C++

Podróż po języku dla zaawansowanych

Wydanie III

Bjarne Stroustrup

Helion

Tytuł oryginału: A Tour of C++ (C++ In-Depth Series), 3rd Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-8322-525-8

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/cppoz3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

16

Narzędzia pomocnicze

Czas, którego marnowanie sprawia ci przyjemność,
nie jest czasem zmarnowanym.
— Bertrand Russell

- Wprowadzenie
- Czas
  Zegary. Kalendarze. Strefy czasowe
- Adaptacja funkcji
  Lambdy jako adaptery. `mem_fn()`. `function`
- Funkcje typów
  Predykaty typów. Własności warunkowe. Generatory typów. Typy powiązane
- `source_location()`
- `move()` i `forward()`
- Manipulacja bitami
- Zamykanie programu
- Porady

16.1 Wprowadzenie

Przypisanie komponentu biblioteki do grupy „Narzędzia” niewiele mówi użytkownikom. Oczywiście każdy komponent biblioteki jest w jakimś sensie narzędziem. W tym rozdziale opisuję elementy biblioteki, które mają kluczowe znaczenie w wielu sytuacjach, ale nie pasowały nigdzie indziej. Wiele z nich stanowi element budowy bardziej zaawansowanych narzędzi biblioteki, w tym innych komponentów.

16.2 Czas

Wszystko, co jest potrzebne do pracy z czasem, znajduje się w nagłówku `<chrono>` biblioteki standardowej:

- Zegary, `time_point` i `duration` do pomiaru czasu trwania różnych czynności oraz jako podstawa budowy wszystkiego, co wiąże się z czasem.
- `day`, `month`, `year` oraz `weekdays` do przedstawiania punktów w czasie `time_point` w kontekście życia codziennego.
- `time_zone` i `zoned_time` do pracy z różnicami w czasie w różnych częściach globu.

Zasadniczo każdy większy system wykorzystuje niektóre z tych narzędzi.

16.2.1 Zegary

Oto prosty przykład mierzenia czasu wykonywania jakichś czynności:

```
using namespace std::chrono; // w podrzędnej przestrzeni nazw std::chrono (zobacz 3.3)

auto t0 = system_clock::now();
do_work();
auto t1 = system _clock::now();

cout << t1-t0 << "\n";        // domyślna jednostka: 20223[1/00000000]s
cout << duration_cast<milliseconds>(t1-t0).count() << " ms\n"; // określona jednostka: 2 ms
cout << duration_cast<nanoseconds>(t1-t0).count() << " ns\n";  // określona jednostka: 2022300ns
```

Zegar zwraca `time_point` (punkt w czasie). Wynikiem odejmowania dwóch obiektów `time_point` jest czas trwania (okres). Domyślny operator `<<` typu `duration` dodaje jednostkę w formie przyrostka. Różne zegary zwracają wyniki w różnych jednostkach czasu, „tyknięciach zegara” (użyty przeze mnie mierzy czas w setkach nanosekund), dlatego często powinno się przekonwertować otrzymaną wartość na odpowiednią jednostkę. Do tego służy `duration_cast`.

Zegary umożliwiają wykonywanie szybkich pomiarów. Nigdy nie wypowiadaj się na temat „wydajności” kodu, dopóki nie przeprowadzisz pomiarów. Bardzo rzadko udaje się trafnie zgadnąć w tej kwestii. Szybkie i proste pomiary są lepsze niż ich brak, ale wydajność nowoczesnych komputerów to skomplikowany temat i dlatego nie należy zbytnio się przywiązywać do wyników paru prostych testów. Testy zawsze wykonuj wielokrotnie, aby zmniejszyć ryzyko odchylenia wyniku przez jakieś rzadkie zdarzenie lub efekt działania pamięci podręcznej.

Przestrzeń nazw `std::chrono_literals` zawiera definicje przyrostków jednostek czasu (6.6). Na przykład:

```
this_thread::sleep_for(10ms+33us); // czekaj 10 milisekund i 33 mikrosekundy
```

Konwencjonalne nazwy symboliczne wyraźnie zwiększają czytelność i ułatwiają utrzymanie kodu.

16.2.2 Kalendarze

Podczas pracy z codziennymi wydarzeniami rzadko wykorzystujemy milisekundy. Najczęściej korzystamy z jednostek obejmujących lata, miesiące, dni, godziny, minuty, sekundy oraz dni tygodnia. Biblioteka standardowa obsługuje je wszystkie. Na przykład:

```
auto spring_day = April/7/2018;
cout << weekday(spring_day) << '\n';            // Sat
cout << format("{:%A}\n",weekday(spring_day));  // Saturday
```

W moim komputerze `Sat` jest domyślną reprezentacją znakową soboty. Nie podobał mi się ten skrót, więc użyłem formatu (11.6.2), aby uzyskać pełną nazwę dnia. Z niejasnych powodów `%A` oznacza „wydrukuj pełną nazwę dnia tygodnia”. Oczywiście `April` to miesiąc (kwiecień), a dokładniej mówiąc `std::chrono::Month`. Możemy też napisać:

```
auto spring_day = 2018y/April/7;
```

Przyrostek y odróżnia lata od zwykłych liczb całkowitych, które są używane w odniesieniu do dni miesiąca ponumerowanych od 1 do 31.

Istnieje możliwość zdefiniowania nieprawidłowej daty. W razie wątpliwości można użyć funkcji `ok()` do sprawdzenia:

```
auto bad_day = January/0/2024;
if (!bad_day.ok())
    cout << bad_day << " is not a valid day\n";
```

Oczywiście funkcja `ok()` jest najbardziej przydatna do sprawdzania dat uzyskanych w wyniku obliczeń.

Daty są składane dzięki przeciążeniu operatora / (ukośnik) przez typy `year`, `month` i `int`. Powstały w wyniku typ `Year_month_day` ma operacje konwersji na `time_point` i w drugą stronę, co umożliwia szybkie i precyzyjne wykonywanie operacji na datach. Na przykład:

```
sys_days t = sys_days{February/25/2022}; // punkt w czasie z precyzją dni
t += days{7};                            // tydzień po 25 lutego 2022 r.
auto d = year_month_day(t);              // konwersja punktu w czasie z powrotem na kalendarz

cout << d << '\n';
// 2022-03-04
cout << format("{:%B}/{}/{}\n", d.month(), d.day(), d.year()); // Marzec/04/2022
```

Te obliczenia wymagają zmiany miesiąca i wiedzy na temat lat przestępnych. Domyślnie implementacja podaje datę w standardowym formacie ISO 8601. Aby otrzymać nazwę miesiąca w postaci słowa „Marzec”, musimy rozbić datę na poszczególne pola i zastosować detale formatujące (11.6.2). Z niejasnych powodów `%B` oznacza „wydrukuj pełną nazwę miesiąca”.

Takie operacje często można wykonywać w czasie kompilacji, dzięki czemu są one zaskakująco szybkie:

```
static_assert(weekday(April/7/2018) == Saturday); // true
```

Kalendarze są skomplikowane i subtelne. To typowe i odpowiednie dla „systemów” przeznaczonych dla „zwykłych ludzi” na przestrzeni wieków, a nie dla programistów, którzy chcieliby uprościć

swoją pracę. System kalendarzy biblioteki standardowej może być (i jest) rozszerzany o inne kalendarze, takie jak juliański, islamski, tajski itd.

16.2.3 Strefy czasowe

Jednym z największych wyzwań w pracy z czasem jest poprawne posługiwanie się strefami czasowymi. Trudno je zapamiętać, ponieważ są arbitralne oraz zmieniają się na wiele różnych sposobów, które nie są ustandaryzowane w ujęciu całej planety. Na przykład:

```
auto tp = system_clock::now(); // tp to time_point
cout << tp << '\n';                                    // 2021-11-27 21:36:08.2085095

zoned_time ztp { current_zone(),tp };                  // 2021-11-27 16:36:08.2085095 EST
cout << ztp << '\n';

const time_zone est {"Europe/Copenhagen"};
cout << zoned_time{ &est,tp } << '\n';                  // 2021-11-27 22:36:08.2085095 GMT+1
```

Obiekt typu `time_zone` reprezentuje czas względny w odniesieniu do standardu (zwanego GMT lub UTC) używanego przez `system_clock`. Biblioteka standardowa przeprowadza synchronizację z globalną bazą danych (IANA), aby zwracać poprawne wyniki. Ta synchronizacja może odbywać się automatycznie w systemie operacyjnym lub pod kontrolą administratora systemu. Nazwy stref czasowych są łańcuchami w stylu języka C w formie „kontynent/duże miasto”, np. `"America/New_York"`, `"Asia/Tokyo"`, `"Africa/Nairobi"`. Obiekt typu `zoned_time` to `time_zone` wraz z `time_point`.

Strefy czasowe, tak jak kalendarze, odnoszą się do kwestii, które powinniśmy pozostawić bibliotece standardowej zamiast stosować własnoręcznie przygotowane rozwiązania. Na przykład: o której godzinie ostatniego dnia lutego 2024 r. w Nowym Jorku zmieni się data w New Delhi? Kiedy skończył się czas letni w Denver w amerykańskim stanie Kolorado w 2020 r.? Kiedy będzie następny rok przestępny? Biblioteka standardowa to wszystko „wie”.

16.3 Adaptacja funkcji

Gdy funkcja jest przekazywana jako argument funkcji, typ tego argumentu musi dokładnie odpowiadać wymaganiom określonym w deklaracji funkcji wywoływanej. Jeśli dany argument tylko „prawie spełnia wymagania”, programista ma do wyboru różne rozwiązania:

- użyć lambdy (16.3.1);
- za pomocą funkcji `std::mem_fn()` utworzyć obiekt funkcyjny z funkcji składowej (16.3.2);
- zdefiniować funkcję tak, aby przyjmowała `std::function` (16.3.3).

Jest jeszcze wiele innych sposobów, ale zazwyczaj jedna z powyższych metod okazuje się najlepsza.

16.3.1 Lambdy jako adaptery

Spójrz na klasyczny przykład rysowania wszystkich kształtów:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),[](Shape* p) { p->draw(); });
}
```

Tak jak wszystkie algorytmy z biblioteki standardowej, `for_each()` wywołuje swój argument przy użyciu tradycyjnej składni wywołania funkcji `f(x)`, ale funkcja `draw()` klasy `Shape` stosuje tradycyjną notację obiektową `x->f()`. Lambda bez problemu przełącza się między tymi dwiema notacjami.

16.3.2 mem_fn()

Dla danej funkcji składowej adapter funkcji `mem_fn(mf)` tworzy obiekt funkcyjny, który można wywoływać jako funkcję niebędącą składową klasy. Na przykład:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),mem_fn(&Shape::draw));
}
```

Zanim w C++11 wprowadzono lambdy, funkcja `mem_fn()` i jej odpowiedniki stanowiły podstawowy mechanizm zamiany obiektowej metody wywoływania na funkcyjną.

16.3.3 function

Typ `function` z biblioteki standardowej może przechowywać dowolny obiekt, który można wywołać za pomocą operatora wywołania `()`. Innymi słowy, obiekt typu `function` jest obiektem funkcyjnym (6.3.2). Na przykład:

```
int f1(double);
function<int(double)> fct1 {f1};            // inicjalizacja do f1

int f2(string);
function fct2 {f2};                         // typ fct2 to function<int(string)>

function fct3 = [](Shape* p) { p->draw(); }; // typ fct3 to function<void(Shape*)>
```

W przypadku `fct2` typ funkcji pozostawiłem do dedukcji z inicjalizatora — `int(string)`.

Oczywiście obiekty typu `function` są przydatne przy tworzeniu wywołań zwrotnych, przekazywaniu operacji w argumentach, przekazywaniu obiektów funkcyjnych itd. To jednak w porównaniu z bezpośrednimi wywołaniami może powodować pewien narzut. W szczególności, w przypadku obiektu `function`, którego rozmiar nie jest obliczany w czasie kompilacji, alokacja w pamięci wolnej może fatalnie wpłynąć na działanie aplikacji, w której krytyczne znaczenie ma wydajność. Rozwiązanie pojawi się w C++23 w postaci `move_only_function`.

Innym problemem jest to, że funkcja, będąc obiektem, nie uczestniczy w przeciążaniu. Jeśli chcesz przeciążać obiekty funkcyjne (także lambdy), możesz użyć typu `overloaded` (15.4.1).

16.4 Funkcje typów

Funkcja typu to taka funkcja, której wartość jest określana w czasie kompilacji, przyjmująca typ jako argument lub zwracająca typ. Biblioteka standardowa zawiera wiele funkcji typu, dzięki którym implementatorzy bibliotek (a także inni programiści) mogą pisać kod wykorzystujący aspekty języka, biblioteki standardowej i ogólnie kodu.

W odniesieniu do typów numerycznych szerokie spektrum informacji dostarcza `numeric_limits` z nagłówka `<limits>` (17.7). Na przykład:

```
constexpr float min = numeric_limits<float>::min(); // najmniejsza dodatnia liczba
                                                    // zmiennoprzecinkowa
```

Natomiast rozmiar obiektu można sprawdzić za pomocą wbudowanego operatora `sizeof` (1.4). Na przykład:

```
constexpr int szi = sizeof(int); // liczba bajtów w int
```

W nagłówku `<type_traits>` biblioteki standardowej znajduje się wiele funkcji do sprawdzania właściwości typów. Na przykład:

```
bool b = is_arithmetic_v<X>;      // prawda, jeśli X jest jednym z wbudowanych typów arytmetycznych
using Res = invoke_result_t<decltype(f)>; // Res to int, jeśli f jest funkcją zwracającą int
```

Konstrukcja `decltype(f)` to wywołanie wbudowanej funkcji typu `decltype()`, która zwraca zadeklarowany typ swojego argumentu — tutaj `f`.

Niektóre funkcje typu tworzą nowe typy na podstawie danych wejściowych. Na przykład:

```
typename<typename T>
using Store = conditional_t(sizeof(T)<max, On_stack<T>, On_heap<T>);
```

Jeśli pierwszym (logicznym) argumentem funkcji `conditional_t` jest `true`, to wynikiem jest pierwsza z alternatywnych opcji. W przeciwnym razie — druga. Zakładając, że `On_stack` i `On_heap` udostępniają te same funkcje dostępu do `T`, mogą alokować swoje `T` zgodnie z tym, co sugerują ich nazwy. Dzięki temu użytkowników `Store<X>` można dostosować odpowiednio do rozmiaru `X` obiektów. Korzyści w zakresie wydajności wynikające z tego wyboru alokacji mogą być znaczne. To jest prosty przykład tego, jak możemy tworzyć nowe funkcje typu na podstawie standardowych lub przy użyciu koncepcji.

Koncepcje są funkcjami typu. Używane w wyrażeniach są predykatami typów. Na przykład:

```
template<typename F, typename... Args>
auto call(F f, Args... a, Allocator alloc)
{
    if constexpr (invocable<F,alloc,Args...>) // potrzebny alokator?
        return f(f,alloc,a...);
    else
        return f(f,a...);
}
```

W wielu przypadkach koncepcje są najlepszymi funkcjami typu, ale większość biblioteki standardowej została napisana przed wprowadzeniem koncepcji i musi uwzględniać kod, w którym nie są one używane.

Konwencje notacyjne są mało przejrzyste. W bibliotece standardowej funkcje typu zwracające wartości są oznaczone przyrostkiem _v, a funkcje typu zwracające typy są oznaczone przyrostkiem _t. Jest to pozostałość po słabym typizowaniu języka C i czasach sprzed wprowadzenia koncepcji w C++. Żadna funkcja typu biblioteki standardowej nie zwraca i typu, i wartości, więc te przyrostki są zbędne. Dzięki koncepcjom, zarówno w bibliotece standardowej, jak i gdziekolwiek indziej, żaden przyrostek nie jest potrzebny ani używany.

Funkcje typu są częścią mechanizmu języka C++ do wykonywania obliczeń w czasie kompilacji umożliwiającego ściślejszą kontrolę typów i pozwalającego uzyskać większą wydajność niż ta, którą można byłoby osiągnąć bez niego. Posługiwanie się funkcjami typu i koncepcjami (rozdział 8., podrozdział 14.5) często nazywa się **metaprogramowaniem** lub (kiedy wykorzystywane są szablony) **metaprogramowaniem szablonowym**.

16.4.1 Predykaty typów

W nagłówku <type_traits> biblioteki standardowej znajduje się wiele prostych funkcji zwanych **predykatami typów** (ang. *type predicate*), które dostarczają podstawowe informacje o typach. W poniższej tabeli znajduje się opis kilku z nich.

Jednym z tradycyjnych zastosowań tych predykatów jest ograniczanie argumentów szablonów. Na przykład:

```
template<typename Scalar>
class complex {
    Scalar re, im;
public:
    static_assert(is_arithmetic_v<Scalar>, "Niestety obsługuję tylko liczby zespolone.");
    // ...
};
```

To jednak — podobnie jak inne tradycyjne zastosowania — można wyrazić prościej i bardziej elegancko za pomocą koncepcji:

```
template<Arithmetic Scalar>
class complex {
    Scalar re, im;
public:
    // ...
};
```

W wielu przypadkach predykaty typów takie jak `is_arithmetic` dla ułatwienia zostają zaszyte w definicjach koncepcji. Na przykład:

```
template<typename T>
concept Arithmetic = is_arithmetic_v<T>;
```

Co ciekawe nie istnieje koncepcja `std::arithmetic`.

Często możemy definiować koncepcje, które są bardziej ogólne niż predykaty typów z biblioteki standardowej. Wiele z nich odnosi się tylko do typów wbudowanych. Programista może zdefiniować koncepcję w odniesieniu do wymaganych operacji, co sugeruje definicja koncepcji Number (8.2.4):

Wybrane predykaty typów T, A i U są typami; wszystkie predykaty zwracają wartość logiczną	
`is_void_v<T>`	Czy T jest void?
`is_integral_v<T>`	Czy T jest typu całkowitoliczbowego?
`is_floating_point_v<T>`	Czy T jest typu zmiennoprzecinkowego?
`is_class v<T>`	Czy T jest klasą (a nie unią)?
`is_function_v<T>`	Czy T jest funkcją (a nie obiektem funkcyjnym lub wskaźnikiem do funkcji)?
`is_arithmetic_v<T>`	Czy T jest typem całkowitoliczbowym lub zmiennoprzecinkowym?
`is_scalar_v<T>`	Czy T jest typem arytmetycznym, wyliczeniem, wskaźnikiem lub wskaźnikiem do składowej?
`is_constructible_v<T, A...>`	Czy T można utworzyć z listy argumentów A...?
`is_default_constructible_v<T>`	Czy T można utworzyć bez jawnych argumentów?
`is_copy_constructible_v<T>`	Czy T można utworzyć z innego T?
`is_move_constructible_v<T>`	Czy T można przenieść lub skopiować do innego T?
`is_assignable_v<T,U>`	Czy U można przypisać do T?
`is_trivially_copyable_v<T,U>`	Czy U można przypisać do T bez operacji kopiowania zdefiniowanych przez użytkownika?
`is_same_v<T,U>`	Czy T jest tym samym typem co U?
`is_base_of_v<T,U>`	Czy U pochodzi od T lub czy U jest tym samym typem co U?
`is_convertible_v<T,U>`	Czy T można niejawnie przekonwertować na U?
`is_iterator_v<T>`	Czy T jest typem iteratora?
`is_invocable_v<T, A...>`	Czy T można wywołać z listą argumentów A...?
`has_virtual_destructor_v<T>`	Czy T ma destruktor wirtualny?

```
template<typename T, typename U = T>
concept Arithmetic = Number<T,U> && Number<U,T>;
```

Predykaty typów biblioteki standardowej najczęściej można spotkać głęboko w implementacji podstawowych usług, gdzie często wskazują przypadki optymalizacji. Na przykład część implementacji funkcji `std::copy(Iter,Iter,Iter2)` mogłaby optymalizować ważny przypadek związany z ciągłymi sekwencjami obiektów typów prostych, takich jak liczby całkowite:

```
template<class T>
void cpy1(T* first, T* last, T* target)
{
    if constexpr (is_trivially_copyable_v<T>)
        memcpy(first, target, (last - first) * sizeof(T));
    else
        while (first != last) *target++ = *first++;
}
```

Ta prosta optymalizacja bije swój niezoptymalizowany wariant o około 50% w niektórych implementacjach. Nie angażuj się w takie sprawy, dopóki nie sprawdzisz, czy standard nie robi czegoś lepiej. Ręcznie optymalizowany kod jest zazwyczaj trudniejszy w utrzymaniu od prostszych alternatyw.

16.4.2 Właściwości warunkowe

Spójrz na poniższą definicję „inteligentnego wskaźnika”:

```
template<typename T>
class Smart_pointer {
    T& operator*() const;
    T* operator->() const; // -> powinien działać tylko wtedy, gdy T jest klasą
};
```

Operator `->` powinien być zdefiniowany tylko wtedy, gdy `T` jest typem klasowym. Na przykład `Smart_pointer<vector<T>>` powinien mieć `->`, a `Smart_pointer<int>` — nie. Nie możemy użyć instrukcji `if` kompilacji, ponieważ nie jesteśmy wewnątrz funkcji. Dlatego piszemy:

```
template<typename T>
class Smart_pointer {
    //...
    T& operator*() const;
    T* operator->() const requires is_classv<T>; // -> zdefiniowany tylko wtedy, gdy T jest klasą
};
```

Predykat typu bezpośrednio wyraża ograniczenie dotyczące funkcji `operator->()`. W tym przypadku można by było użyć także koncepcji. W bibliotece standardowej nie ma koncepcji reprezentującej wymóg, że typ musi być klasowy (tzn. musi być klasą, strukturą lub unią), ale możemy taką zdefiniować:

```
template<typename T>
concept Class = is_class v<T> || is_union_v<T>;              // unie są klasami

template<typename T>
class Smart_pointer {
    // ...
    T& operator*() const;
    T* operator->() const requires Class<T>; // -> jest zdefiniowany tylko, jeśli T jest klasą lub unią
};
```

Koncepcja często jest bardziej ogólna albo po prostu bardziej odpowiednia niż bezpośrednie użycie predykatu typu z biblioteki standardowej.

16.4.3 Generatory typów

Wiele funkcji typu zwraca typy, często nowe, które zostały przez nie obliczone. Takie funkcje nazywam **generatorami typów**, aby odróżnić je od predykatów typów. W standardzie jest ich kilka. Poniższa tabela przedstawia kilka przykładowych.

Wybrane generatory typów	
`R=remove_const_t<T>`	`R` to typ `T` z usuniętą ewentualną deklaracją `const`
`R=add_const_t<T>`	`R` to `const T`
`R=remove_reference_t<T>`	Jeśli `T` jest referencją `U&`, to `R` odpowiada `U`, a w przeciwnym razie `R` odpowiada `T`
`R=add_lvalue_reference_t<T>`	Jeśli `T` jest referencją l-wartościową, to `R` odpowiada `T`, a w przeciwnym razie `R` odpowiada `T&`
`R=add_rvalue_reference_t<T>`	Jeśli `T` jest referencją r-wartościową, to `R` odpowiada `T`, a w przeciwnym razie `R` odpowiada `T&&`
`R=enable_if_t<b,T =void>`	Jeśli `b` jest prawdą, to `R` odpowiada `T`, a w przeciwnym razie `R` jest niezdefiniowane
`R=conditional_t<b,T,U>`	`R` odpowiada `T`, jeśli `b` jest prawdą, a w przeciwnym razie `R` odpowiada `U`
`R=common_type_t<T...>`	Jeśli istnieje typ, na który można niejawnie przekonwertować wszystkie `T`, to `R` jest tym typem, a w przeciwnym razie `R` jest niezdefiniowane
`R=underlying_type_t<T>`	Jeśli `T` jest wyliczeniem, to `R` jest jego podstawowym typem, a w przeciwnym razie występuje błąd
`R=invoke_result_t<T,A...>`	Jeśli `T` można wywołać z argumentami `A...`, to `R` jest typem zwrotnym, a w przeciwnym razie występuje błąd

Te funkcje typów są zazwyczaj używane w implementacjach narzędzi, a nie w zwykłym kodzie aplikacji. Z nich wszystkich `enable_if` występuje chyba najczęściej w kodzie napisanym przed wprowadzeniem koncepcji. Na przykład warunkowo włączany operator `->` dla inteligentnego wskaźnika tradycyjnie jest implementowany mniej więcej tak:

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*();
    enable_if<is_class v<T>,T&> operator->();       // -> jest zdefiniowany tylko, jeśli T jest klasą
};
```

Niezbyt przyjemnie się to czyta, a bardziej skomplikowane przypadki są jeszcze gorsze. Definicja `enable_if` bazuje na subtelnej cesze języka o nazwie **SFINAE** (ang. *Substitution Failure Is Not An Error* — nieudane podstawienie nie jest błędem). Poszukaj informacji na ten temat, jeśli ich potrzebujesz.

16.4.4 Typy powiązane

Wszystkie kontenery biblioteki standardowej (12.8) i wszystkie kontenery zdefiniowane na ich wzór mają pewne **typy powiązane**, takie jak ich typy wartości i iteratorów. W nagłówkach `<iterator>` i `<ranges>` biblioteki standardowej znajdują się następujące nazwy:

Wybrane generatory typów	
`range_value_t<R>`	Typ elementów zakresu `R`
`iter_value_t<T>`	Typ elementów wskazywanych przez iterator `T`
`iterator_t<R>`	Typ iteratora zakresu `R`

16.5 source_location

Drukując na wyjściu zawartość stosu lub komunikat o błędzie, często chcemy dodać informację na temat lokalizacji w kodzie źródłowym. Do tego służy klasa `source_location` z biblioteki:

```
const source_location loc =  source_location::current();
```

Funkcja `current()` zwraca obiekt klasy `source_location` wskazujący miejsce w kodzie źródłowym, w którym się znajduje. Klasa `source_location` ma funkcje składowe `file()` i `function_name()`, które zwracają łańcuchy w stylu C, oraz funkcje składowe `line()` i `column()`, które zwracają liczby całkowite bez znaku.

Wywołując je w funkcji, możemy utworzyć całkiem przydatny komunikat dziennika:

```
void log(const string& mess = "", const source_location loc =  source_location::current())
{
    cout << loc.file_name()
        << '(' << loc.line() << ':' << loc.column() << ") "
        << loc.function_name() ":  "
        << mess;
}
```

Wywołanie funkcji `current()` jest domyślnym argumentem, dzięki czemu otrzymujemy lokalizację wywołującego funkcji `log()`, a nie lokalizację samej funkcji `log()`:

```
void foo()
{
    log("Hello");   // myfile.cpp (17,4) foo: Hello
    // ...
}

int bar(const string& label)
{
    log(label);     // myfile.cpp (23,4) bar: <<wartość label>>
    // ...
}
```

W kodzie napisanym przed powstaniem C++20 lub przeznaczonym do kompilacji w starszych kompilatorach w takiej sytuacji używane byłyby makra `__FILE__` i `__LINE__`.

16.6 move() i forward()

Wybór między przenoszeniem i kopiowaniem zwykle odbywa się zakulisowo (3.6). Kompilator wybiera przenoszenie, kiedy dany obiekt ma być zniszczony (jak przy zwracaniu), ponieważ ta operacja jest uważana za prostszą i mniej czasochłonną. Czasami jednak programista musi wyrazić swój zamiar wprost. Na przykład wskaźnik `unique_ptr` jest jedynym właścicielem obiektu, w związku z czym nie może być kopiowany. Jeśli więc jest potrzebny w innym miejscu, należy go przenieść. Na przykład:

```
void f1()
{
    auto p = make_unique<int>(2);
    auto q = p;         // błąd: nie można skopiować wskaźnika unique_ptr
    auto q = move(p); // teraz p zawiera nullptr
    // ...
}
```

Co zaskakujące, funkcja `std::move()` niczego nie przenosi, tylko rzutuje swój argument na referencję r-wartości, zaznaczając w ten sposób, że jej argument nie będzie już używany, a więc może zostać przeniesiony (6.2.2). Powinna zatem nazywać się `rvalue_cast`. Ma ona zastosowanie w kilku ważnych przypadkach. Weźmy na przykład prostą zamianę:

```
template <typename T>
void swap(T& a, T& b)
{
    T tmp {move(a)}; // konstruktor T „widzi" r-wartość i dokonuje przeniesienia
    a = move(b);     // przypisanie T „widzi" r-wartość i dokonuje przeniesienia
    b = move(tmp);   // przypisanie T „widzi" r-wartość i dokonuje przeniesienia
}
```

Nie powinniśmy wielokrotnie przenosić potencjalnie dużych obiektów i dlatego żądamy przenoszenia za pomocą funkcji `std::move()`.

Tak jak w przypadku innych rodzajów rzutowania, funkcja `std::move()` ma pewne kuszące, choć niebezpieczne zastosowania. Na przykład:

```
string s1 = "Witaj,";
string s2 = "świecie";
vector<string> v;
v.push_back(s1);         // argument const string&; push_back() wykona kopię
v.push_back(move(s2)); // użycie konstruktora przenoszącego
v.emplace_back(s1);      // alternatywa; umieszcza kopię s1 na nowej pozycji końcowej v (12.8)
```

W tym przykładzie obiekt `s1` zostaje skopiowany (przez funkcję `push_back()`), podczas gdy `s2` zostaje przeniesiony. Czasami (tylko czasami) może to przyspieszyć działanie funkcji `push_back()` w takiej sytuacji jak z `s2`. Sęk w tym, że obiekt, z którego nastąpiło przeniesienie, jest pozostawiany. Jeśli ponownie użyjemy `s2`, będziemy mieć kłopoty:

```
cout << s1[2]; // drukuje 't'
cout << s2[2]; // awaria?
```

Moim zdaniem to zastosowanie funkcji `std::move()` jest zbyt ryzykowne, aby było powszechne. Nie korzystaj z tej metody, dopóki nie wykażesz, że daje ona wyraźne korzyści pod względem wydajności. Później przez przypadek ktoś może użyć obiektu, z którego nastąpiło przeniesienie.

Kompilator „wie", że wartość zwrotna nie jest ponownie używana w funkcji, więc wywoływanie wprost funkcji `std::move()`, np. `return std::move(x)`, jest zbędne, a nawet może utrudniać optymalizację.

Stan obiektu, z którego nastąpiło przeniesienie, jest generalnie nieokreślony, ale wszystkie typy biblioteki standardowej pozostawiają takie obiekty w stanie, w którym można je zniszczyć lub coś do nich przypisać. Byłoby niemądrze nie podążyć tym tropem. W przypadku kontenera (np. `vector` lub `string`) obiekt, z którego nastąpiło przeniesienie, będzie „pusty". W przypadku wielu typów domyślna wartość jest dobrym stanem określającym pustkę: ma odpowiednie znaczenie i nie zajmuje dużo zasobów.

Ważnym zastosowaniem przenoszenia jest dalsze przekazywanie argumentów (8.4.2). Czasami trzeba przesłać zbiór argumentów do innej funkcji, nic w nich nie zmieniając (aby osiągnąć „doskonałe przekazywanie"):

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    return unique_ptr<T>{new T{std::forward<Args>(args)...}}; // przekazuje każdy argument
}
```

Funkcja `forward()` z biblioteki standardowej różni się od prostszej funkcji `std::move()` tym, że prawidłowo radzi sobie z zawiłościami dotyczącymi r-wartości i l-wartości (6.2.2). Funkcji `std::forward()` należy używać wyłącznie do przekazywania i niczego nie powinno się przekazywać więcej niż raz. Przekazany dalej obiekt nie należy już do nas.

16.7 Manipulowanie bitami

W nagłówku `<bit>` znajdują się funkcje służące do niskopoziomowej pracy z bitami. Jest to specjalistyczny, ale często niezbędny rodzaj czynności. Podczas pracy na poziomie bliskim sprzętu zwykle mamy do czynienia z bitami, zmieniamy ich ułożenie w bajtach i słowach oraz zamieniamy surową pamięć na typizowane obiekty. Na przykład typ `bit_cast` umożliwia konwersję wartości jednego typu na inny typ o takim samym rozmiarze:

```
double val = 7.2;
auto x = bit_cast<uint64_t>(val);    // pobiera bitową reprezentację 64-bitowej liczby zmiennoprzecinkowej
auto y = bit_cast<uint64_t>(&val);   // pobiera bitową reprezentację 64-bitowego wskaźnika

struct Word { std::byte b[8]; };
std::byte buffer[1024];
// ...
auto p = bit_cast<Word*>(&buffer[i]); // p wskazuje na 8 bajtów
auto i = bit_cast<int64_t>(*p);       // konwersja tych 8 bajtów na liczbę całkowitą
```

Typ `std::byte` (przedrostek `std::` jest konieczny) z biblioteki standardowej reprezentuje bajty, ale nie jako reprezentacje znaków lub liczb całkowitych. Ma tylko bitowe operacje logiczne, więc

nie obsługuje operacji arytmetycznych. Najlepsze typy do wykonywania operacji bitowych to w większości przypadków liczby całkowite bez znaku i `std::byte`. Pisząc „najlepsze", mam na myśli najszybsze i powodujące najmniej niespodzianek. Na przykład:

```
void use(unsigned int ui)
{
    int x0 = bit_width(ui)              // najmniejsza liczba bitów potrzebna do reprezentacji ui
    unsigned int ui2 = rotl(ui,8)       // obrót w lewo o 8 bitów (uwaga: nie zmienia ui)
    int x1 = popcount(ui);              // liczba jedynek w ui
    // ...
}
```

Zobacz także typ `bitset` (15.3.2).

16.8 Zamykanie programu

Czasami program napotyka problemy, z którymi nie jest w stanie sobie poradzić:

- Jeśli dany rodzaj błędu występuje często i bezpośredni moduł wywołujący powinien go obsłużyć, należy zwracać jakiś kod zwrotu (4.4).
- Jeśli dany rodzaj błędu występuje rzadko lub bezpośredni moduł wywołujący nie powinien go obsłużyć, należy zgłosić wyjątek (4.4).
- Jeśli błąd jest tak poważny, że żadna zwykła część programu nie jest w stanie go obsłużyć, należy zamknąć program.

Biblioteka standardowa zawiera narzędzia potrzebne w ostatnim z wymienionych przypadków (zamykanie programu):

- `exit(x)` — wywołuje funkcje zarejestrowane w funkcji `atexit()`, a następnie zamyka program z wartością zwrotną `x`. Jeśli masz taką potrzebę, poszukaj informacji o funkcji `atexit()` — jest to zasadniczo prosty mechanizm destrukcji dzielony z językiem C.
- `abort()` — zamyka program natychmiast i bezwarunkowo z wartością zwrotną oznaczającą nieudane zakończenie. Niektóre systemy operacyjne udostępniają narzędzia, które wymuszają modyfikację tego prostego wyjaśnienia.
- `quick_exit(x)` — wywołuje funkcje zarejestrowane w funkcji `at_quick_exit()`, a następnie zamyka program z wartością zwrotną `x`.
- `terminate()` — wywołuje procedurę `terminate_handler`. Domyślna procedura `terminate_handler` to `abort()`.

Te funkcje są przeznaczone do obsługi poważnych błędów. Nie wywołują destruktorów, tzn. nie wykonują zwykłych czynności porządkowych. Różne procedury obsługi pozwalają wykonywać pewne czynności przed zamknięciem programu. Muszą one być bardzo proste, ponieważ jedną z przyczyn ich wywołania jest to, że program znalazł się w nieprawidłowym stanie. Jedną z sensownych i całkiem popularnych czynności jest „restart systemu do dobrze zdefiniowanego stanu bez obecnego programu". Inną, odrobinę bardziej ryzykowną, ale też niepozbawioną sensu czynnością jest „zapisanie wiadomości o błędzie w dzienniku i zamknięcie programu". Problem

z zapisaniem wiadomości w dzienniku polega na tym, że system wejścia-wyjścia może nie działać z tego samego powodu, z którego funkcja zamknięcia programu została wywołana.

Obsługa błędów to jeden z najtrudniejszych rodzajów programowania. Nawet czyste zamknięcie programu może być niełatwe.

Żadna biblioteka ogólnego przeznaczenia nie powinna zamykać programu bezwarunkowo.

16.9 Porady

[1] Biblioteka nie musi być duża ani skomplikowana, aby była przydatna (16.1).
[2] Mierz czas wykonywania swoich programów, zanim cokolwiek stwierdzisz na temat ich wydajności (16.2.1).
[3] Używaj `duration_cast`, aby zwracać wyniki pomiarów czasu wykonywania we właściwej jednostce (16.2.1).
[4] Aby zaprezentować datę bezpośrednio w kodzie źródłowym, użyj notacji symbolicznej (np. `November/28/2021`) (16.2.2).
[5] Jeśli data jest wynikiem obliczeń, sprawdź jej poprawność za pomocą funkcji `ok()` (16.2.2).
[6] Do pracy z wartościami czasowymi w różnych miejscach używaj `zoned_time` (16.2.3).
[7] Do wyrażania drobnych zmian w konwencjach wywoływania używaj lambdy (16.3.1).
[8] Używaj funkcji `mem_fn()` lub lambd do tworzenia obiektów funkcyjnych mogących wywoływać funkcję składową po wywołaniu przy użyciu tradycyjnej notacji wywoływania funkcji (16.3.1, 16.3.2).
[9] Używaj `function`, gdy chcesz zapisać coś, co można wywołać (16.3.3).
[10] Używaj koncepcji zamiast bezpośrednio używać predykatów typów (16.4.1).
[11] Możesz pisać kod wprost zależny od właściwości typów (16.4.1, 16.4.2).
[12] Preferuj koncepcje zamiast cech i `enable_if` zawsze, gdy to możliwe (16.4.3).
[13] Używaj klasy `source_location` do wstawiania informacji o miejscu w kodzie źródłowym do komunikatów o błędach i dziennikowych (16.5).
[14] Unikaj używania wprost funkcji `std::move()` (16.6) [CG: ES.56].
[15] Funkcji `std::forward()` używaj wyłącznie do przekazywania (16.6).
[16] Nigdy nie odczytuj obiektu po wywołaniu na nim funkcji `std::move()` lub `std::forward()` (16.6).
[17] Używaj typu `std::byte` do reprezentowania danych, które (jeszcze) nie mają określonego typu (16.7).
[18] Do pracy z bitami używaj liczb całkowitych bez znaku lub obiektów typu `bitset` (16.7).
[19] Zwracaj kod błędu z funkcji, jeśli jej bezpośredni wywołujący powinien zająć się problemem (16.8).
[20] Zgłaszaj wyjątek przez funkcję, jeśli jej bezpośredni wywołujący nie powinien zająć się problemem (16.8).
[21] Wywołuj funkcje `exit()`, `quick_exit()` lub `terminate()`, aby zamknąć program po wystąpieniu problemu, którego nie da się rozwiązać (16.8).
[22] Żadna biblioteka ogólnego przeznaczenia nie powinna bezwarunkowo zamykać programu (16.8).

S

Skorowidz

Są dwa rodzaje wiedzy:
kiedy posiadamy wiedzę w jakimś przedmiocie
lub kiedy wiemy,
gdzie znaleźć potrzebne informacje.
— Samuel Johnson

H

I

J

K

P

R

S

Ś

T

U

W

Z

Oto nowoczesny C++: poznaj jego ukryte możliwości!

C++ zmienił się nie do poznania: jest nowocześniejszy, bardziej precyzyjny, pozwala też na pisanie zwięzłego i efektywnego kodu. Programista ma dokładniejszą kontrolę nad działaniem programu, co przekłada się na szybkość pracy i lepsze wykorzystywanie zasobów sprzętowych. Dodatkowo ekosystem C++ oferuje mnóstwo bibliotek, narzędzi czy środowisk programistycznych. Aby jednak pisać w nim dobry kod, trzeba sprawnie poruszać się po świecie C++.

To drugie wydanie zwięzłego przewodnika po C++ dla doświadczonych programistów, zaktualizowane do standardu C++20. Dzięki niemu zaznajomisz się z najważniejszymi elementami języka i biblioteki standardowej, koniecznymi do efektywnej pracy w takich stylach programowania jak programowanie zorientowane obiektowo czy generyczne. W tym wydaniu opisano wiele nowych elementów dodanych w C++20, w tym moduły, koncepcje, współprocedury i zakresy. Omówiono też wybrane komponenty biblioteki, które pojawią się dopiero w standardzie C++23.

Jeśli jesteś programistą C lub C++ i zależy Ci, by lepiej poznać najnowsze możliwości języka C++, albo biegle posługujesz się innym językiem programowania i chcesz ogólnie zaznajomić się z zaletami nowoczesnego C++ — nie znajdziesz bardziej zwięzłego i prostszego przewodnika niż ten.

W książce między innymi:

- nowe możliwości języka w standardzie C++20
- moduły, klasy i obsługa błędów
- operacje, zarządzanie zasobami i wejście-wyjście
- generatory, potoki, kontenery
- współbieżność i wielozadaniowość

Dr Bjarne Stroustrup jest duńskim informatykiem, twórcą języka C++ i autorem wielu książek. Obecnie pełni funkcję dyrektora działu technologicznego banku Morgan Stanley w Nowym Jorku, jest też profesorem wizytującym na Columbia University. Laureat wielu nagród, członek National Academy of Engineering, IEEE Fellow, CHM Fellow i Churchill College Cambridge Fellow.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-525-8
9 788383 225258
Cena: 79,00 zł
Pearson
Addison-Wesley