок далі і розглянемо реальну складову структуру: арифметичні вирази, що складаються з унарних і бінарних арифметичних операторів, змінних і констант. Саме для цього випадку в книзі GOF є патерн Interpreter.

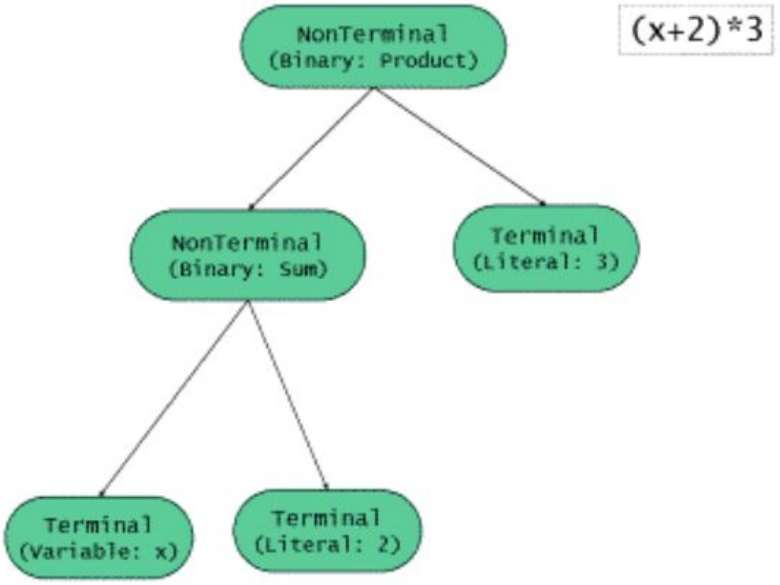
Шаблон віразів. Арифметичні вирази.

Паттерн `Interpreter` передбачає подання мови у вигляді абстрактного синтаксичного дерева і інтерпретатор, який використовує це синтаксичне дерево для інтерпретації мовних конструкцій. Це окремий випадок паттерна `Composite`. Ставлення "частина-ціле" паттерна `Composite` відповідає відношенню виразу і підвиразу в паттерні `Interpreter` (інтерпретатор).

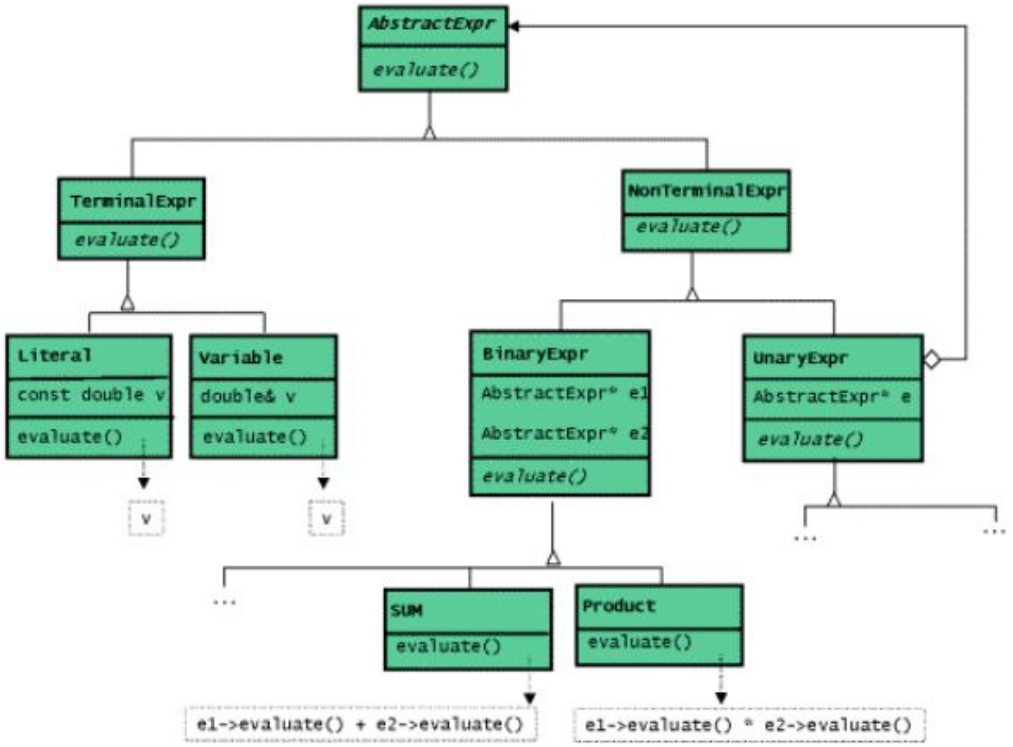
- `Leaf` є термінальним виразом.
- `Composite` є нетермінальним виразом.
- Обчислення компонентів є інтерпретацією синтаксичного дерева і його виразів.

Синтаксичне дерево представляється такими арифметичними виразами, як (a + 1) * c або `log(abs(x-N)`. Є два типи терміналів: *числові літерали* і *числові змінні.* Літерали мають константне значення, в той час як значення змінних можуть змінюватися між інтерпретаціями виразу. Нетерміналом є унарні або бінарні вираження, що складаються з одного або двох подвираженій. Вирази мають різну семантику, таку як `+, -, *, ++, -, exp, log, sqrt.`

Візьмемо конкретний приклад вираження, скажімо (x + 2) * 3. Складова структура, тобто синтаксичне дерево для цього виразу буде виглядати наступним чином:



Класичний об'єктно-орієнтований підхід для реалізації паттерна `Interpreter`, буде включати в себе наступні класи:



Відповідний вихідний код реалізації показаний в лістингу 9. Базовий клас для UnaryExpr реалізується за аналогією з класом BinaryExpr і все конкретні унарні і бінарні вираження наслідують приклад класу Sum.

Лістинг 9: Об'єктно-орієнтована реалізація інтерпретатора для арифметичних виразів

```
class AbstractExpr {
public:
   virtual double eval() const = 0;
};
class TerminalExpr : public AbstractExpr {
};
class NonTerminalExpr : public AbstractExpr {
};
class Literal : public TerminalExpr {
public:
   Literal(double v) : _val(v) {}
   double eval() const { return _val; }
private:
   const double _val;
};
class Variable : public TerminalExpr  {
public:
   Variable(double& v) : _val(v) {}
   double eval() const { return _val; }
private:
   double& _val;
};
class BinaryExpr : public NonTerminalExpr {
protected:
   BinaryExpr(const AbstractExpr* e1, const AbstractExpr* e2)
     : _expr1(e1),_expr2(e2) {}
   virtual ~BinaryExpr ()
   { delete const_cast<AbstractExpr*>(_expr1);
     delete const_cast<AbstractExpr*>(_expr2);
   }
   const AbstractExpr* _expr1;
   const AbstractExpr* _expr2;
};
class Sum : public BinaryExpr {
public:
   Sum(const AbstractExpr* e1, const AbstractExpr* e2)
     : BinExpr(e1,e2) {}
   double eval() const
   { return _expr1->eval() + _expr2->eval(); }
};
...
```

Лістинг 10 показує, як буде використовуватися інтерпретатор для обчислення виразу (x + 2) * 3.

Лістинг 10: Використання інтерпретатора для арифметичних виразів

```
void someFunction(double x)
{
  Product expr(new Sum(new Variable(x),new Literal(2)), new Literal(3));
  cout << expr.eval() << endl;
}
```

Спочатку створюється об'єкт вираження `expr`, який представляє вираз (x + 2) * 3 і потім обчислюється об'єкт вираження. (Звичайно, це вкрай неефективний спосіб

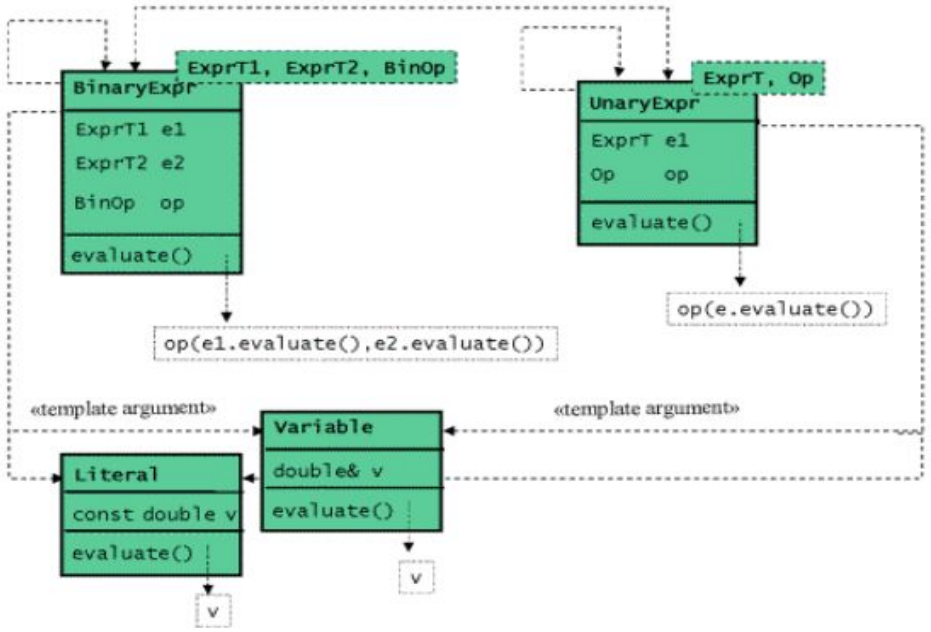
обчислення результату такого примітивного вираження, як (x + 2) * 3). Тепер перетворимо об'єктно-орієнтований підхід на вирішення на основі шаблону.

Як і раніше, у випадку скалярного добутку ми усунемо всі абстрактні базові класи, так як шаблонні рішення засновані на однаковості імен, а не успадкування. З цієї причини, ми не потребуємо базових класах. Замість подання всіх термінальних і нетермінальних виразів за допомогою автономних класів, використовуватимемо функцію оцінки з ім'ям `eval()`.

Потім ми виразимо всі нетермінальні вирази, такі як сума або добуток, через класи, породжені від шаблонів класів `UnaryExprt` і `BinaryExprt`, кожен з яких параметризується структурною інформацією. Ці шаблони класів братимуть типи їх подвираженій як "типові" шаблонні аргументи. Крім того, ми параметризуємо шаблони класів виразів типом операції, які вони представляють, тобто, фактична операція (`+, -, *, ++, -, exp, log, sqrt.`) буде надана як об'єкт функції і її тип буде одним з шаблонних аргументів в шаблоні класу виразу.

Термінальні вирази будуть реалізовані як звичайні (нешаблонні) класи як при об'єктно-орієнтованому підході.

Замість рекурсії на етапі виконання використовуватимемо рекурсію на етапі компіляції: ми замінимо рекурсивний виклик віртуальної обчислювальної функції на рекурсивне створення екземплярів шаблонів класів виразів.



Вихідний код реалізації приведений в лістингу 11.

Лістинг 11: Шаблонна реалізація інтерпретатора арифметичних виразів

```
class Literal {
public:
   Literal(const double v) : _val(v) {}
   double eval() const { return _val; }
private:
   const double _val;
};
class Variable {
public:
   Variable(double& v) : _val(v) {}
   double eval() const { return _val; }
private:
   double& _val;
};

template <class ExprT1,class ExprT2, class BinOp>
class BinaryExpr {
public:
   BinaryExpr(ExprT1 e1, ExprT2 e2,BinOp op=BinOp())
      : _expr1(e1),_expr2(e2),_op(op) {}
   double eval() const
   { return _op(_expr1.eval(),_expr2.eval()); }
private:
   ExprT1 _expr1;
   ExprT2 _expr2;
   BinOp  _op;
};
...
```

Шаблон класу для `UnaryExprt` реалізується за аналогією з класом `BinaryExprt`. Як операцій ми можемо використовувати зумовлені в STL типи об'єктів функцій плюс, мінус, множення і ділення, і т.д., а також можемо визначити наші власні типи об'єктів функцій в разі потреби. Наприклад, бінарне вираз, що представляє суму, матиме тип `BinaryExprt <ExprtT1, ExprtT2, plus <double>>` . Оскільки це ім'я типу досить громіздке, для більш зручного використання додають Створювальні функції.

Створювальні функції

Створювальні функції - техніка, що широко застосовується в поєднанні з шаблонним програмуванням. У STL існує багато прикладів створюють функцій, наприклад, `make_pair()` одна з них. Створювальні функції є допоміжними функціями, які використовують той факт, що компілятор автоматично виводить типові аргументи шаблонів функцій, в той час як для шаблонів класів такого автоматичного висновку не існує.

Кожен раз, коли ми створюємо об'єкт типу, який генерується з шаблону класу, ми повинні повністю вказати згенероване ім'я типу, включаючи всі типові аргументи шаблону. Часто ці генеровані типові імена є дуже довгими, складними для читання та розуміння. Як приклад розглянемо пару пар. Таким типом буде щось на зразок `pair <pair <string, complex <double>>, <pair <string, complex <double>>>`. Створювальні функції полегшують користування шаблонами: створюють функції, створюють об'єкт типу, який генерується з шаблону класу без необхідності вказівки довгих імен типу.

Точніше, те що створюється функціями є шаблони функцій, і вони мають ті ж самі типові шаблонні аргументи, що і клас шаблону, який описує тип об'єкта, який повинен бути створений. У прикладі пар, шаблон класу `pair` має два аргументи типу T1 і T2, які представляють типи включаються елементів і створює функція `make_pair()` має ті ж два аргументи типу.

```
template <class T1, class T2>
class pair {
public:  pair(T1 t1,T2 t2);
};
template <class T1, class T2>
pair<T1,T2> make_pair(t1 t1, T2 t2)
{ return pair<T1,T2>(t1,t2); }
```

Створювальні функції схожі на конструктор: аргументи, які передаються до створювальної функції є такими ж аргументами, які б передалися конструктору створюваного об'єкта в іншому випадку. Так як компілятор автоматично виводить аргументи шаблонів функцій, немає необхідності вказувати всі шаблонні аргументи при виклику створювальної функції; компілятор буде автоматично виводити їх з аргументів, переданих створювальній функції. Замість створення пари через конструктор

```
pair< pair<string,complex<double>>, pair<string,complex<double> > >
( pair<string,complex<double> >("origin", complex<double>(0,0)),
  pair<string,complex<double> >("saddle", aCalculation())
)
```

можна створювати пару засобами створювальної функції:

```
make_pair(make_pair("origin", complex<double>(0,0)),
          make_pair("saddle", aCalculation())
         )
```

У лістингу 12 нижче показано два приклади створювальних функцій:

Лістинг 12: Створювальні функції для об'єктів виразів

```
template  <class ExprT1,class ExprT2>
BinaryExpr<ExprT1,ExprT2,plus<double> >
makeSum(ExprT1 e1, ExprT2 e2)
{ return BinaryExpr<ExprT1,ExprT2,plus<double> >(e1,e2); }
template  <class ExprT1,class ExprT2>
BinaryExpr <ExprT1,ExprT2,multiplies<double> >
makeProd(ExprT1 e1, ExprT2 e2)
{ return    BinaryExpr<ExprT1,ExprT2,multiplies<double> >(e1,e2);  }
```

Лістинг 13 показує використання шаблонної реалізації інтерпретатора для розрахунку вираження (x + 2) * 3.

Лістинг 13: Використання шаблонного інтерпретатора для арифметичних виразів

```
void someFunction (double x)
{
  BinaryExpr< BinaryExpr < Variable,Literal,plus<double> >,
              Literal,
              multiplies<double> >
  expr = makeProd (makeSum (Variable(x), Literal(2)), Literal(3));
  cout << expr.eval() << endl;
}
```

По-перше, створюється об'єкт вираження, який представляє вираз (x + 2) * 3, а потім розраховується сам вираз. Зауважимо, що в цьому рішенні тип об'єкта вираження вже відображає структуру синтаксичного дерева.

Часто досить довге ім'я типу об'єкта вираження навіть не потрібно вказувати. У наведеному вище прикладі не потрібна змінна `expr` ми можемо безпосередньо використовувати результати створює функції `makeProd()` для розрахунку вираження:

```
cout
  << makeProd(makeSum(Variable(x),Literal(2)),Literal(3)).eval()
  << endl;
```

Оцінка

Що ми отримали за допомогою реалізації інтерпретатора на основі шаблонів, а не успадкування? Якщо компілятор вбудовує всі створювальні функції, конструктори і функції `Eval()` (швидше за все так і буде, так як вони тривіальні) вираз

```
cout
  << makeProd(makeSum(Variable(x),Literal(2)),Literal(3)).eval()
  << endl;
```

зводиться до (x + 2) * 3.

Порівняйте це з

```
Product expr(new Sum(new Variable(x),new Literal(2)), new Literal(3)).eval
```

(див. Лістинг 10). Це призводить до ряду розподілів пам'яті і наступного конструювання, а також декільком викликам віртуальної функції `eval()`. Швидше за все, жоден з викликів `eval()` не вбудованим, так як компілятори зазвичай не вбудовують функції, які викликаються через покажчики.

Рішення на основі шаблонів виконується **набагато швидше**, ніж об'єктно-орієнтована реалізація.

Подальше вдосконалення рішення на основі шаблонів

Налаштуємо шаблони виразів, що перетворить їх в щось дійсно корисне. Спочатку покращимо їх читабельність. Ми хочемо зробити вираз, такий як

```
makeProd(makeSum(Variable(x),Literal(2)),Literal(3)).eval()
```

зручнішим для читання в тому сенсі, аби вінвиглядав більш-менш схоже на вираз, який він представляє, а саме: (x + 2) * 3. Це може бути досягнуто шляхом перевантаження операторів. Шляхом незначних модифікацій висловом можна надати такий вигляд `eval((v+2)*3.0)`.

Перша зміна полягає в перейменуванні створювальних функцій так, щоб вони були перевантаженими операторами; тобто перейменуємо `makeSum()` в `operator+()`, `makeProd()` в `operator*()`, і так далі. Тоді

```
makeProd(makeSum(Variable(x),Literal(2)),Literal(3))
```

перетворюється на:

```
((Variable(x) + Literal(2)) * Literal(3))
```

Та все одно ціль полягає в тому, аби написати ((x + 2) * 3). Таким чином, потрібно усунути створення змінних і літералів, які, як і раніше захаращують вираз.

Для того щоб з'ясувати, як ми можемо покращити наше рішення, розглянемо, що означає вираз x + 2, яке ми перейменували з створює функції `makeSum()` в `operator+()`.

Лістинг 14: створити функція для вираження суми через перевантажений `operator+()`

```
template  <class ExprT1,class ExprT2>
BinaryExpr<ExprT1,ExprT2,plus<double> >
operator+(ExprT1 e1, ExprT2 e2)
{ return BinaryExpr<ExprT1,ExprT2,plus<double> >(e1,e2); }
```

Та, необхідно, аби x + 2 відповідав `operator+(x,2)`, який раніше був `makeSum(x,2)`. З цієї причини x + 2 є результатом створення об'єкта бінарного вираження, що представляє собою суму, де в якості аргументів були передані змінна x типу `double` і буквальний 2 типу `int`. Більш точно, це безіменний об'єкт, створений як BinaryExpr <`double`, `int, plus <double>>(x,2)`. Слід зауважити, що тип об'єкта, не є метою. Потрібно створити об'єкт типу `BinaryExprt <Variable, Literal, plus<double>>` , але при автоматичному виведення шаблонних аргументів невідомо, що x є змінною, а 2 літералом. Компілятор виводить тип `double` з аргументу x і тип `int` з аргументу 2, тому що він перевіряє типи аргументів, переданих функції.

Тож, необхідно скерувати виведення компілятора. Якби ми передали об'єкт типу `Variable` замість оригінальної змінної x, то автоматичний висновок аргументів дав би результат типу `BinaryExprt <Variable, int, plus<double>>`.

Лістинг 15: Використання шаблонного інтерпретатора для арифметичних виразів

```
void someFunction (double x)
{
  Variable v = x;
  cout << ((v + 2) * 3).eval() << endl;
}
```

При використанні об'єкта v типу `Variable` замість простої числової змінної, вдалося домоглися того, що такий вислів, як v + 2 розраховується як неіменований об'єкт `BinaryExprt <Variable, int, plus<double>>(v,2)`. Такий об'єкт `BinaryExprt` має два члена даних типу `Variable` та `int` відповідно. Обчислювальна функція `BinaryExprt <Variable, int, plus<double>>::eval()` буде повертати суму двох членів даних. Суть в тому, що член даних `int` не знає, як себе обчислити. Необхідно перетворити літерал 2 в об'єкт

типу `Literal`, який вже знає, як себе знайти. Для вирішення даної проблеми, визначимо властивості виразів `traits`.

Властивості traits

Використання властивостей є ще однією поширеною методикою програмування в поєднанні з шаблонним програмуванням. Клас властивостей це супроводжуючий клас, який слідує разом з іншим типом і містить інформацію, яка пов'язана з цим іншим типом.

Стандартна бібліотека C ++ має декілька прикладів властивостей: прикладом є властивості символів . Як ви знаєте, стандартний клас `string` є шаблоном, параметризрваним символьним типом для подання вузьких і широких символів. В принципі, шаблон класу `string`, який насправді називається `basic_string`, може створюватися з будь-яким типом символів, а не тільки з двома символьними типами, які зумовлені в мови C ++. Якщо, наприклад, потрібно представляти японські символи структурою `Jchar`, то шаблон `basic_string` може використовуватися для створення строкового класу для японських символів, а саме `basic_string <Jchar>`.

Наприклад, реалізовуватимемо такий шаблон строкового класу. Виявиться, що є уся потрібна інформація, але яка не міститься в символьному типі. Як тоді, наприклад, можна вирахувати довжину рядка. Підрахунком всіх символів в рядку до тих пір, поки не знайдеться символ кінця рядка. Як дізнатися який символ є символом кінця рядка? Відомо, що це '\ 0' в разі вузьких символів типу `char`, і існує відповідний символ кінця рядка для символів типу `wchar_t`, але як визначити символ кінця рядка для японських символів типу `Jchar`? Очевидно, що інформація про символ кінця рядка є частиною інформації, асоційованої з типом кожного символу, але не міститься в символьному типі. І це саме те, для чого використовуються властивості: вони надають інформацію, пов'язану з типом, але не міститься в типі.

Тип властивостей є супроводжуючим типом, як правило, це шаблон класу, який може бути створений або спеціалізований для групи типів і надає інформацію, пов'язану з кожним типом. Символьні властивості, наприклад, містить статичний символьну константу, що позначає символ кінця рядка для символьного типу.

Застосовуватимемо техніку `traits` для вирішення проблеми шляхом перетворення числових літералів в об'єкти типу `Literal`. Визначимо властивості виразів, які для кожного типу виразу передбачають інформацію про те, як вони повинні зберігатися всередині об'єктів виразів, операндами яких вони є. Все сутності числових типів

повинні зберігатися у вигляді об'єктів типу `Literal`; всі об'єкти типу `Variable` повинні зберігатися як `Variables`; і всі нетермінальні об'єкти виразів повинні також зберігатися.

Лістинг 16: Властивості виразів

```
template <class ExprT> struct exprTraits
{ typedef ExprT expr_type; };
template <> struct exprTraits<double>
{ typedef Literal expr_type; };
template <> struct exprTraits<int>
{ typedef Literal expr_type; };
...
```

Клас властивостей виразів визначає вкладений тип `exprt_type`, який представляє собою тип виразу для об'єкта вираження. Існує шаблон загальних властивостей, який визначає тип виразу для всіх виразів, які відносяться до типів класів, таких як `BinaryExprt, UnaryExprt` або `Variable`. Крім того, існують спеціалізації шаблону класу для всіх вбудованих числових типів, таких як `Variable short, int, long, float, double` і т. Д. Для всіх Некласові виразів тип виразу визначається як тип `Literal`.

Всередині визначення класів `BinaryExprt, UnaryExprt` використовуватимемо властивості виразів для визначення типів членів даних, що містять підвирази.

Лістинг 17: Використання властивостей виразів

```
template <class ExprT1,class ExprT2, class BinOp>
class BinaryExpr {
public:
   BinaryExpr(ExprT1 e1, ExprT2 e2,BinOp op=BinOp())
      : _expr1(e1),_expr2(e2),_op(op) {}
   double eval() const
   { return _op(_expr1.eval(),_expr2.eval()); }
private:
   exprTraits<ExprT1>::expr_type _expr1;
   exprTraits<ExprT2>::expr_type _expr2;
   BinOp  _op;
};
```

Завдяки використанню властивостей виразів об'єкт типу `BinaryExprt<Variable, int, plus<double>>`міститиме два операнда у вигляді об'єктів типів `Variable` і `Literal`, що й був необхідно.

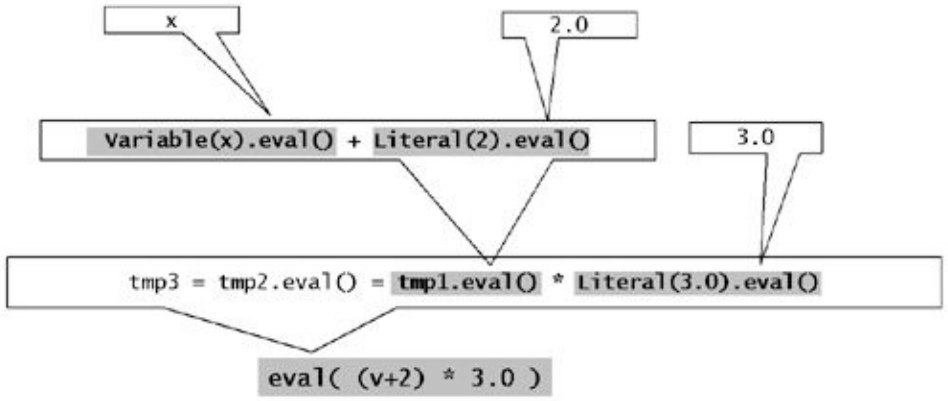
Тепер вираз, такий як `((v+2)*3).eval()`, де v є `Variable` - обгорткою `double` змінної x, буде обчислюватися як (x + 2) * 3. Зробимо деякі незначні зміни, для

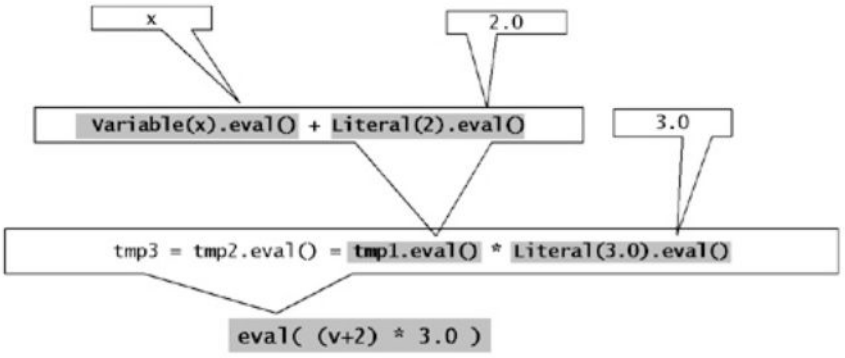
поліпшення зручності. Якщо визначити допоміжну функцію, можна буде перетворити вираз `((v+2)*3).eval()` будь-що то, схоже на `eval((v+2)*3)`.

Лістинг 18: Допоміжна функція `eval()`

```
1 template <class ExprT>
2 double eval(ExprT e) { return e.eval(); }
```

Рисунок нижче ілюструє, як вираз `((v+2)*3).eval()`, де v є `Variable` - обгорткою `double` змінної x, поступово розгортається під час компіляції до вираження (x + 2) * 3.





Посилання:

https://en.wikipedia.org/wiki/Metaprogramming
http://www.angelikalanger.com/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm
https://youtu.be/vhjL_qbmhyE
http://cpp-reference.ru/articles/expression-templates/
https://en.wikipedia.org/wiki/Template_(C%2B%2B)