



# C++17 i STL

Metaprogramowanie



# Literatura



- B.Stroustrup: *Język C++. Kompendium wiedzy. Wydanie 4. Rozdział 28: Metaprogramowanie*. Wydawnictwo Helion, Gliwice 2014.
- P.Białas, W.Palacz: *Zaawansowane CPP / Wykład 8: Metaprogramowanie*. Warszawa, 2006.  
[https://wazniak.mimuw.edu.pl/index.php?title=Zaawansowane CPP/Wyk%C5%82ad 8: Metaprogramowanie](https://wazniak.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_8:_Metaprogramowanie)
- Honey the codewitch: *Metaprogramming in C++: A Gentle Introduction*. USA, 2021.  
<https://www.codeproject.com/Articles/5297814/Metaprogramming-in-Cplusplus-A-Gentle-Introduction>



# Metaprogramowanie




- Szablony to generatory służące do wytwarzania klas i funkcji; programowanie szablone jest pisaniem programów obliczanych w czasie kompilacji i generujących klasy i funkcje; jest to nazywane **metaprogramowaniem szablonym**.
- Powody, dla których programiści stosują techniki metaprogramowania:
  - **poprawa bezpieczeństwa typowego:** można dokładnie wyznaczyć typy potrzebne 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 wykonania 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 szablone 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 implementacyjne 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.

# Przykład metaprogramu

- Zadanie polega na obliczeniu potęgi liczby 3. Ponieważ w programowaniu za pomocą szablonów musimy posługiwać się rekurencją, to zaczynamy od sformułowania problemu w sposób rekurencyjny:  
 $3^N = 3 * 3^{N-1}$  dla  $N > 0$   
 $3^0 = 1$

- Za pomocą szablonów implementujemy to tak:

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

- Teraz wyrażenie:  
`auto i = Pow3<4>::val;`  
jest obliczane w czasie kompilacji i efekt jest taki sam jak:  
`auto i = 81;`



# Przykład metaprogramu

- Można też zastosować szablon funkcji:

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

- Teraz wyrażenie:  
`auto i = Pow3<4>();`  
jest obliczane w czasie kompilacji i efekt jest taki sam jak:

```
auto i = 81;
```

- 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 };
};
```

- Tutaj już nie można wykorzystać szablonu funkcji, bo nie zezwala on na specjalizację częściową.



# 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;
    // ...
};
```

- ▶ Szablonu `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 (!!! Poprawić przykład)

- ▶ 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 {
private:
    T* p; // wskaźnik do obiektu w pamięci wolnej
public:
    On_heap() : p(new T){} // alokuje
    ~On_heap() { delete p; } // dealokuje
    T& operator*() { return *p; }
    T* operator->() { return p; }
    // blokada kopiowania
    On_heap(const On_heap&) = delete;
    On_heap& operator=(const On_heap&) = delete;
};
```



# 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 {
private:
    T x; // obiekt
public:
    Scoped() {}
    T& operator*() { return x; }
    T* operator->() { return &x; }
    // blokada kopiowania
    Scoped(const Scoped&) = delete;
    Scoped& operator=(const Scoped&) = delete;
};
```

# 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

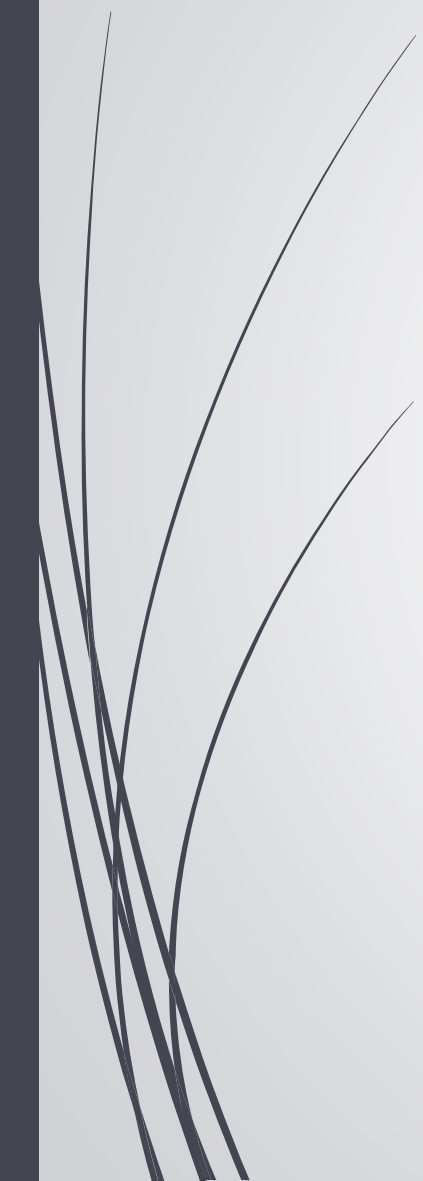
- ▶ Mając daną strukturę `iterator_traits` dla wskaźnika, możemy posługiwać się składowymi `value_type` i `difference_type` tego wskaźnika, mimo że wskaźniki nie mają składowych:

```
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;
    // ...
}
```





# Struktury sterujące



- ▶ Funkcje typowe `Conditional` i `Select` zwracają typy (jak musisz wybrać jedną z pary 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<0>() {
    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<0> {
    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 zwrótnego 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 zwrótny).

- 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`.