C++17 i STL

Metaprogramowanie I

Metaprogramowanie

- Szablony to generatory służące do wytwarzania klas i funkcji; programowanie szablonowe jest pisaniem programów obliczanych w czasie kompilacji i generujących klasy i funkcje; jest to nazywane **metaprogramowaniem szablonowym**.
- Powody, dla których programiści stosują techniki metaprogramowania:
  - **poprawa bezpieczeństwa typowego:** można dokładnie wyznaczyć typy potrzebne są dla struktury danych lub algorytmu; (można więc wyeliminować wiele przypadków zastosowania jawnej konwersji);
  - **poprawa wydajności wykonywania:** można obliczać wartości w czasie kompilacji i wybierać funkcje do wywołania w czasie wykonywania (nie trzeba więc wykonywać tych obliczeń w czasie wykonywania i można zamienić wiele polimorficznych wywołań na wywołania bezpośrednie).

Metaprogramowanie

- Szablony są bardzo ogólne i zdolne do generowania optymalnego kodu: obsługują arytmetykę, wybieranie i rekurencję.
- W istocie szablony stanowią kompletny funkcyjny język programowania wykonywany w czasie kompilacji.
- Mechanizmy szablonowe języka C++ działające w czasie kompilacji dostarczają czysto funkcyjny język programowania: można tworzyć wartości różnych typów, ale nie ma możliwości posługiwania się zmiennymi, przypisaniami, operatorami inkrementacji itp.

Różnice między programowaniem uogólnionym a metaprogramowaniem

Programowanie uogólnione

- Definiując ogólny typ lub algorytm należy skoncentrować się na definiowaniu wymogów dotyczących argumentów – programowanie uogólnione to przede wszystkim metodologia projektowania.

Metaprogramowanie

- W metaprogramowaniu nacisk kładzie się na obliczenia, przy których wykonywaniu często trzeba dokonywać różnych wyborów oraz stosować jakąś formę iteracji – metaprogramowanie to przede wszystkim zbiór technik implementacyjnych.

Poziomy metaprogramowania

- Brak obliczeń – tylko przekazanie argumentów typowych i wartościowych.
- Proste obliczenia na typach lub wartościach bez wykonywania testów ani iteracji w czasie kompilowania (na przykład operacja && typów logicznych lub dodawanie jednostek).
- Obliczenia przy użyciu jawnych testów wykonywanych w czasie kompilacji (na przykład instrukcja `if` czasu kompilacji).
- Obliczenia przy użyciu iteracji w czasie kompilowania w formie rekurencji.

Przypadki użycia metaprogramowania

- Metaprogram to wykonywane w czasie kompilacji instrukcje, których wynikiem mają być typy lub funkcje przeznaczone do użytku w czasie działania programu.
- Programowanie ogólne najczęściej zalicza się do pierwszej kategorii – brak obliczeń.
- W programowaniu ogólnym skupiamy się na specyfikacji interfejsu, podczas gdy w metaprogramowaniu najważniejsze jest samo programowanie.
- Obliczenia mogą być wykonywane przy użyciu funkcji `constexpr` – wywołanie funkcji `constexpr` ukrywającej metaprogram czy wydobywanie typu z szablonowej funkcji typu.

Funkcje typowe

- **Funkcja typowa** to funkcja, która przyjmuje przynajmniej jeden argument typowy lub zwraca przynajmniej jeden typ jako wynik swojego działania. Na przykład `sizeof(T)` to wbudowana funkcja typowa, która dla argumentu typu `T` zwraca rozmiar obiektu.
- Większość funkcji typowych wcale nie przypomina zwykłych funkcji. Na przykład `is_polymorphic<T>` z biblioteki standardowej przyjmuje argument jako argument szablonowy i zwraca wynik jako składową o nazwie `value`, która może mieć wartość `true` lub `false`.

Funkcje typowe

- W bibliotece standardowej przyjęto konwencję, że funkcja typowa zwracająca typ zwraca go poprzez składową o nazwie `type`. Na przykład:

```
enum class Axis : char { x, y, z };
enum Flags { off, x=1, y=x<<1, z=x<<2, t=x<<3 };
typename std::underlying_type<Axis>::type v1;
    // v1 to char
typename std::underlying_type<Flags>::type v2;
    // v2 to prawdopodobnie int
```

Funkcje typowe

- Funkcja typowa może przyjmować więcej niż jeden argument i zwracać kilka wyników. Na przykład:

```
template<typename T, int N>
struct Array_type {
    using type = T;
    static const int dim = N;
    // …
};
```

- Funkcję `Array_type` można użyć następująco:

```
using Array = Array_type<int,3>;
// …
Array::type x; // x to int
constexpr int s = Array::dim; // s to 3
```

Funkcje typowe

- Funkcje typowe są wykonywane w czasie kompilacji, a więc mogą przyjmować tylko argumenty (typy i wartości) znane już w czasie kompilacji oraz zwracać wyniki (typy i wartości), których można użyć w czasie kompilacji.
- Poniżej znajduje się funkcja typowa zwracająca typ całkowitoliczbowy odpowiedniej liczby bajtów:

```
template<int N>
struct Integer {
    using Error = void;
    using type = Select<N, Error, signed char,
       short, Error, int, Error, Error, Error, long long>;
};
// …
typename Integer<4>::type i4 = 8;
    // 4-bajtowa liczba całkowita
typename Integer<1>::type i1 = 9;
    // 1-bajtowa liczba całkowita
```

Funkcje typowe

- Do wyrażania obliczeń na wartościach wykonywanych w czasie kompilacji zazwyczaj lepsze są funkcje constexpr.
- Funkcje typowe w języku C++ to przeważnie szablony – za ich pomocą można wykonywać bardzo ogólne obliczenia przy użyciu typów i wartości (stanowią one też podstawę metaprogramowania).
- Na przykład zadanie zaalokowania obiektu na stosie, jeśli jest on mały, oraz zaalokowania go w pamięci wolnej w przeciwnym przypadku:

```
constexpr int on_stack_max = sizeof(std::string);
    // maksymalny rozmiar obiektu, jaki zostanie
    // alokowany na stosie
template<typename T>
struct Obj_holder {
    using type = typename std::Conditional<
        (sizeof(T) <= on_stack_max),
        Scoped<T>, // pierwsza możliwość
        On_heap<T> // druga możliwość
    >::type;
};
```

Funkcje typowe

- Szablonu `Obj_holder` można użyć następująco:

```
void f()
{
    typename Obj_holder<double>::type v1;
        // liczba typu double zostaje na stosie
    typename Obj_holder<array<double ,200>>::type v2;
        // tablica idzie do pamięci wolnej
    //…
    *v1 = 7.7; // dostęp uzyskuje się poprzez wskaźniki
        // (v1 zawiera wartość typu double)
    (*v2)[77] = 9.9; // On_heap zapewnia dostęp poprzez
        // wskaźniki (v2 zawiera tablicę)
}
```

Funkcje typowe

- Do implementacji szablonów `Scoped` i `On_heap` nie trzeba stosować technik metaprogramowania:

```
template<typename T>
struct On_heap {
    On_heap() : p(new T){} // alokuje
    ~On_heap() { delete p; } // dealokuje
    T& operator*() { return *p; }
    T* operator->() { return p; }
    On_heap(const On_heap&) = delete;
        // uniemożliwia kopiowanie
    On_heap& operator=(const On_heap&) = delete;
private:
    T* p; // wskaźnik do obiektu w pamięci wolnej
};
```

Funkcje typowe

- `On_heap` i `Scoped` to dobre przykłady tego, jak programowanie ogólne i metaprogramowanie zmuszają programistę do opracowania jednolitego interfejsu do różnych implementacji danego ogólnego pomysłu (w tym przypadku jest nim alokacja obiektu):

```
template<typename T>
struct Scoped {
    Scoped() {}
    T& operator*() { return x; }
    T* operator->() { return &x; }
    Scoped(const Scoped&) = delete;
        // uniemożliwia kopiowanie
    Scoped& operator=(const Scoped&) = delete;
private:
    T x; // obiekt
};
```

Aliasy typów

- Podczas używania `typename` i `::type` do sprawdzenia typu składowej uwidaczniają się szczegóły implementacyjne szablonu `Obj_holder`.
- Szczegóły implementacyjne `::type` możemy ukryć przy użyciu aliasu szablonu i sprawić, by funkcja typowa bardziej przypominała wyglądem funkcję zwracającą typ.
- Przykład:

```
template<typename T>
using Holder = typename Obj_holder<T>::type;
// …
Holder<double> v1;
    // double idzie na stos
Holder<array<double ,200>> v2;
    // tablica idzie do pamięci wolnej
// …
*v1 = 7.7; // dostęp uzyskuje się poprzez wskaźniki
    // (v1 zawiera wartość typu double)
(*v2)[77] = 9.9; // On_heap zapewnia dostęp poprzez
    // wskaźniki (v2 zawiera tablicę)
```

Predykaty typów

- Predykat to funkcja zwracająca wartość logiczną – jeśli planuje się pisać funkcje przyjmujące typy jako argumenty, to naturalną będzie możliwość zadawania pytań na temat typów tych argumentów.
- Przykład:

```
template<typename T>
void copy(T* p, const T* q, int n)
{
    if (std::is_pod<T>::value) memcpy(p, q, n*sizeof(T));
        // użycie zoptymalizowanego kopiowania pamięci
    else for (int i = 0; i != n; ++i) p[i] = q[i];
        // kopiowanie pojedynczych wartości
}
```

- Predykat `is_pod` z biblioteki standardowej sprawdza, czy typ jest zwykły, czy wymaga osobnego kopiowania.

Predykaty typów

- Tak jak w przypadku składowej `::type`, posługiwanie się wartością `::value` jest żmudne i powoduje ujawnienie szczegółów implementacyjnych – funkcja zwracająca wartość typu `bool` powinna być wywoływana przy użyciu operatora `()`:

```
template<typename T>
void copy(T* p, const T* q, int n)
{
    if (is_pod<T>())
    // …
}
```

- Standard pozwala na to w przypadku wszystkich predykatów typów z biblioteki standardowej – w bibliotece standardowej znajduje się wiele gotowych takich predykatów, przykładowo: `is_integral`, `is_pointer`, `is_empty`, `is_polymorphic` oraz `is_move_assignable`.

Wybieranie funkcji

- Obiekt funkcyjny jest obiektem pewnego typu, a więc w celu wyboru funkcji można też używać technik wyboru typów i wartości.
- Na przykład:

```
struct X {
    void operator()(int x) { /* … */ }
    //…
};
struct Y {
    void operator()(int y) { /* … */ }
    //…
};
// …
Conditional<(sizeof(int)>4),X,Y>{}(7);
    // tworzy obiekt typu X lub Y i go wywołuje
// …
using Z = Conditional<(Is_polymorphic<X>()),X,Y>;
Z zz; // tworzy X lub Y
zz(7); // wywołuje X lub Y
```

Cechy / trejty

- W bibliotece standardowej powszechnie wykorzystywane są cechy (albo trejty), które wiążą typy z ich właściwościami.
- Strukturę cechującą można traktować jak funkcję typową zwracającą wiele wyników albo jak zbiór funkcji typowych.
- W bibliotece standardowej znajdują się struktury cechujące `allocator_traits`, `char_traits`, `iterator_traits`, `regex_traits` i `pointer_traits` oraz dodatkowo konstrukcje `time_traits` i `type_traits`, które w rzeczywistości są prostymi funkcjami typowymi.

Cechy / trejty

- Na przykład właściwości iteratora są zdefiniowane w strukturze cechującej iterator_traits:

```
template<typename Iterator>
struct iterator_traits
{
    using difference_type =
        typename Iterator::difference_type;
    using value_type =
        typename Iterator::value_type;
    using pointer =
        typename Iterator::pointer;
    using reference =
        typename Iterator::reference;
    using iterator_category =
        typename Iterator::iterator_category;
};
```

Cechy / trejty

- Mając daną strukturę `iterator_traits` dla wskaźnika, możemy posługiwać się składowymi `value_type` i `difference_type` tego wskaźnika, mimo że wskaźniki nie mają składowych:

```
template<typename Iter>
Iter search(Iter p, Iter q,
    typename iterator_traits<Iter>::value_type val)
{
    typename iterator_traits<Iter>::difference_type
        m = q-p;
    // …
}
```