Zaawansowane CPP/Wykład 8: Metaprogramowanie

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Metaprogramowanie

Ogólnie rzecz biorąc metaprogramowanie oznacza pisanie programów, które piszą programy lub pisania programu, który pisze się sam. W naszym kontekście będzie to oznaczało wykonywanie obliczeń za pomocą szablonów, przy czym obliczenia te są wykonywane podczas kompilacji. Podstawą do tych obliczeń jest rekurencyjna konkretyzacja szablonów. Taką metodą można generować w czasie kompilacji całkiem skomplikowane fragmenty kodu, stąd określenie metaprogramowanie. Przykłady takich "metaszablonów" poznaliśmy już na wykładzie o funkcjach typów. W szczególności działanie na listach typów to właśnie przykłady metaprogramowania. W tym wykładzie przyjrzymy się dokładniej temu zagadnieniu i przeanalizujemy kolejne przykłady.

Potęgi

Zaczniemy od bardzo prostego przykładu (zob. D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"*, rozdz. 17). Napiszemy szablon który ma za zadanie obliczać potęgi liczby 3. Ponieważ w programowaniu za pomocą szablonów musimy posługiwać się rekurencją to zaczynamy od sformułowania problemu w sposób rekurencyjny. To akurat jest bardzo proste:

PRZYKŁAD 8.1

$$3^N = 3 * 3^{N-1}, \quad 3^0 = 1$$

Za pomocą szablonów implementujemy to tak (Źródło: Pow.cpp):

```
template<int  N> struct Pow3 {
  enum {val=3*Pow3<N-1>::val};
};
template<> struct Pow3<0> {
  enum {val=1};
};
```

teraz możemy już użyć w programie np. wyrażenie:

```
i=Pow3<4>::val;
```

Podstawową zaletą metaprogramowania i głównym powodem jego używania jest fakt, że to wyrażenie jest obliczane w czasie kompilacji i efekt jest taki sam jak podstawienia:

```
i=81;
```

Można też zastosować szablon funkcji:

```
template<int N> int pow3() {
  return 3*pow3<N-1>();
};
template<> int pow3<0>() {return 1;}
cout<<pow3<4>()<<endl;
```

(Źródło: Powf.cpp)

Nietrudno jest uogólnić powyższy kod tak aby wyliczał potęgi dowolnej liczby:

```
template<int K,int N> struct Pow { enum
 {val=K*Pow<K,N-1>::val}; };
template<int K> struct Pow<K,0> {
 enum {val=1};
};
```

(Źródło: Pow.cpp)

Tutaj już nie można wykorzystać szablonu funkcji, bo nie zezwala on na specjalizację częściową.
Ograniczeniem dla takich obliczeń jest implementacja kompilatora, przede wszystkim założony limit głębokości rekurencyjnego konkretyzowania szablonów. Dla kompilatora g++ jest on ustawiany za pomocą opcji i defaultowo wynosi 500, dlatego już konkretyzacja `Pow<1,500>::val` się nie powiedzie. Konkretyzacja szablonów wymaga też pamięci i może się zdarzyć, że kompilator wyczerpie limit pamięci lub czasu.
Kolejne ograniczenie to konieczność rachunków na liczbach całkowitych. Wiąże się to z faktem, że tylko stałe całkowitoliczbowe mogą być parametrami szablonów.

Ciąg Fibonacciego

Po opanowaniu powyższych przykładów obliczanie wyrazów ciągu Fibonacciego jest prostym zadaniem. Przytoczymy je jednak tutaj, aby zaprezentować pewną bardzo sympatyczną cechę metaprogramowania za pomocą szablonów. Ciąg Fibonacciego jest definiowany rekurencyjnie:

PRZYKŁAD 8.2

$$f_n = f_{n-1} + f_{n-2}, \quad f_1 = f_2 = 1$$

więc jego implementacja jest natychmiastowa:

```
template<int N> struct Fibonacci {
 enum {val = Fibonacci<N-1>::val+Fibonacci<N-2>::val};
};
template<> struct Fibonacci<1> {
 enum {val = 1};
};
template<> struct Fibonacci<2> {
 enum {val = 1};
};
```

(Źródło: fibonacci_template.cpp)

Przykład ten nie wart byłby może i wspomnienia gdyby nie fakt, że rekurencyjna implementacja ciągu Fibonacciego jest bardzo nieefektywna. Jeśli zaimplementujemy ją w zwykłym kodzie

```
int fibonacci(int n) {
 if(n==1) return 1;
 if(n==2) return 1;
 return fibonacci(n-1)+fibonacci(n-2);
}
```

(Źródło: fibonacci.cpp)

to obliczanie `fibonacci(45)` zajmie np.na moim komputerze ok. 8 sekund. Tymczasem szablon kompiluje sie poniżej jednej sekundy! Skąd taka różnica? Czyżby kompilator był bardziej wydajny niż generowany przez niego kod? W przypadku zwykłego kodu długi czas wykonania bierze się z ogromnej liczby wywołań funkcji `fibonacci`. Liczba ta rośnie wykładniczo z `n` i większość czasu jest marnowana na wielokrotne wywoływanie funkcji z tymi samymi argumentami.

W przypadku użycia metaprogramu szablony konkretyzowane są tylko raz. Więc jeśli już raz policzymy np. `Fibonacci<25>::val` to kolejne żądanie nie spowoduje już rozwinięcia rekurencyjnego, a tylko podstawienie istniejącej wartości. Jak widzieliśmy zysk jest ogromny. Takie pamiętanie wyników raz wywołanych funkcji nazywane jest też programowaniem dynamicznym. Jedynym znanym mi językiem, który bezpośrednio wspiera taki mechanizm jest `Mathematica`.

Pierwiastek kwadratowy

Rozważymy teraz trudniejszy przykład szablonu obliczającego pierwiastek kwadratowy (zob. D. Vandervoorde, N. Josuttis: *"C++ Szablony, Vademecum profesjonalisty"*, rozd. 17), choć tak naprawdę ten kod jest bardziej uniwersalny i, jak zobaczymy, łatwo za jego pomocą zaimplementować inne funkcje. Ponieważ jesteśmy ograniczeni do arytmetyki liczb całkowitych, tak naprawdę nie liczymy pierwiastka z `n` ale jego przybliżenie: największą liczbę całkowitą `k`, taką że `k*k<=n`. W tym celu zastosujemy algorytm przeszukiwania binarnego:

```
int sqrt(int n,int low, int high) {
 if(low==high) return low;
 int mid=(low+high+1)/2;
 if(mid*mid > n )
   return sqrt(n,low,mid-1);
 else
   return sqrt(n,mid,high);
}
```

co już łatwo przetłumaczyć na szablon:

```
template<int N,int L=1,int H=N> struct Sqrt{
 enum  {mid=(L+H+1)/2};
 enum  {res= (mid*mid> N)? (int)Sqrt<N,L,mid-1>::res :
  (int)Sqrt<N,mid,H>::res};
};
template<int N,int L> struct Sqrt<N,L,L> {
 enum {res=L};
};
```

(Źródło: sqrt.cpp)

Łatwo sprawdzić, że kod ten działa poprawnie. Niestety posiada on istotną wadę. W trakcie konkretyzacji szablonu konkretyzowane są oba szablony występujące w wyrażeniu warunkowym, nawet ten, który nie będzie potem używany. Tak więc wykonywana jest duża liczba konkretyzacji, z których tylko ułamek jest potrzebny

(zob. rysunek). Jakie to obciążenie dla kompilatora to łatwo sprawdzić kompilując kod, w którym wywołujemy `Sqrt<10000>`. Ja nie doczekałem się na koniec kompilacji.

Na szczęście istnieje rozwiązanie - należy użyć szablonu `If_then_else` (zob. wykład 6.2.1):

```
template<int N,int L=1,int H=N> struct Sqrt{
 enum  {mid=(L+H+1)/2};
 typedef typename If_then_else<
  (mid*mid> N),
  Sqrt<N,L,mid-1>,
  Sqrt<N,mid,H> >::Result tmp;
 enum  {res= tmp::res};
};
template<int N,int L> struct Sqrt<N,L,L> {
 enum {res=L};
};
```

(Źródło: sqrt_ifte.cpp)

Ten kod powoduje już tylko konkretyzację rzeczywiście wymaganych szablonów, a tych jest dużo mniej: rzędu $O(\log N)$. Tym razem kompilacja wyrażenia `Sqrt<10000>` powiedzie się bez trudu.

Pow(x)

Jak dotąd używaliśmy metaprogramowania do wyliczania wartości stałych. Teraz postaramy się wygenerować funkcje. Zacznijmy od pytania: po co? Pomijając syndrom Mount Everestu (wchodzę na niego bo jest), to głównym powodem jest nadzieja uzyskania bardziej wydajnego kodu. Weźmy dalej za przykład liczenie potęgi, tym razem dowolnej liczby zmiennoprzecinkowej:

```
double pow_int(double x,int n) {
double res=1.0;
for(int i=0;i<n;++i)
 res*=x;
return res;
};
```

(Źródło: powx.cpp)

Patrząc na ten kod widzimy, że w pętli wykonywana jest bardzo prosta instrukcja. Możemy więc się obawiać, że instrukcje związane z obsługą pętli mogą stanowić spory narzut. Co więcej, ich obecność utrudnia kompilatorowi optymalizację kodu oraz może uniemożliwić rozwinięcie funkcji w miejscu wywołania. Najlepiej by było zaimplementować tę funkcję w ten sposób:

```
pow(x,n)= x*...*x; /*n razy*/
```

np.:

```
double pow2(double x) {return x*x;}
double pow3(double x) {return x*x*x;}
double pow4(double x) {return x*x*x*x;}
...
```

Wymaga to jednak kodowania ręcznego dla każdej potęgi, której potrzebujemy. Ten sam efekt możemy osiągnąc za pomocą następującego szablonu funkcji:

```
template<int N> inline double pow(x) {return x*pow<N-1>(x);}
template<>  inline double pow<0>(x) {return 1.0;}
```

(Źródło: powx.cpp)

pod warunkiem, że kompilator rozwinie wszystkie wywołania.

Poniżej zamieszczam wyniki pomiarów wykonania 100 milionów wywołań każdej funkcji (Źródło: powx.cpp). Czas jest podany w sekundach. Zamieściłem wyniki dla różnych ustawień optymalizacji.

	pow_int(x,5)	pow<5>(x)
-O0	7.22	14.78
-O1	0.37	0.04
-O2	0.42	0.05
-O3	0.42	0.05

Widać dramatyczną różnicę po włączeniu optymalizacji. Wiąże się to prawdopodobnie z umożliwieniem rozwijania funkcji `inline`. Potem wyniki już się nie zmieniają ale widać, że szablon `pow` wygenerował funkcję 10 razy szybszą od pozostałych. Pokazuje to dobitnie, że optymalizacja ręczna ciągle ma sens.

Sortowanie bąbelkowe

Zakończymy ten wykład bardziej skomplikowanym przykładem pokazującym, że metaprogramowanie można stosować nie tylko do obliczeń numerycznych. Popatrzmy na kod implementujący sortowanie bąbelkowe:

```
inline void swap (int &a,int &b) {int       tmp=a;a=b;b=tmp;};
void bubble_sort_function (int* data, int N) {
 for(int i = N-1; i>0; --i)
  for(int j=0;j<i;++j)
   if(data[j]>data[j+1])
    swap(data[j],data[j+1]);
}
```

(Źródło: bubble_template.cpp)

Znów widzimy tu dwie pętle i wszystkie uwagi dotyczące funkcji `pow_int` tu też się stosują. Postaramy się więc zdefiniować szablon, który dokona rozwinięcia tych obu pętli. Np. dla `N=3` chcielibyśmy otrzymać następujący kod:

```
//i=2 j=0
if(data[0]>data[1]) swap(data[0],data[1]);
//i=2 j=1
if(data[1]>data[2]) swap(data[1],data[2]);
//i=1 j=0
if(data[0]>data[1]) swap(data[0],data[1]);
```

Jeśli Państwo śledzili wykład (przynajmniej ten), to już Państwo wiedzą, że pierwszym krokiem musi być przepisanie kodu sortowania na postać rekurencyjną:

```
void bubble_sort_function (int* data, int N) {
 for(int j=0;j<N-1;++j)
  if(data[j]>data[j+1])
   swap(data[j],data[j+1]);
 if(N>2)
```

```
  bubble_sort_function(data,N-1);
}
```

To jeszcze nie to co trzeba, bo musimy zapisać pętle w postaci rekurencyjnej. Jeśli oznaczymy sobie:

```
void loop(int * data,int N) {
 for(int j=0;j<N-1;++j)
  if(data[j]>data[j+1])
   swap(data[j],data[j+1]);
}
```

to łatwo zauważyć, że `loop` można zdefiniować następująco:

```
loop(int *data,int N) {
 if(N>0) {
  if(data[0]>data[1]) swap(data[0],data[1]);
  loop(++data,N-1);
 }
}
```

co natychmiast tłumaczy się na szablony:

```
template<int N> inline void loop(int *data) {
 if(data[0]>data[1]) std::swap(data[0],data[1]);
 loop<N-1>(++data);
}
template<> inline void loop<0>(int *data) {};
```

Szablon funkcji `bubble_sort_template` ma więc postać:

```
template<int N>  inline void bubble_sort_template(int * data) {
 loop<N-1>(data);
 bubble_sort_template<N-1>(data);
}
template<>  inline void bubble_sort_template<2>(int * data) {
 loop<1>(data);
};
```

(Źródło: bubble_template.cpp)

Poniżej znów podaję porównanie czasu wykonywania się 100 milionów wywołań funkcji `bubble_sort_function` i `bubble_sort_template` dla tablicy zawierającej 12 liczb całkowitych w kolejności malejącej.

	bubblesortfunction(a,12)	bubblesorttemplate<12>(a)
-O0	43.3	42.2
-O1	21.0	4.8
-O2	20.0	3.5
-O3	20.0	3.6

Widać, że wersja na szablonach jest ok. 4-5 razy szybsza. Zachęcam do własnych eksperymentów. Zaprojektowany szablon działa tylko dla tablic liczb całkowitych. Jest to ewidentne ograniczenie, które powinniśmy zlikwidować poprzez dodanie doatkowego parametru szablonu. Niestety, prowadzi to do konieczności dokonania specjalizacji częściowej, która nie jest dozwolona dla szablonów funkcji. Na szczeście nie jest trudno przepisać naszą implementację używając szablonów klas. Pozostawiam to jako ćwiczenie dla czytelników :).

Rozmiar kodu

Jak pokazałem, kod generowany przez szablon `bubble_sort_template` jest bardziej efektywny. Dzieje się to jednak kosztem jego rozmiaru. Są ku temu dwa powody. Po pierwsze podwójna pętla wewnątrz procedury `bubble_sort_function` wykonuje $(N-1)*(N-2)/2$ iteracji i tyle linijek powinien mieć w pełni rozwinięty kod w szablonie `bubble_sort_template`. Po drugie każda instancja szablonu jest osobną funkcją, stąd `bubble_sort_template<50>` i `bubble_sort_template<51>` generują osobny kod każda. W celu sprawdzenia tych przewidywań przedstawiam poniżej rozmiar wynikowego kodu w bajtach dla programu, który kompilował funkcję `bubble_sort_template<N>`.

N	size
10	9333
30	17846
50	21847
70	40399
90	33296
100	53606

Widać, że choć rozmiar w zasadzie rośnie z N, to ta zależność nie jest nawet monotoniczna. Wynika to pewnie z tego, że dla większych N kompilator nie dokonuje całkowitego rozwiniecią kodu.
