C++17 i STL

Metaprogramowanie II

Struktury sterujące

- Funkcje typowe `Conditional` i `Select` zwracają typy (jak musisz wybrać jedną z paru wartości, wystarczy Ci operator `?:`).
- `Conditional`: umożliwia wybór jednego z dwóch typów (alias `std::conditional`);
- `Select`: umożliwia wybór jednego z kilku typów.
- Szablon `conditional` należy do biblioteki standardowej i znajduje się w nagłówku `<type_traits>`.

Struktury sterujące – instrukcja warunkowa

- Implementacja szablonu `conditional`:

```
// ogólny szablon
template<bool C, typename T, typename F>
struct conditional {
    using type = T;
};
// specjalizacja dla false
template<typename T, typename F>
struct conditional<false,T,F> {
    using type = F;
};
```

- Specjalizacja umożliwia oddzielenie ogólnego przypadku od jednego lub większej liczby specjalnych przypadków.

Struktury sterujące – instrukcja warunkowa

- Przykład użycia szablonu `conditional`:

```
typename conditional<
        (std::is_polymorphic<T>::value), X, Y
    >::type Z;
```

- Wybory takie są w całości dokonywane w czasie kompilacji, więc nie powodują żadnego narzutu w czasie działania programu.
- Aby poprawić składnię, można zrobić alias typu:

```
template<bool B, typename T, typename F>
using Conditional =
    typename std::conditional<B, T, F>::type;
```

- Teraz można napisać:

```
Conditional<(is_polymorphic<T>::value), X, Y> Z;
```

Struktury sterujące – instrukcja warunkowa

- Przykład:

```
struct Square {
    constexpr int operator()(int i)
    { return i * i; }
};
struct Cube {
    constexpr int operator()(int i)
    { return i * i * i; }
};
```

- Wybieramy typ, tworzymy domyślny obiekt tego typu oraz go wywołujemy:

```
Conditional<(My_cond<T>()), Square, Cube>{}(99);
```

Struktury sterujące – instrukcja wyboru

- Wybieranie spośród N możliwości jest bardzo podobne do wybierania jednej z dwóch.
- Ogólna wersja struktury `select` zdefiniowana przy użyciu szablonów ze zmienną liczbą parametrów:

```
// przypadek ogólny, nigdy nie konkretyzowany
template<unsigned N, typename... Cases>
struct select;
// przypadek zredukowany
template<unsigned N, typename T, typename... Cases>
struct select<N,T,Cases...> :select<N-1, Cases...>
{};
// ostateczny przypadek dla N==0
template<typename T, typename... Cases>
struct select<0,T,Cases...>
{ using type = T; };
```

- Aby poprawić składnię, można zrobić alias typu:

```
template<unsigned N, typename... Cases>
using Select =
    typename select<N,Cases...>::type;
```

Struktury sterujące – iteracja i rekurencja

- Podstawowe techniki obliczania wartości w czasie kompilacji można przedstawić na przykładzie funkcji obliczającej silnię:

```
template<int N>
constexpr int fac() {
    return N * fac<N-1>();
}
template<>
constexpr int fac<1>() {
    return 1;
}
```

- Przykład użycia:

```
constexpr int x5 = fac<5>();
```

- W tym przykładzie funkcja silni została zaimplementowana przy użyciu rekurencji, nie zaś pętli – ma to sens, ponieważ nie dysponujemy zmiennymi w czasie kompilacji.

Struktury sterujące – iteracja i rekurencja

- W rozważanym przypadku obliczenia można wykonać też w bardziej konwencjonalny sposób:

```
constexpr int fac(int i) {
    return (i<2)?1:i*fac(i-1);
}
```

- Przykład użycia:

```
constexpr int x6 = fac(6);
```

- Wersja nieszablonowa jest minimalnie łatwiejsza w obsłudze dla kompilatora, ale wydajność obu wersji w czasie wykonywania jest oczywiście identyczna.

Struktury sterujące – rekurencja przy użyciu klas

- Iteracje obejmujące bardziej skomplikowane stany lub parametryzacje można obsłużyć przy użyciu klas.
- Na przykład program obliczający silnię można zaimplementować tak:

```
template<int N>
struct Fac {
    static const int value = N*Fac<N-1>::value;
};
template<>
struct Fac<1> {
    static const int value = 1;
};
```

- Przykład użycia:

```
constexpr int x7 = Fac<7>::value;
```

Kiedy stosować metaprogramowanie?

- Przy użyciu przedstawionych wcześniej struktur sterujących można wykonać wszelkie obliczenia w czasie kompilacji (w zakresie dozwolonym przez limity translacji) – ale po co to robić?
  - Technik tych należy używać, jeśli pozwalają uzyskać bardziej przejrzysty, wydajniejszy i łatwiejszy w utrzymaniu kod.
  - Jedną z najbardziej oczywistych wad metaprogramowania jest to, że kod oparty na skomplikowanych szablonach może być trudny do zrozumienia i jeszcze trudniejszy do diagnozowania.
  - Poza tym skomplikowane szablony mogą spowalniać proces kompilacji.

Kiedy stosować metaprogramowanie?

- Metaprogramowanie szablonowe przyciąga inteligentnych programistów:
  - metaprogramowanie umożliwia osiągnięcie nieosiągalnego w inny sposób poziomu bezpieczeństwa typowego i wydajności – jeśli zyski są znaczne, a kod pozostaje zrozumiały;
  - można wykazać się, jakim jest się inteligentnym – to oczywiście nie jest wystarczający powód do stosowania tych technik.

Definicja warunkowa

- Funkcja typowa `enable_if` zdefiniowana jest w pliku nagłówkowym `<type_traits>`.
- W celu uproszczenia notacji definiujemy alias:

```
template<bool B, typename T =void>
using Enable_if =
    typename std::enable_if<B, T>::type;
```

- Jeśli warunek konstrukcji `Enable_if` ma wartość `true`, to jej wynikiem jest drugi argument `T`. A jeśli warunek ten ma wartość `false`, następuje zignorowanie całej deklaracji funkcji.

Definicja warunkowa

- Przykład zastosowania konstrukcji `Enable_if` – chcemy warunkowo dostarczyć operator `->` gdy pracujemy z klasą, jako parametrem sprytnego wskaźnika:

```
template<typename T>
constexpr bool Is_class() {
    return std::is_class<T>::value;
}
template<typename T>
class Smart_pointer {
    //…
    // zwraca referencję do całego obiektu
    T& operator*();
    // wybiera składową (tylko dla klas)
    template<typename U = T> // zasada SFINAE
    Enable_if<Is_class<T>(), T>* operator->();
    //…
};
```

Definicja warunkowa

- Mając definicję `Smart_pointer` z użyciem `Enable_if`, otrzymujemy:

```
void f(
  Smart_pointer<double> p,
  Smart_pointer<complex<double>> q
) {
    auto d0 =*p; // OK.
    auto c0 =*q; // OK.
    auto d1 = q->real(); // OK.
    auto d2 = p->real(); // błąd:
        //nie wskazuje obiektu klasy
    //…
}
```

Definicja warunkowa

- Użycie `Enable_if` do oznaczenia typu zwrotnego sprawia, że konstrukcja ta znajduje się na froncie, w widocznym miejscu, do którego logicznie należy, ponieważ ma wpływ na całą deklarację (nie tylko na typ zwrotny).
- Przykład użycia:

```
template<typename T>
class vector<T> {
public:
    // n elementów typu T o wartości val
    vector(size_t n, const T& val);
    template<
      typename Iter,
      typename =Enable_if<Input_iterator<Iter>()>
    >
    vector(Iter b, Iter e); // inicjacja z <b,e)
    //…
};
```

- Ten nieużywany domyślny argument szablonu zostanie skonkretyzowany, ponieważ z pewnością nie da się wydedukować nieużywanego parametru szablonu – to oznacza, że deklaracja `vector(Iter,Iter)` nie powiedzie się, chyba że `Iter` będzie typu `Input_iterator`.

Definicja warunkowa

- Techniki z użyciem `Enable_if` działają tylko dla szablonów funkcji (wliczając funkcje składowe szablonów klas i specjalizacji).
- Implementacja i sposób użycia `Enable_if` zależą od reguł przeciążania szablonów funkcji – w konsekwencji nie można tej konstrukcji używać do kontrolowania deklaracji klas, zmiennych ani funkcji nieszablonowych.

Definicja warunkowa

- Implementacja `Enable_if` jest bardzo prosta:

```
template<bool B, typename T = void>
struct std::enable_if {
    typedef T type;
};
template<typename T>
struct std::enable_if<false, T> {};
    // brak ::type, jeśli B==false
template<bool B, typename T = void>
using Enable_if =
    typename std::enable_if<B,T>::type;
```

- Zwróć uwagę, że można opuścić argument typowy i dostać domyślnie `void`.