Metaprogramowanie

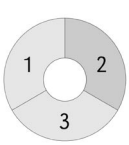
algorytmy wykonywane w czasie kompilacji

Metaprogramowaniem nazywa się tworzenie programów, które w wyniku działania dostarczają programów. Metaprogramy stosujemy aby zwiększyć szybkość działania programów oraz ich czytelność, a także aby unikać powielania kodu, wtedy gdy te same operacje chcemy wykonać dla grupy typów.

Dowiesz się:
- Jak tworzyć programy działające w czasie kompilacji;
- Jak używa się kolekcji typów;
- Jak wykorzystać kontenery i algorytmy dostarczane przez bibliotekę boost::mpl.

Powinieneś wiedzieć:
- Jak pisać proste programy w C++;
- Co to są szablony.

Poziom trudności



Narzędziami do metaprogramowania w C++ jest preprocesor oraz mechanizm szablonów, mechanizmy te dostarczają instrukcji *wyższego rzędu*, które pozwalają na manipulacje kodem źródłowym. Mechanizm szablonów oferuje większe możliwości i wygodniejszą składnię niż makrodefinicje preprocesora, szablony są lepiej wspierane przez kompilator, dlatego są częściej używane do tworzenia metaprogramów i my także będziemy z nich korzystać, pomijając możliwości tworzenia takich rozwiązań przez preprocesor. Szablony umożliwiają implementację dowolnego algorytmu, z punktu widzenia teorii automatów są one równoważne maszynie Turinga. Algorytmy te są wykonywane w czasie kompilacji, wyniki ich działania są umieszczane w kodzie wynikowym.

Zwiększanie czytelności kodu

Jednym z powodów tworzenia metaprogramów jest chęć zwiększenia czytelności i elastyczności kodu. Metaprogram tego typu, pokazany na Listingu 1, oblicza wartość funkcji silnia podczas kompilacji. Konkretyzacja szablonu `Silnia<N>` (gdzie N jest liczbą całkowitą dodatnią) jest równoważna, dla kodu wykonywalnego, dostarczeniu stałej równej N! , ponieważ wartość składowej `value` dla tego szablonu jest obliczana w czasie kompilacji. Szablon `Silnia` nadaje znaczenie stałej, więc kod jest bardziej czytelny, nie musimy obliczać wartości funkcji narzędziem zewnętrznym, unikamy pomyłek związanych z umieszczaniem w programie wartości obliczonych poza nim.

Algorytm obliczający silnię (Listing 1) został zdefiniowany rekurencyjnie i wykorzystuje specjalizację szablonów. Rekurencja jest techniką bardzo powszechną w tego typu programach, ponieważ metaprogramy mogą wykorzystywać jedynie byty dostępne w czasie kompilacji (stałe całkowite, typy itd.), nie możemy użyć zmiennej ani tworzyć pętli. Metaprogramy są podobne do programów tworzonych w językach funkcyjnych. Program pokazany na Listingu 2 (zaczerpnięty z książki Abrahams, Gurtovoy, Język C++, metaprogramowanie za pomocą szablonów) pozwala zapisywać stałe całkowite w kodzie dwójkowym, co zwiększa czytelność, jeżeli to bity są istotne.

Obliczenia w czasie kompilacji

Dostarczając algorytmy, które wykonują się w czasie kompilacji, przyspieszamy działanie programów, ponieważ część przetwarzania będzie wykonywana w czasie kompilacji, co w pewnych przypadkach zwiększa wielkość kodu. Metaprogramy mogą używać fragmentów kodu, a nie tylko stałych, co pokazano na przykładzie szablonu `Power` dostarczającego funkcji potegi całkowitej dla argumentu rzeczywistego. Szablonu tego używa się zamiast funkcji kwadratowej, `Power<2>(x)` dostarcza funkcji `x*x`, `Power<3>(x)` dostarcza `x*x*x` itd. Szablon ten, przedstawiony na Listingu 3, po pierwsze skraca zapis, po drugie kod jest bardziej czytelny, po trzecie kod wykonuje się szybciej niż implementacja algorytmu potęgowania wykonywana w czasie działania programu, dostarczana przez `std::power`, która zawiera pętlę wykonującą wielokrotne mnożenie.

Jeżeli wykładnik jest dużą liczbą całkowitą (co zdarza się na przykład przy kodowaniu RSA), to algorytm potęgowania przez wielokrotne mnożenie, przedstawiony na Listingu 3, jest mało wydajny. Wielkość kodu wynikowego może znacznie wzrosnąć, ponieważ utworzona w czasie kompilacji funkcja jest długa. Niedogodności powyższe możemy usunąć, korzystając z faktu, że w szablonach można implementować złożenia funkcji. Ponieważ x^n, dla $n=2^m$ (gdy n jest potęgą dwójki, tzn. n=1,2,4,8,16, itd.) można obliczać jako m krotne podnoszenie do kwadratu, czyli $x^n=(...((x^2)^2)...)^2$, jeżeli podnoszenie do kwadratu oznaczymy jako sqr, to x^n=sqr × sqr × ... × sqr(x), gdzie × oznacza złożenie funkcji. Używając tego sposobu dla n=1024, wykonamy tylko m=10 operacji podnoszenia do kwadratu, a nie 1024 mnożenia. Potęgę dla dowolnej liczby całkowitej do-

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Niektóre przykłady korzystają z udogodnień dostarczanych przez bibliotekę boost::mpl, warunkiem ich uruchomienia jest instalacja bibliotek boost (w wersji 1.36 lub nowszej) Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła dołączono jako materiały pomocnicze.

datniej n można uzyskać, wykorzystując pokazany powyżej sposób, jeżeli n zapiszemy binarnie $n=b_m*2^m + b_{(m-1)}*2^{(m-1)} + ... + b_1*2 + b_0 = (b_m b_{(m-1)}...b_1 b_0)_2$, to x^n jest iloczynem składników, które będziemy wielokrotnie podnosić do kwadratu, na przykład $x^{35} = x^{(32+2+1)} = ((((x^2)^2)^2)^2)^2*(x^2)*x$. Na Listingu 4 przedstawiono szablon funkcji `Pow`, która realizuje przedstawioną koncepcję, generując w czasie kompilacji wyrażenie, które oblicza potęgę.

Metaprogramy mogą dostarczać przybliżenia funkcji matematycznych z założoną dokładnością. Przykładowo funkcja wykładnicza może być przybliżana szeregiem

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

więc dostarczając szablon, można obliczać ją z założoną precyzją; patrz Listing 5.

Kod tworzony przez metaprogramy często wykorzystuje się w bibliotekach numerycznych, na przykład do mnożenia czy odwracania macierzy. Obliczenia wykonywane w czasie kompilacji dostarczają stałych lub funkcji, które możemy obliczyć lub wyprowadzić za pomocą innych narzędzi (na przykład ręcznie), a następnie umieścić je w programie. Stosowanie metaprogramów zwiększa elastyczność dostarczanych rozwiązań oraz zmniejsza ryzyko popełnienia pomyłki, jest też bardziej czytelne, ponieważ nazwa szablonu zawiera informacje o znaczeniu.

Kolekcje typów

Metaprogramy, których argumentami są typy, pozwalają wyrażać pojęcia trudne do opisu innymi technikami, na przykład pozwala wykonywać podobne operacje na grupie wskazanych typów. W takich przypadkach wykorzystujemy kolekcje typów, które można utworzyć za pomocą szablonów. Przykład takiej kolekcji, przechowującej typy w liście jednokierunkowej (propozycja opisana w książce Andreia Alexandrescu *Nowoczesne projektowanie w C++*), pokazuje Listing 6.

Poszczególne elementy listy przechowują dowolne typy, natomiast ostatni element

Listing 1. *Metaprogram obliczający wartość funkcji silnia*

```
template<unsigned int n> struct Silnia {
   static const unsigned int value = n*Silnia<n-1>::value;
};
template<> struct Silnia<0> { //specjalizacja, kończy rekurencję
   static const unsigned int value = 1;
};
unsigned int i = Silnia<0>::value; //przykład użycia, równoważne i = 1
i = Silnia<5>::value; //równoważne i = 120
```

Listing 2. *Metaprogram, który pozwala używać stałych zapisanych binarnie*

```
template <unsigned long n> struct Binary { //stałe binarne
   static const unsigned long value = Binary<n/10>::value << 1 — n % 10;
};
template <> struct Binary<0> { //stop dla rekurencji
   static const unsigned long value = 0;
};
//przykład użycia
const int FLAGS = Binary<1100>::value; //zamiast 12 lub 0xC
```

Listing 3. *Metaprogram dostarczający funkcji do potęgowania (wykładnik całkowity dodatni)*

```
template <unsigned n> inline double Power(double x) {
   return x * Power<n-1>(x); //rekurencyjna definicja funkcji
}
template <> inline double Power<0>(double x) {
   return 1.0; //specjalizacja kończy rekurencję
}
```

Listing 4. *Szablon generuje wyrażenie obliczające potęgę o wykładniku całkowitym*

```
template <unsigned n> double Pow(double x) {
   return Pow<2>( Pow<n/2>(x) )*Pow<n%2>(x);
}
//specjalizacje dla kwadratu i innych warunków brzegowych
template <> double Pow<2>(double x) { return x*x;}
template <> double Pow<1>(double x) { return x;}
template <> double int_power<0>(double x) { return 1.0;}
```

Listing 5. *Szablon generuje wyrażenie obliczające wartość funkcji wykładniczej z założoną precyzją*

```
template<unsigned int N> inline double Exp(double x) {
   //wykorzystuje szablony z Listingu 1 oraz Listingu 4
   return Exp<N-1>(x) + Pow<N>(x)/Silnia<N>::value;
}
template <> inline double Exp<0>(double x) {
   return 1.0;
}
```

Szablony– przypomnienie

Szablony (*templates*) dostępne w języku C++ umożliwiają implementację generycznych, to znaczy niezależnych od typów, algorytmów oraz struktur danych. Przykładowy szablon `swap`, pokazany poniżej, zamienia zawartość dwu obiektów, możemy go wołać dla dowolnych obiektów tego samego typu, jeżeli dostarczają one konstruktora kopiującego i operatora przypisania.

```
template<typename T> void swap(T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

Podczas kompilacji następuje konkretyzacja szablonu, co oznacza generowanie kodu dla właściwych typów. Kod generowany na podstawie szablonów nie różni się od kodu tworzonego *ręcznie*, nie ma żadnych narzutów pamięciowych i czasowych, jedyną niedogodnością jest dłuższy czas kompilacji. Specjalizacja to wersja szablonu, która będzie użyta do generacji kodu zamiast wersji ogólnej, gdy parametrami będą odpowiednie typy.

jest zawsze typu `NullType` (lista z wartownikiem). Za pomocą tak zdefiniowanej listy możemy utworzyć kolekcję typów o dowolnej długości, na Listingu 6 pokazano listę przechowującą trzy typy: `short`, `int` oraz `long`. Listing 7 pokazuje operację obliczania długości kolekcji (liczba elementów listy).

boost metaprogramming library (MPL)

Tworzenie listy typów za pomocą szablonu `TList` jest niewygodne i nieczytelne, ze względu na konieczność rekurencyjnego zagłębiania tych szablonów. Biblioteka `boost::mpl` dostarcza kolekcji typów oraz operacji, zwalnia ona programistę z konieczności implementacji własnych rozwiązań. Kolekcje typów dostarczane przez tę bibliotekę mają interfejs zbliżony do zestawu metod oferowanych przez kolekcje z biblioteki standardowej, ponadto biblioteka dostarcza zbiór przydatnych metafunkcji, pozwalając unikać zapisów rekurencyjnych. Kolekcje typów to szablony: `mpl::vector`, `mpl::list`, `mpl::set` oraz `mpl::map`, każdy z nich pozwala przechowywać dowolną liczbę typów, różnią się one pewnymi właściwościami, na przykład `mpl::set` usuwa kolejne wystąpienia tego samego typu w kolekcji. Najczęściej używa się kolekcji `mpl::vector`, natomiast pozostałe wykorzystuje się wtedy, gdy odpowiednie właściwości okażą się przydatne. Dostarczane są operacje modyfikacji tych kolekcji, to znaczy dodawania i usuwania elementów, badania właściwości kolekcji, badania występowania danego typu, stosowania pewnej operacji dla każdego z typów w kolekcji, tworzenie nowych typów na podstawie typów przechowywanych w kolekcji. Przykład użycia kolekcji został pokazany na Listingu 8. Każda z operacji zakłada, że wynik, który jest typem, jest składową `type`, natomiast wynik, który jest wartością jest dostępny jako składowa `type::value`. Oczywiście algorytmy wykonują się w czasie kompilacji.

Algorytm `mpl::for_each` jest nietypowy, woła on dostarczoną funkcję lub dostarczony funktor (obiekt klasy, która dostarcza operatora wołania funkcyjnego) dla każdego typu, który jest przechowywany w kolekcji. Specyfika tego algorytmu polega na tym, że tworzy on kod, który jest wykonywany w czasie działania, patrz Listing 9, gdzie pokazano rozwiązanie (a raczej szkic rozwiązania), które wykorzystuje bibliotekę mpl do rejestracji klas w fabryce obiektów.

Biblioteka `mpl` jest używana przez wiele innych bibliotek dostarczanych przez boost, na przykład przez biblioteki tworzące obiekty funkcyjne `bind`, bibliotekę dostarczającą `variant` (obiekt, który przechowuje jedną z wybranych wartości, podobnie jak `union` w C) czy bibliotekę `spirit`, służącą do generowania gramatyk.

Listing 6. *Kolekcja typów zdefiniowana rekurencyjnie*

```
template <class H, class T> struct TList { // lista rekurencyjna
   typedef H Head;
   typedef T Tail;
};
struct NullType { }; //typ kończący listę
//przykładowa kolekcja typów
typedef TList<short, TList<int, TList<long, NullType> > > Integers;
```

Listing 7. *Badanie długości listy typów*

```
template <class TList> struct Size {
   static const int value = Size<typename TList::Tail>::value + 1;
};
template <> struct Size< NullType> {
   static const int value = 0;
};
```

Listing 8. *Przykład operacji na kontenerach z biblioteki mpl*

```
typedef mpl::vector<int,char,long,int> Types; // przykładowa kolekcja
typedef mpl::push_back<Types, int>::type NewTypes; // modyfikacja
cout << mpl::size<NewTypes>::type::value; // wydrukuje wartość 5
//algorytm count zlicza wystąpienia danego typu w kolekcji
cout << mpl::count<Types, int>::type::value; //drukuje liczbę 2
```

Listing 9. *Przykład wykorzystania algorytmu for_each z biblioteki mpl*

```
//mamy hierarchię klas, gdzie klasą bazową jest Figure
class Square : public Figure { //przykładowa klasa konkretna
public:
   //klasa konkretna dostarcza metodę statyczną create
   static Figure* create() { return new Square; }
   static int id_; //klasa konkretna przechowuje identyfikator
};
struct Factory { //fabryka figur konkretnych, przedstawiony kod bardzo uproszczony
public:
   typedef Figure* (*CreateFig)(); //typ funkcji tworzącej obiekty
   static Factory& getInstance(); //dostęp do obiektu fabryki
   int registerFig(CreateFig fun); //metoda rejestracji, dostarcza funkcję
                       tworzącą obiekt, zwraca id
};
struct RegisterFigure { //szablon użyty do rejestracji
   template<typename T> void operator()(T){
      T::id_ = Factory::getInstance().registerFig( T::create );
   }
};
typedef mpl::vector<Square, Circle, Rectangle> Types; // kolekcja typów dla klas
                       konkretnych
mpl::for_each<Types>(RegisterFigure()); //woła w czasie wykonania operację dla
                       każdego typu
```

Podsumowanie

Metaprogramowanie umożliwia przetwarzanie informacji podczas kompilacji. Techniki te pozwalają zmniejszać rozmiar kodu, zwiększając jego czytelność, oraz sprawiać, że programy wykonują się szybciej. Metaprogramowanie znalazło wiele zastosowań zwłaszcza przy tworzeniu ogólnych, przenośnych i efektywnych rozwiązań. Zostanie ono wsparte w nowej wersji standardu C++0x.

W Sieci

- *http://www.boost.org* – dokumentacja bibliotek boost;
- *http://www.open-std.org* – dokumenty opisujące nowy standard C++.

ROBERT NOWAK

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji dla biologii i medycyny, programuje w C++ od ponad 10 lat. Kontakt z autorem:rno@o2.pl