Лекція 17

Шаблонне метапрограмування

Компілятор мови C++ має подвійну природу. З одної сторони, він являє собою типовий транслятор, призначений для перекладу тексту програми на машинну мову. З іншої сторони він має властивості інтерпретатора, що дозволяє використовувати його для обчислення статичних виразів і створювати програми, що інтерпретуються на етапі компіляції. Такі програми називаються *статичними метапрограмами*. Вони являють собою частковий випадок *узагальнених метапрограм*, що генерують інші програми.

Статичні метапрограми

В основі статичних метапрограм лежать шаблонні класи і рекурсія. Зрозуміло, що шаблонне метапрограмування використовує лише деякі з широких можливостей мови C++ і обмежує програміста певним набором процедур. Розглянемо ці прийоми детальніше.

Автором першої шаблонної метапрограми був Ервін Унрух (Erwin Unruh), що реалізував рекурсивний алгоритм розпізнавання простих чисел в процесі компіляції. Перший детальний аналіз шаблонних метапрограм і прийомів метапрограмування був виконаний Тоддом Фельдхойзеном (Todd Veldhuizen). В своїй статті він виклав основні прийоми шаблонного метапрограмування на прикладі сортування цілочисельного масиву бульбашковим методом і перелічив основні прийоми інтерпретації керуючих конструкцій на етапі компіляції. Програма, наведена нижче, являє собою аналог програми, що була описана Фельдхойзеном в своїй статті. Шаблонні класи мають додатковий параметр, що розширює їх спектр спеціалізації. Крім того, з метою порівняння результатів в програму включені два варіанти традиційного сортування бульбашковим методом — ітеративний і рекурсивний. Помітимо, що в цій програмі використовується часткова спеціалізація — можливість, яку не підтримує більшість компіляторів. Перш ніж виконувати її (програму) необхідно впевнитися, що ваш компілятор дозволяє використовувати часткову спеціалізацію шаблонних класів.

Метапрограма сортування масиву бульбашковим методом[i]

```
#include <iostream.h>

template <typename T>
void swap(T& x, T& y)
{
    T tmp=x;
    x=y;
    y=tmp;
}

template <typename T>
void bubble_iterative(T* array, int N)
{
    for(int i=N-1; i>0;i--)
    for(int j=1; j<i+1; j++)
       if (array[j]<array[j-1]) swap(array[j-1],array[j]);
}

template <typename T>
void bubble_recursive(T* array, int N)
{
    for(int j=1; j<N; j++)
       if (array[j]<array[j-1]) swap(array[j-1],array[j]);
    if (N > 2) bubble_recursive(array, N-1);
}

template <typename T, int N>
class MetaBubble
{
    public:
```

```
        static inline void bubble(T* array)
        {
            MetaBubbleLoop<T,N-1,0>::loop(array);
            MetaBubble<T,N-1>::bubble(array);
        }
};

template <typename T>
class MetaBubble<T,1>
{
    public:
       static inline void bubble(T* array) { }
};

template <typename T, int I, int J>
class MetaBubbleLoop
{
    enum { go = (J <= I-2) } ;
    public:
       static inline void loop(T* array)
       {
          MetaBubbleSwap<T,J,J+1>::compareAndSwap(array);
          MetaBubbleLoop<T, go ? I: 0, go ? (J+1) : 0> ::loop(array);
       }
};

template <typename T>
class MetaBubbleLoop<T,0,0>
{
    public:
       static inline void loop(T*){};
};

template<typename T, int I, int J>
class MetaBubbleSwap {
    public:
       static inline void compareAndSwap(T* array)
       {
         if (array[I] > array[J])
            swap(array[I], array[J]);
       }
};

template <typename T>
void view(T* array, int N)
{
    for(int i=0; i<N; i++)
    cout << array[i] << " ";
    cout << endl;
}

int main()
{
    // Ітеративна версія

    int n=3;
    int vector[3]={9,8,7};

    bubble_iterative(vector, n);
    view(vector, n);
```

```
    // Рекурсивна версія

    int array[3]={9,8,7};
    bubble_recursive(array, n);
    view(array, n);

    // Статична версія — сортування на етапі компіляції

    int data[3]={9,8,7};
    MetaBubble<int,3>::bubble(data);
    view(data, n);

    return 0;
}
```

Виконання програми призводить до наступних результатів:

```
7 8 9
7 8 9
7 8 9
```

Проаналізуємо механізм шаблонного метапрограмування, що лежить в основі цієї програми. Для впорядкування масиву `data`, що містить `N` об'єктів класу `T`, викликається функція-член `bubble()` шаблонного класу `MetaBubble<T,N>`. В свою чергу ця функція викликає функцію `loop()`, що являє собою рекурсивний член класу `MetaBubbleLoop(T,N-1,0)`. Рекурсивні виклики функції `loop()` реалізуються циклом з лічильником `j`. Базою рекурсії є значення `N = 1`. Треба звернути особливу увагу на те, що перевірка базової умови виконується за допомогою переліченої константи `go` і тернарного оператора. Якщо значення `go` дорівнює нулю, шаблонні параметри `I` та `J` замінюються нулями. Перестановка сусідніх елементів масиву виконується функцією `compareAndSwap()`, що належить шаблонному класу `MetaBubbleSwap<I,J+1>`. Як бачимо, усі елементи метапрограм є статичними — їх значення відомі на етапі компіляції.

Перелічний тип

Оскільки статична метапрограма може оперувати лише інтегральними константами періоду компіляції, програміст має вибір: використовувати константи переліченого типу чи статичні константи. В наведеній вище програмі використовується перша з двох можливостей. Для того, щоб продемонструвати другу, замінимо перелічену константу її статичним аналогом. Інакше кажучи, замість рядка

```
enum { go = (J <= I-2) } ;
```

в класі `MetaBubbleLoop` напишемо оголошення

```
static int const go = (J <= I-2);
```

І перший і другий варіанти метапрограми володіють однаковими функціональними можливостями. Виникає природне питання: якому варіанту необхідно надати перевагу? Оскільки метапрограмування оперує константами, необхідно обмежуватись лише тими сутностями, які не мають адреси і не можуть знаходитися в лівій частині оператора присвоювання. Таким чином, статична константа — не найкращий варіант. Вона має адресу і, крім того, може отримувати значення при ініціалізації. На відміну від неї константа переліченого типу практично нічим не відрізняється від літеральної константи, а отже, ідеально підходить для статичного метапрограмування.

2. Рекурсія

В основі статичного метапрограмування лежить рекурсія. Компілятор не здатен виконувати ітеративні процедури, тому всі цикли необхідно реалізувати за допомогою рекурсивної конкретизації шаблонних класів, в яких лічильник циклу відіграє роль шаблонного параметру.

В попередньому прикладі рекурсивна процедура сортування містить один цикл.

```
for(int j=1; j<N; j++)
    if (array[j]<array[j-1]) swap(array[j-1],array[j]);
if (N > 2) bubble_recursive(array, N-1);
```

Використовуючи засоби шаблонного метапрограмування, його можна реалізувати за допомогою первинного шаблону та часткової спеціалізації.

```
// Первинний шаблон
template <typename T, int I, int J>
class MetaBubbleLoop
{
    enum { go = (J <= I-2) } ;
    public:
       static inline void loop(T* array)
       {
          MetaBubbleSwap<T,J,J+1>::compareAndSwap(array);
          MetaBubbleLoop<T, go ? I: 0, go ? (J+1) : 0> ::loop(array);
       }
};

// Часткова спеціалізація
template <typename T>
class MetaBubbleLoop<T,0,0>
{
    public:
       static inline void loop(T*){};
};
```

Компілятор виконує цикл шляхом рекурсивної конкретизації шаблону, доки значення переліченої константи `go` не стане рівним нулю. В цьому випадку буде виконана часткова спеціалізація і процес конкретизації завершиться.

Основні керуючі конструкції

Обмеження, що накладаються на метапрограми, змушують програмістів використовувати особливі варіанти керуючих конструкцій. Тодд Фельдхойзен запропонував наступні статичні аналоги основних примітивів.

Оператор if-else

Традиційна версія

```
if (условие)
    оператор1;
else
    оператор2;
```

Статична версія

```
template <bool Condition>
class IfElse { };

class IfElse<true> {
public:
    static inline void f()
    { оператор1; }          // Умова істинна
};

class IfElse<false> {
public:
    static inline void f()
    { оператор2; }          // Умова не істинна
};

// Застосування
IfElse<умовний вираз>::f();
```

В залежності від умовного виразу, метапрограма виконує спеціалізацію одного з двох шаблонних класів, параметром якого є булева змінна. Якщо компілятор не підтримує роботу з булевими змінними, шаблонний параметр `Condition` можна зробити цілочисельним.

Оператор switch

Традиційна версія

```
int i;

switch(i)
{
    case значення1:
         оператор1;
         break;
    case значення2:
         оператор2;
         break;
    default:
         оператор3;
         break;
}
```

Статична версія

```
template<int I>
class Switch {
public:
    static inline void f()
    { оператор3; }
};

class Switch<значення1> {
public:
    static inline void f();
    { оператор1; }
};


class Switch<значення2> {
public:
    static inline void f();
    { оператор2; }
};

// Застосування

  Switch<I>::f()
```

В залежності від значення константи `I`, компілятор спеціалізує шаблонні класи `Switch<значення1>` чи `Switch<значення2>`.

Цикл do

Традиційна версія

```
int i = N;
do{
   оператор;
} while (--i > 0);
```

Статична версія

```
template<int I>
class Do {
private:
    enum { go = (I-1) != 0);
public:
    static inline void f()
```

```
    {
       оператор в тілі циклу;
       Do < go ? (I-1) : 0>::f();
    }
};

// База рекурсії
class Do<0> {
    public:
       static inline void f() {}
};

// Застосування

  Do<N>::f();
```

Компілятор `N` раз генерує оператор в тілі циклу, причому значення оператора залежить від шаблонного параметру `I`.

Цикл while

Традиційна версія

```
int i = 0;
while (i < N){
   оператор;
};
```

Статична версія

```
template<int I, int N>
class While {
private:
    enum { go = (I+1) != N);
public:
    static inline void f()
    {
       оператор в тілі циклу;
       While < go ? (I+1) : N, N>::f();
    }
};

// База рекурсії
class While<N,N> {
public:
    static inline void f() {}
};

// Застосування

  While<0,N>::f();
```

Як і для оператора `do`, компілятор `N` раз генерує оператор, залежний від шаблонного параметру `I`. Але оскільки в даному випадку реалізується цикл, що має не лише нижню, але і верхню границі, нам знадобилося два шаблонні параметри. Перший являє собою лічильник циклу (його нижня границя визначається при першій спеціалізації) а другий параметр визначає верхню границю циклу. Таким чином, базою рекурсії є спеціалізація `While<N,N>`.

Цикл for

Традиційна версія

```
for( int i = 0; i < N; i++)
{
   оператор;
};
```

Статична версія

```
template<int I, int N>
class For {
private:
    enum { go = (I+1) != N);
public:
    static inline void f()
    {
       оператор в тілі циклу;
       For < go ? (I+1) : N, N>::f();
    }
};

// База рекурсії
class For<N,N> {
public:
    static inline void f() {}
};

// Застосування

  For<0,N>::f();
```

Так як семантика оператору `for` нічим не відрізняється від семантики оператору `while`, описаного вище, його опис класів `While` і `For` збігаються.

Статичні метафункції

Метафункція являє собою шаблонову конструкцію, що генерує константні дані під час компіляції. Зокрема, їх можна застосовувати для рекурсивного обчислення математичних функцій.

Елементарні функції

В своїй статті Тодд Фельдхойзен продемонстрував корисні прийоми обчислення бібліотечних функцій під час компіляції. В основі цих функцій лежить здатність компіляторів оптимізувати вирази, використовуючи завчасно відомі значення (значення, що обчислювались раніше і вже є відомими), а також виконувати цілочисельні константні обчислення. Застосуємо ці методи для обчислення функції $\exp(x)=1+x+x^2/2+x^3/3+\ldots$, використовуючи схему Горнера. Спочатку наведемо традиційну реалізацію цієї функції за допомогою звичайної програми, написаній мовою C++.

Традиційне обчислення експоненти

```
#include <iostream.h>
#include <math.h>

double expon(double, int);

int main()
{
    double x = 0.5;
    cout << "expon(1,10) = " << expon(1,10) << endl;
    cout << "exp(1.0)    = " << exp(1.0) << endl;

    return 0;
}

double expon(double x, int n)
{
```

```
    // Обчислюємо exp(x), використовуючи n членів ряду Тейлора
    double value = 1;
    for (int i = n-1; i > 0; --i)
    value = 1 + x/i*value;
    return value*x;
}
```

Результати приведені нижче

```
expon(1,10) = 2.71838

exp(1.0)    = 2.71838
```

Використовуючи прийоми шаблонного метапрограмування, ми можемо створити еквівалентну програму, яка буде виконуватись компілятором.

Обчислення експоненти за допомогою метафункції

```
#include <iostream.h>

template <int N, int I>
class Exp {
public:
    static inline float exponenta()
    {
       return float(I/N)*ExpSeries<N,I,10,1>::sum();
    }
};

// Обчислюємо J членів ряду Тейлора, використовуючи параметр K
// як лічильник циклу.
// Аргумент x представляється у вигляді дробу I/N

template<int N, int I, int J, int K>
class ExpSeries {
public:
    enum { go = (K+1 != J) };
    static inline float sum()
    {
        cout << K << endl;
        return 1 + float(I/N)/K*ExpSeries<N*go, I*go,
J*go,(K+1)*go>::sum();
    }
};

// Спеціалізація бази рекурсії
class ExpSeries<0,0,0,0> {
public:
    static inline float sum()
    { return 0; }
};

int main()
{
    float e = Exp<1,1>::exponenta();
    cout << "exp(1) = " << e;
    return 0;
}
```

В результаті отримаємо наступне повідомлення:

```
exp(1) = 2.71828
```

Як бачимо, для обчислення елементарної метафункції необхідно представити її у вигляді рекурсивної схеми і реалізувати у вигляді шаблонних класів. Необхідно відмітити, що раціональний аргумент функції `exp()` ми були змушені представити у вигляді дробу `I/N`, оскільки *механізм шаблонів під час компіляції дозволяє виконувати лише цілочисельні операції над константами*. Спроба ввести шаблонний параметр `float` чи `double` приречена на невдачу.

За допомогою описаних вище прийомів можна реалізувати довільну рекурсивну функцію. Ось як, наприклад, може виглядати шаблонна метафункція, що виводить на екран шістнадцяткове представлення заданого цілого числа.

Виведення на екран шістнадцяткового числа

```
#include <iostream.h>

template<int n>
class DisplayHex {
public:
    enum { go = (n/16 > 0) };
    static inline void Hex()
    {
       DisplayHex<(n/16)*go>::Hex();
       cout << hex << n%16;
    }
};

// Спеціалізація бази рекурсії
class DisplayHex<0> {
public:
    static inline void Hex()
    {
      return;
    }
};

void displayHex(int n)
{
    if (n/16 > 0)
       displayHex(n/16);
    cout << hex << n%16;
}

int main()
{
    cout << "Шаблонна метафункція: ";
    DisplayHex<160>::Hex();
    cout << endl;
    cout << "Рекурсивна функція  : ";
    displayHex(160);
    return 0;
}
```

Результат її роботи має наступний вигляд:

```
Шаблонна метафункція: a0

Рекурсивна функція  : a0
```

Метод перетворення рекурсивної функції в метафункцію достатньо простий:

1) Створюємо шаблонний клас, що містить перелічення і статичну inline-функцію, що реалізує рекурсивний алгоритм;
2) Передбачаємо окрему часткову спеціалізацію для базі рекурсії.

Перелічення дозволяє розгортати рекурсію, а статична функція виконує обчислення, рекурсивно викликаючи саму себе за допомогою послідовних спеціалізацій шаблонного класу, доки відповідний шаблонний параметр не стане рівним нулю. В кінці кінців виконується спеціалізація шаблонного класу, що

описує базу рекурсії, і обчислення припиняються. Результат обчислень зберігається чи в створеному об'єкті, чи є безпосереднім продуктом роботи статичної inline-функції.

Обчислення норми вектора

Як правило, виконання програм, що містять довгі цикли, важко оптимізувати. В таких ситуаціях статичне метапрограмування, що базується на процедурі розгортання циклу, допомагає досягти відмінних результатів. Розглянемо цей підхід на прикладі обчислення скалярного добутку.

Метапрограма для обчислення норми вектора

```
#include <iostream.h>
#include <math.h>

// Первинний шаблон для обчислення скалярного добутку (x,x)
template <typename T, int N>
class MetaScalar
{
public:
    static T square(T* x)
    {
       return  x[0]*x[0] + MetaScalar<T,N-1>::square(x+1);
    }
};

// Часткова спеціалізація скалярного добутку - база рекурсії
template <typename T>
class MetaScalar<T,0>
{
public:
    static T square(T* x)
    {
       return 0 ;
    }
};

template <typename T, int N>
inline T scalar(T* x)
{
    return sqrt(MetaScalar<T,N>::square(x));
}

int main ()
{
    double x[] = { 1.0, 2.0, 2.0};
    cout << scalar<double,3>(x);

    return 0;
}
```

Компілятор виконує рекурсивну конкретизацію шаблонів, доки не досягне бази рекурсії. Синхронно з конкретизацією шаблонів виконується сумування добутків компонент вектора, піднесених до квадрату. Часткова спеціалізація, що відповідає базі рекурсії, завершує процес обчислення норми вектора.

Порівняння типів

Механізм метафункцій можна застосувати для вибору одного з перелічених типів в залежності від істинності умови, що перевіряється..

Порівняння типів (перший варіант)

```
#include <iostream.h>

template<bool Condition, typename First, typename Second>
```

```
struct If
{
    typedef First Result;
    Result x;
};

template<typename First, typename Second>
struct If<false, First, Second>
{
    typedef Second Result;
    Result x;
};

int main()
{
    If<true,int,double> x;
    cout << "sizeof(int)    = "    << sizeof(x) << endl;
    If<false,int,double> y;
    cout << "sizeof(double) = " << sizeof(y) << endl;

    return 0;
}
```

Програма виводить на екран наступні повідомлення:

```
sizeof(int)    = 4

sizeof(double) = 8
```

Її логіка така: якщо перевіряємо умова істинна і булава змінна `Condition` приймає значення `true`, повертається тип `First`. В іншому випадку виконується часткова спеціалізація `struct If<false, First, Second>` і тип `Result` стає типом `Second`.

Цю конструкцію можна застосувати для порівняння типів за розміром. Наприклад, це необхідно при приведенні типів, оскільки тип, що приводиться повинен мати менший розмір, ніж результуючий, інакше неминучою є втрата інформації. Застосуємо для порівняння розмірів типів структуру `If`.

Порівняння типів (другий варіант)

```
#include <iostream.h>
template<bool Condition, typename First, typename Second>
struct If
{
    typedef First Result;
    Result x;
};

template<typename First, typename Second>
struct If<false, First, Second>
{
    typedef Second Result;
    Result x;
};

template<typename A, typename B>
struct Greater
{
    typedef If<sizeof(A)>=sizeof(B),A,B> Result;
    Result x;
};

template<typename A, typename B>
struct LessThen
{
```

```
    typedef If<sizeof(A)<sizeof(B),A,B> Result;
    Result x;
};


int main()
{
    Greater<int,double> x;
    cout << "sizeof(Greater<int,double>)  = "    << sizeof(x) << endl;
    LessThen<int,double> y;
    cout << "sizeof(LessThen<int,double>) = "    << sizeof(y) << endl;

    return 0;
}
```

Результат наведений нижче:

```
sizeof(Greater<int,double>)  = 8

sizeof(LessThen<int,double>) = 4
```

Статична диспетчерізація

В багатьох ідіомах узагальненого програмування, використовується конструкція, що дозволяє генерувати різні типи в залежності від значення цілочисельної константи..

Шаблонний клас IntToType

```
#include <iostream.h>

template <int i>
struct IntToType
{    enum { value = i };
};

template <>
struct IntToType<0>
{    typedef int Result;
    Result x;
};

template <>
struct IntToType<1>
{   typedef double Result;
    Result x;
};
int main()//*********************************************
{
    IntToType<0> x;
    cout << "sizeof<IntToType<0> = " << sizeof(x) << endl;
    IntToType<1> y;
    cout << "sizeof<IntToType<1> = " << sizeof(y) << endl;
    return 0;
}
```

Різним значенням `i` клас `IntToType` ставить у відповідність різні типи. Для цього необхідно визначити окремі спеціалізації класу для конкретних значень шаблонного параметру. В результаті отримаємо на екрані наступні повідомлення:

```
sizeof<IntToType<0> = 4

sizeof<IntToType<1> = 8
```

Метафункції вищого порядку

Метафункції, що є результатами інших метафункцій, називаються метафункціями вищого порядку. Розглянемо як приклад метафункцію, аргументом якої є деяка умова. В залежності від істинності цієї умови функція повертає певний тип.

Подвійна диспетчерізація за допомогою метафункцій

```
#include <iostream.h>

struct First
{
    template<class T, class U>
    struct Select
    {   typedef T Result;
    };
};

struct Second
{   template<class T, class U>
    struct Select
    {
       typedef U Result;
    };
};


template <bool Condition>
struct Dispatcher
{   typedef First Result;
};

template<>
struct Dispatcher<false>
{
    typedef Second Result;
};

template <bool Condition, class T, class U>
class IF
{
    typedef Dispatcher<Condition>::Result Selector;

public:
    typedef Selector::Select<T,U>::Result Result;
    Result x;
};

int main()
{
    IF<true,int,double> x;
    cout << "sizeof( IF<true,int,double>)  = " << sizeof(x) << endl;
    IF<false,int,double> y;
    cout << "sizeof( IF<false,int,double>) = " << sizeof(y) << endl;

    return 0;
}
```

Результат виглядає наступним чином:

```
sizeof( IF<true,int,double>)  = 4

sizeof( IF<false,int,double>) = 8
```

Метафункція `IF()`, на відміну від попередніх прикладів, не використовує механізм часткової спеціалізації. Вона просто аналізує значення булевої змінної `Condition` і, в залежності від її істинності, генерує тип `First` чи `Second`. Зверніть увагу на те, що класи `First` і `Second` містять локальний клас `Select`, який і визначає тип, що повертається — `T` чи `U`. Ці локальні класи дозволяють приховати часткову спеціалізацію під оболонкою класу `Dispatcher`.

Для передачі одній метафункції іншої часто використовуються *шаблонні шаблонні параметри*, тобто шаблонні параметри, що самі є шаблонами.

Дискримінація типів

Метафункції можна застосовувати для перевірки умов, які накладаються на класи. Розглянемо в якості прикладу підхід, що дозволяє перевірити, чи є клас, що задається шаблонним параметром T, вказівником. Цей метод був розроблений американськими програмістами М. Маркусом (Mat Marcus) і Дж. Джоунсом (Jesse Jones). Застосуємо його для розпізнавання елементарних вказівників.

Дискримінация вказівників[ii]

```
#include <iostream.h>

struct TrueType  { char array[1]; };
struct FalseType { char array[2]; };

TrueType  ptr_discriminator(void*);
FalseType ptr_discriminator(...);

template<class T>
class IsPointer
{
private:
    static T t;
    public:
    enum { value = sizeof(ptr_discriminator(t))==sizeof(TrueType) };
};

template <>
class IsPointer<void>
{
public:
    enum { value = false };
};

template <>
class IsPointer<const void>
{
public:
    enum { value = false };
};

template <>
class IsPointer<volatile void>
{
public:
    enum { value = false };
};

int main()
{
    cout << "int*    -> " << IsPointer<int*>::value    << endl;
    cout << "int     -> " << IsPointer<int>::value     << endl;
    cout << "char*   -> " << IsPointer<char*>::value   << endl;
    cout << "char    -> " << IsPointer<char>::value    << endl;
    cout << "double* -> " << IsPointer<double*>::value << endl;
```

```
    cout << "double  -> " << IsPointer<double>::value  << endl;
    return 0;
}
```

Результати роботи цієї програми виглядають наступним чином:

```
int*    -> 1

int     -> 0

char*   -> 1

char    -> 0

double* -> 1

double  -> 0
```

Одиниця означає, що шаблонним параметром є вказівник, а нуль свідчить про те, що тип не є вказівником.

Функція `ptr_discriminator()` є перевантаженною. В першому варіанті вона отримує аргумент `void*` і повертає об′єкт класу `TrueType`. В другому варіанті вона може отримувати довільний інший аргумент (цей факт відображається трикрапкою) і повертає об′єкт класу `FalseType`. Якщо шаблонний параметр є вказівником, викликається функція `ptr_discriminator(PointerShim)`, а значенню `value` присвоюється значення `true`. В іншому випадку визивається функція `ptr_discriminator(...)`, що повертає об′єкт типу `FalseType`. Розміри типів `TrueType` і `FalseType` спеціально задаються різними. Якщо шаблонний параметр є вказівником, функція `ptr_discriminator()` повертає об′єкт класу `TrueType` і умова `sizeof(ptr_discriminator(t))==sizeof(TrueType)` стає істинною. В іншому випадку умова не є істинною, а значення `value` дорівнює `false`.

Необхідно відмітити, що описана вище програма розпізнає не лише вказівники основних типів. Класи, що містять операторну функцію `operator*()`, вона також розпізнає як вказівники. Для того, щоб запобігти цьому, Маркус і Джесс запропонували використовувати допоміжний клас `PointerShim`.

```
struct PointerShim
{
   PointerShim(const volatile void*);
};
```

Тепер для правильного розпізнавання типів перший варіант функції `ptr_discriminator()` необхідно визначити наступним чином.

```
TrueType  ptr_discriminator(PointerShim);
```

Розглянемо ще одну корисну метафункцію, що дозволяє визначити, чи є один клас похідним від іншого. Автором цього підходу є А. Александреску. Його ідея, як і раніше, полягає в застосуванні оператора `sizeof` з перевантаженною функцією.

Дискримінація похідних класів

```
#include <iostream.h>

class Base {};
class Derived: public Base{};

template <class A, class B>
class Test
{
    class  First  { char dummy[1]; };
    class  Second { char dummy[2]; };
    static First  Compare(B);
    static Second Compare(...);
    static A MakeA();
public:
```

```
    enum { flag = (sizeof(Compare(MakeA())) == sizeof(First)) };
};

template <class T>
class Test<T,T>
{
public:
    enum{ flag = 1};
};

#define IsDerived(T,U) Test<const T*, const U*>::flag

int main()
{
    using namespace std;
    cout << "double  -> int         " << Test<double,int>::flag << endl;
    cout << "char    -> char*       " << Test<char,char*>::flag << endl;;
    cout << "int     -> int         " << Test<int,int>::flag    << endl;
    cout << "char    -> vector<int> " << Test<char, vector<char> >::flag
         << endl;
    cout << "Derived -> Base        " << IsDerived(Derived,Base);
    return 0;
}
```

Результати роботи цієї програми виглядає так.

```
double  -> int         1

char    -> char*       0

int     -> int         1

char    -> vector<int> 0

Derived -> Base        1
```

Розпізнавання успадкування використовує механізм неявного перетворення аргументів функції. Викличемо перевантажену функцію `Compare()`, що очікує аргумент класу A, передаючи їй аргумент класу B. Якщо функція поверне об′єкт класу `FirstType`, значить, тип A конвертується в тип B, в іншому випадку викликається друга версія, якій байдужий тип аргумента.

Як і в попередньому випадку, для перевірки використовуються два фіктивних класи `First` і `Second`, що мають заздалегідь різні розміри, а також перевантажена функція `Compare()`, що повертає об'єкти цих класів, в залежності від типу отриманного аргумента. Для завдання довільної альтернативи при перевантаженні функції `Compare` використовується багатокрапка (елліпсис). Тіло функції не визначається, оскільки вона насправді не викликається. Зверніть увагу на те, що у виразі `sizeof(Compare(MakeA()) == sizeof(First))` використовується функція `MakeA()`, що повертає об'єкт класу `A`. Це пов'язано з тим, що конструкція `sizeof(Compare(A()) == sizeof(First))` працює некоректно, через те, що конструктор класу A, передбачений за замовчуванням, є закритим.

Аналогічно можна означити дискримінуючи функції для будь-яких інших типів, наприклад `IsConst`, `IsReference`, `IsVolatile` і т.п. Наприклад, використовуючи ту обставину, що посилання і об'єкти типу `void` неможливо зберігати в масиві, можна створити програму, що розпізнає ці типи.

Дискримінация посилань і типу void

```
#include <iostream.h>

struct TrueType  { char array[1]; };
struct FalseType { char array[2]; };

template<class T>
class IsNonArrayed
{
```

```
static TrueType  discriminator(...);
static FalseType discriminator(T[1]);
static T t;
public:
    enum {value =
sizeof(IsNonArrayed<T>::discriminator(t))==sizeof(TrueType) };
};

template <typename T>
class IsNonArrayed<T&>
{
public:
    enum { value = -1 };
};

template <>
class IsNonArrayed<void>
{
public:
    enum { value = -2 };
};

template <>
class IsNonArrayed<void const>
{
public:
    enum { value = -3 };
};

template <>
class IsNonArrayed<void volatile>
{
public:
    enum { value = -4 };
};

int main()
{
    cout << "int           -> " << IsNonArrayed<int>::value           <<
endl;
    cout << "int&          -> " << IsNonArrayed<int&>::value          <<
endl;
    cout << "int*          -> " << IsNonArrayed<int*>::value          <<
endl;
    cout << "void          -> " << IsNonArrayed<void>::value          <<
endl;
    cout << "const void    -> " << IsNonArrayed<const void>::value    <<
endl;
    cout << "volatile void -> " << IsNonArrayed<volatile void>::value <<
endl;
    return 0;
}
```

Результати наведені нижче.

```
int           -> 1

int&          -> -1

int*          -> 1
```

```
void          -> -2

const void    -> -3

volatile void -> -4
```

Наведемо ще один приклад метапрограми. Цього разу будемо розпізнавати кваліфікатор `const`.

Розпізнавання кваліфікатора const

```
#include <iostream.h>

struct TrueType  { char array[1]; };
struct FalseType { char array[2]; };

template<class T>
TrueType discriminator(const T&);
template<class T>
FalseType discriminator(T&);


template<class T>
class IsConst
{
public:
static T t;
    enum {value = (sizeof(discriminator<T>(t))==sizeof(TrueType)) };
};

int main()
{
    IsConst<int> x;
    cout << "int          -> " << x.value  << endl;
    IsConst<const int> y;
    cout << "const int    -> " << y.value << endl;
    IsConst<int> u;
    cout << "double       -> " << u.value  << endl;
    IsConst<const double> v;
    cout << "const double -> " << v.value << endl;

    return 0;
}
```

Виконання цієї програми призводить до наступних результатів.

```
int          -> 0

const int    -> 1

double       -> 0

const double -> 1
```

Зверніть увагу на те, що в даному варіанті, ми використовували шаблонну перевантажену дискримінуючу функцію `discriminator()`, яка не є членом класу. Якщо б ми спробували визначити її статичним членом класу, довелося б ввести ще один шаблонний параметр і означення класу виглядало б приблизно так:

```
template<class T>
class IsConst
{
public:
template<class W>
static TrueType discriminator(const W&);
template<class W>
```

```
static FalseType discriminator(W&);

static T t;
    enum {value = (sizeof(discriminator<T>(t))==sizeof(TrueType)) };
};
```

В цьому описі принципіально важливо, щоб функція-член `discriminator()` була шаблонною. Саме шаблонний параметр дозволяє розрізняти спеціалізації, що відповідають типам `T` і `const T`. Неможливо розрізнити кваліфікатор `const`, безпосередньо аналізуючи аргументи функції — компілятор діагностує неоднозначність, не розрізняючи аргументи типу `T` і `const T`.

Завершуючи опис мета програм, перелічимо їхні основні властивості:

1) ефективність;
2) виразність;
3) корректність.

Завдяки цьому в теперішній час отримали широку популярність узагальнені бібліотеки, такі як Boost C++ (`http://www.boost.org`), Lambda (`http://lambda.cs.uti.fi`), Loki (`http://www.awl.com/cseng/titles/0-201-70431-5`), Blitz++ (`http://www.oonumerics.org/blitz`).

Резюме

- Компілятор мови C++ має подвійну природу. З одної сторони, він являє собою типовий транслятор, призначений для перекладу тексту програми на машинну мову. З іншої сторони він має властивості інтерпретатора, що дозволяє використовувати його для обчислення статичних виразів і створювати програми, що інтерпретуються на етапі компіляції
- Програми, що інтерпретуються на етапі компіляції, називаються статичними метапрограмами.
- Оскільки статична метапрограма може оперувати лише константами періоду компіляції, програміст має вибір: використовувати константи переліченого типу чи статичні константи.
- Оскільки метапрограмування оперує інтегральними константами, необхідно обмежуватись лише тими сутностями, які не мають адреси і не можуть знаходитися в лівій частині оператора присвоювання. Таким чином статична константа — не найкращий варіант. Вона має адресу і, крім того, може отримувати значення при ініціалізації. На відміну від неї константа переліченого типу практично нічим не відрізняється від літеральної константи, а отже, ідеально підходить для статичного метапрограмування.
- В основі статичного метапрограмування лежить рекурсія. Компілятор не здатен виконувати ітеративні процедури, тому всі цикли необхідно реалізувати за допомогою рекурсивної конкретизації шаблонних класів, в яких лічильник циклу відіграє роль шаблонного параметру.
- *Механізм шаблонів під час компіляції дозволяє виконувати лише цілочисельні операції над константами інтегрального типу.*
- Статична функція виконує обчислення, рекурсивно викликаючи саму себе за допомогою послідовних спеціалізацій шаблонного класу, доки відповідний шаблонний параметр не стане рівним нулю. В кінці кінців виконується спеціалізація шаблонного класу, що описує базу рекурсії, і обчислення припиняються. Результат обчислень зберігається чи в створеному об'єкті, чи є безпосереднім продуктом роботи статичної inline-функції.
- Шаблонні шаблонні параметри — це шаблонні параметри, що самі є шаблонами.

Контрольні питання

1. В чому полягає подвійна природа компілятора на C++?.
2. Які програми називаються статичними метапрограмами?
3. Які константи може використовувати програміст в статичній метапрограмі?
4. Які змінні найкраще підходять для статичного метапрограмування і чому?
5. Що лежить в основі статичного метапрограмування лежить рекурсія?
6. Які операції і з якими змінніми може виконувати механізм шаблонів під час компіляції?
7. Як реалізуються обчислення в статичній функції?

8. Що називається шаблонним шаблонним параметром?

Література

1. Александреску А. Современное проектирование на C++. — М.: Издательский дом «Вильямс», 2003.
2. Вандервурд Д., Джосаттис Н. Шаблоны C++. Справочник разработчика. — — М.: Издательский дом «Вильямс», 2003.
3. Страуструп Б. Язык программирования C++. Специальное издание. ── Спб: Невский диалект, 2002.
4. Шилдт Г. Справочник по C++ . — М.: Издательский дом «Вильямс», 2003.