Jacek Galowicz

C++17 STL Receptury

Helion

Packt›

Tytuł oryginału: C++17 STL Cookbook

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-4501-0

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cpp17r
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1

Nowe funkcje w C++17

W tym rozdziale omówię następujące receptury:

- Użycie strukturalnych wiązań do rozpakowania wartości zwrotnej
- Ograniczanie zasięgu zmiennej do konstrukcji `if` i `switch`
- Zalety stosowania nowych reguł inicjalizacji z użyciem składni opartej na nawiasach
- Umożliwienie konstruktorowi automatycznego określenia typu klasy szablonu
- Użycie wyrażenia `constexpr-if` do uproszczenia decyzji podejmowanych podczas kompilacji
- Włączenie bibliotek w postaci samych nagłówków z użyciem osadzonych zmiennych
- Implementowanie za pomocą wyrażeń fold przydatnych funkcji pomocniczych

Wprowadzenie

Język C++ zyskał wiele usprawnień wraz z wydaniem standardów C++11, C++14 i C++17. Obecnie jest to zupełnie inny język w porównaniu do tego, którym był jeszcze dekadę temu. Standard C++ nie dotyczy jedynie języka, który musi być obsługiwany przez kompilatory, ale również standardową bibliotekę szablonów (ang. *standard template library* — STL).

Na podstawie wielu przykładów przedstawionych w tej książce dowiesz się, jak najlepiej wykorzystać bibliotekę STL. W tym rozdziale skoncentruję się na najważniejszych nowych funkcjach wprowadzonych w najnowszym wydaniu C++. Ich opanowanie bardzo pomoże w tworzeniu ekspresyjnego, łatwego w odczycie i późniejszej konserwacji kodu.

Zobaczysz, jak za pomocą strukturalnych wiązań komfortowo uzyskać dostęp do poszczególnych elementów składowych par, krotek i struktur, a także jak ograniczyć zasięg zmiennej, wykorzystując do tego nowe możliwości w zakresie inicjalizacji zmiennej w konstrukcjach `if` i `switch`. Syntaktyczna dwuznaczność wprowadzona przez standard C++11 wraz z nową składnią inicjalizacji z użyciem nawiasu wyszukującą ten sam inicjalizator dla list została poprawiona dzięki *nowym regułom inicjalizacji z użyciem składni opartej na nawiasach*. Konkretny *typ* egzemplarza klasy szablonu może być *ustalony* na podstawie rzeczywistych argumentów konstruktora. Jeżeli poszczególne specjalizacje szablonu klasy będą powodowały powstawanie zupełnie odmiennego kodu, teraz można to łatwo wyrazić za pomocą wyrażenia `constexpr-if`. Obsługa parametrów wariadycznych w funkcjach szablonu stała się w wielu przypadkach znacznie łatwiejsza dzięki pojawieniu się nowych *wyrażeń fold*. Wreszcie znacznie bardziej komfortowe jest definiowanie statycznych, dostępnych globalnie obiektów w bibliotekach w postaci jedynie nagłówków z użyciem nowych możliwości deklarowania osadzonych zmiennych, co wcześniej było możliwe jedynie w funkcjach.

Część przykładów w tym rozdziale może być bardziej interesująca dla programistów opracowujących biblioteki niż zajmujących się tworzeniem aplikacji. Część wymienionych funkcji przedstawiłem tutaj dla porządku, nie musisz dokładnie zrozumieć wszystkich przykładów w tym rozdziale, aby móc opanować materiał zaprezentowany w pozostałej części książki.

Użycie strukturalnych wiązań do rozpakowania wartości zwrotnej

Standard C++17 oferuje nową funkcję łączącą w sobie syntaktyczny lukier z automatycznym ustaleniem typu: **strukturalne wiązanie**. Pomaga ono w przypisywaniu poszczególnym zmiennym wartości pochodzących z par, krotek i struktur. W innych językach programowania to zadanie nosi nazwę **rozpakowania**.

Jak to zrobić?

Zastosowanie strukturalnego wiązania w celu przypisania wielu wartości pochodzących z większej struktury zawsze ma postać pojedynczego kroku. Najpierw zobacz, jak taka operacja odbywała się przed wprowadzeniem standardu C++17. Następnie przejdę do kilku przykładów pokazujących, jak można to zrobić w C++17.

- Zaczynam od uzyskania dostępu do poszczególnych wartości w `std::pair`. Przyjmuję założenie o istnieniu funkcji matematycznej, `divide_remainder()`, która akceptuje *dzielną* (parametr `dividend`) i *dzielnik* (parametr `divisor`) oraz zwraca wynik dzielenia liczb wraz z resztą. Wspomniane wartości są zwracane za pomocą algorytmu `std::pair`:

```
std::pair<int, int> divide_remainder(int dividend, int divisor);
```

Spójrz na sposób, w jaki uzyskujemy dostęp do poszczególnych wartości znajdujących się w wyniku działania funkcji:

```
const auto result (divide_remainder(16, 3));
        std::cout << "16 / 3 to "
                  << result.first << " plus reszta z dzielenia "
                  << result.second << '\n';
```

Zamiast stosować podejście przedstawione powyżej poszczególne wartości można przypisać zmiennym za pomocą ekspresyjnych nazw, które są znacznie łatwiejsze w odczycie. Spójrz na ten fragment kodu:

```
auto [fraction, remainder] = divide_remainder(16, 3);
std::cout << "16 / 3 to "
          << fraction << " plus reszta z dzielenia "
          << remainder << '\n';
```

- Strukturalne wiązanie działa również z algorytmem `std::tuple`. Przyjmujemy założenie o istnieniu przedstawionej poniżej funkcji, która zwraca pobrane z internetu informacje o kursie akcji:

```
std::tuple<std::string,
           std::chrono::system_clock::time_point, unsigned>
stock_info(const std::string &name);
```

Przypisanie wyniku poszczególnym zmiennym odbywa się dokładnie tak samo jak w poprzednim przykładzie:

```
const auto [name, valid_time, price] = stock_info("INTC");
```

- Strukturalne wiązanie sprawdza się również w przypadku pracy z niestandardowymi strukturami. Przyjmuję założenie o istnieniu następującej struktury:

```
struct employee {
    unsigned id;
    std::string name;
    std::string role;
    unsigned salary;
};
```

Dostęp do poszczególnych elementów składowych można uzyskać za pomocą strukturalnego wiązania. Alternatywne podejście polega na użyciu pętli, przy założeniu o istnieniu wektora, jak pokazałem tutaj:

```
int main()
{
    std::vector<employee> employees {
        /* Inicjalizacja odbyła się w innym miejscu */};
    for (const auto &[id, name, role, salary] : employees) {
        std::cout << "Nazwisko: "   << name
                  << "Stanowisko: "   << role
                  << "Wynagrodzenie: " << salary << '\n';
    }
}
```

Jak to działa?

Strukturalne wiązanie jest zawsze stosowane z użyciem następującego wzorca:

```
auto [zmienna1, zmienna2, ...] = <wyrażenie pary, krotki, struktury lub tablicy>;
```

- Lista zmiennych, `zmienna1`, `zmienna2` itd., musi dokładnie odpowiadać liczbie zmiennych znajdujących się w wyrażeniu, z którego będą pochodziły przypisywane wartości.
- Wyrażenie `<pary, krotki, struktury lub tablicy>` musi mieć jedną z następujących postaci:
  - Para: `std::pair`.
  - Krotka: `std::tuple`.
  - Struktura. Żaden element składowy *nie może* być statyczny, a wszystkie muszą być zdefiniowane w tej *samej klasie bazowej*. Pierwszy zadeklarowany element składowy będzie przypisany pierwszej zmiennej, drugi element składowy — drugiej zmiennej itd.
  - Tablica o stałej wielkości.
- Typem może być `auto`, `const auto`, `const auto&` lub nawet `auto&&`.

> Ze względu między innymi na *wydajność* staraj się podczas używania odwołań zminimalizować liczbę niepotrzebnych operacji kopiowania.

Jeżeli podasz zbyt *dużą* lub *małą* liczbę zmiennych w nawiasie kwadratowym, wówczas kompilator wygeneruje komunikat błędu:

```
std::tuple<int, float, long> tup {1, 2.0, 3};
auto [a, b] = tup; // To polecenie nie działa
```

Powyższe polecenie próbuje skopiować dane z trzyelementowej krotki do jedynie dwóch zmiennych. Kompilator natychmiast wychwyci ten błąd i wygeneruje odpowiedni komunikat:

```
error: type 'std::tuple<int, float, long>' decomposes into 3 elements, but
only 2 names were provided
auto [a, b] = tup;
```

Co dalej?

Za pomocą strukturalnych wiązań można natychmiast uzyskać dostęp do wielu podstawowych struktur danych STL bez konieczności zmiany czegokolwiek. Na przykład tutaj przedstawiłem pętlę, która wyświetla wszystkie elementy zdefiniowane w egzemplarzu `std::map`:

```
std::map<std::string, size_t> animal_population {
    {"ludzi",   7000000000},
    {"kurczaków", 17863376000},
```

```
    {"wielbłądów",   24246291},
    {"owiec",    1086881528},
    /* … */
};

for (const auto &[species, count] : animal_population) {
    std::cout << "Mamy " << count << " " << żyjących
              << " na naszej planecie.\n";
}
```

Ten konkretny przykład działa, ponieważ w trakcie iteracji kontenera `std::map` powstaje węzeł `std::pair<const typ_klucza, typ_wartości>` dla każdego kroku iteracji. Dokładniej mówiąc, te węzły są rozpakowywane za pomocą funkcji wiązania — `typ_klucza` to ciąg tekstowy będący nazwą populacji (`string`), natomiast `typ_wartości` to wielkość populacji podana jako `size_t` — w celu uzyskania do nich dostępu w treści pętli.

Przed wprowadzeniem standardu C++17 podobny efekt można było otrzymać po zastosowaniu wywołania `std::tie()`:

```
int remainder;
std::tie(std::ignore, remainder) = divide_remainder(16, 5);
std::cout << "16 % 5 to " << remainder << '\n';
```

Przykład ten pokazuje, jak parę wynikową rozpakować na postać dwóch zmiennych. Wywołanie `std::tie()` oferuje znacznie mniejsze możliwości niż strukturalne wiązanie w tym sensie, że wszystkie zmienne, do których będą dołączane wartości, trzeba *wcześniej* zdefiniować. Jednak z drugiej strony przykład pokazuje zaletę wywołania `std::tie()` *nieoferowaną* przez strukturalne wiązanie: wartość `std::ignore` działa w charakterze imitacji wartości. To będzie wartość reszty z dzielenia usuwana w tym przykładzie, ponieważ jest w tym miejscu niepotrzebna.

> Podczas używania strukturalnego wiązania nie ma imitacji wartości, więc wszystkie wartości są przypisywane nazwanym zmiennym. Tego rodzaju operacja, a następnie zignorowanie niepotrzebnej wartości, jest mimo tego efektywna, ponieważ kompilator bardzo łatwo potrafi zoptymalizować nieużywane wiązania.

We wcześniejszych wydaniach standardu C++ funkcja `divide_remainder()` mogła być zaimplementowana w następujący sposób za pomocą parametrów danych wyjściowych:

```
bool divide_remainder(int dividend, int divisor,
                      int &fraction, int &remainder);
```

Uzyskanie dostępu do wartości odbywało się wówczas z użyciem kodu, takiego jak ten:

```
int fraction, remainder;
const bool success {divide_remainder(16, 3, fraction, remainder)};
if (success) {
    std::cout << "16 / 3 to " << fraction << " plus reszta z dzielenia "
              << remainder << '\n';
}
```

Wielu programistów nadal woli to podejście zamiast zwrotu skomplikowanych struktur danych w stylu na przykład pary lub krotki. Uznają oni, że przedstawiony kod będzie działał *szybciej* ze względu na uniknięcie pośredniego kopiowania wartości. Jednak w nowoczesnych kompilatorach kopiowanie *nie* zachodzi, co oznacza zoptymalizowanie operacji.

> Pomijając brakujące możliwości języka C, dostarczanie skomplikowanych struktur danych za pomocą wartości zwrotnej było przez długi czas uznawane za powolną operację ze względu na konieczność inicjalizacji obiektu, a następnie skopiowania wartości do zmiennej, która po stronie wywołującego powinna zawierać wartość zwrotną. Nowoczesne kompilatory oferują optymalizację wartości zwrotnej (ang. *return value optimization* — RVO), co pozwala na pominięcie pośrednich operacji kopiowania.

Ograniczanie zasięgu zmiennej do konstrukcji if i switch

W dobrym tonie jest maksymalne ograniczanie zasięgu zmiennej. Jednak czasami trzeba najpierw otrzymać wartość, która będzie mogła być dalej przetwarzana tylko po spełnieniu określonego warunku.

Do tego celu standard C++17 oferuje konstrukcje `if` i `switch` wraz z odpowiednimi procedurami inicjalizującymi.

Jak to zrobić?

W tej recepturze składnię inicjalizatora wykorzystam w obu obsługiwanych kontekstach, aby pokazać, jak dzięki niej kod może stać się znacznie bardziej przejrzysty.

- Konstrukcja `if`. Przyjmuję założenie, że za pomocą metody `find()` egzemplarza `std::map` trzeba znaleźć znak w mapowaniu znaków.

```
if (auto itr (character_map.find(c)); itr != character_map.end()) {
    // Iterator *itr jest poprawny i zostanie wykorzystany do pewnych operacji
} else {
    // Iterator itr jest iteratorem końcowym; nie należy przeprowadzać dereferencji
}
// Iterator itr jest tutaj niedostępny
```

- Konstrukcja `switch`. Oto jak przedstawia się operacja pobrania znaku z danych wejściowych i jednocześnie sprawdzenie wartości w konstrukcji `switch`, na przykład w celu zachowania kontroli nad grą:

```
switch (char c (getchar()); c) {
    case 'a': move_left();  break;
    case 's': move_back();  break;
    case 'w': move_fwd();   break;
```

```
        case 'd': move_right(); break;
        case 'q': quit_game();  break;

        case '0'...'9': select_tool('0' - c); break;

        default:
            std::cout << "Nieprawidłowe dane wejściowe: " << c << '\n';
    }
```

Jak to działa?

Konstrukcje `if` i `switch` wraz z inicjalizatorami to w zasadzie lukier syntaktyczny. Dwa przedstawione poniżej fragmenty kodu są swoimi odpowiednikami.

Kod stosowany *przed* wydaniem standardu C++17:

```
{
    auto var (init_value);
    if (condition) {
        // Zmienna var jest dostępna w gałęzi A
    } else {
        // Zmienna var jest dostępna w gałęzi B
    }
    // Zmienna var jest nadal dostępna
}
```

Kod stosowany *od chwili* wydania standardu C++17:

```
if (auto var (init_value); condition) {
    // Zmienna var jest dostępna w gałęzi A
} else {
    // Zmienna var jest dostępna w gałęzi B
}
// Zmienna var nie jest już dostępna
```

To samo ma zastosowanie w przypadku konstrukcji `switch`.

Kod stosowany *przed* wydaniem standardu C++17:

```
{
    auto var (init_value);
    switch (var) {
    case 1: ...
    case 2: ...
    ...
    }
    // Zmienna var jest nadal dostępna
}
```

Kod stosowany *od chwili* wydania standardu C++17:

```
switch (auto var (init_value); var) {
case 1: ...
case 2: ...
  ...
}
// Zmienna var nie jest już dostępna
```

Ta funkcja jest niezwykle użyteczna i pozwala maksymalnie zmniejszyć zasięg zmiennej. Przed wydaniem standardu C++17 osiągnięcie pokazanego efektu było możliwe jedynie po ujęciu kodu w dodatkowy nawias klamrowy, co możesz zobaczyć w przykładach zatytułowanych „Kod stosowany przed wydaniem standardu C++17". Krótszy cykl życiowy zmniejsza liczbę zmiennych w zasięgu, co z kolei przekłada się na większą przejrzystość kodu i jego łatwiejszą refaktoryzację.

Co dalej?

Kolejnym użytecznym przypadkiem jest ograniczenie zasięgu selekcji o znaczeniu krytycznym. Spójrz na ten fragment kodu:

```
if (std::lock_guard<std::mutex> lg {my_mutex}; some_condition) {
    // Dowolna operacja
}
```

W pierwszej chwili wydaje się, że zostanie utworzony egzemplarz `std::lock_guard`. To jest klasa akceptująca egzemplarz `mutex` jako argument konstruktora. Ten kod powoduje nałożenie na muteks *blokady* w konstruktorze, a po opuszczeniu przez niego zasięgu destruktor powoduje *zwolnienie* tej blokady. Dzięki temu praktycznie nie można *zapomnieć* o odblokowaniu muteksu. Przed wydaniem standardu C++17 konieczne było użycie dodatkowej pary nawiasów w celu ustalenia zasięgu i miejsca zwolnienia blokady.

Innym interesującym przypadkiem jest zasięg słabych wskaźników. Spójrz na następujący fragment kodu:

```
if (auto shared_pointer (weak_pointer.lock()); shared_pointer != nullptr) {
    // Współdzielony obiekt nadal istnieje
} else {
    // Zmienna shared_pointer jest dostępna, choć to będzie wskaźnik typu null
}
// Zmienna shared_pointer nie jest już dostępna
```

To jest kolejny przykład, w którym zmienna (tutaj `shared_pointer`) niepotrzebnie wycieka do bieżącego zasięgu. Ta zmienna okazuje się zupełnie bezużyteczna poza blokiem konstrukcji warunkowej `if`.

Konstrukcja `if` wraz z inicjalizatorem jest szczególnie użyteczna podczas pracy z *przestarzałym* API wykorzystującym parametry danych wyjściowych:

```
if (DWORD exit_code; GetExitCodeProcess(process_handle, &exit_code)) {
    std::cout << "Kod wyjścia procesu to: " << exit_code << '\n';
}
// Poza konstrukcją warunkową if zmienna exit_code jest nieprzydatna
```

`GetExitCodeProcess()` to funkcja API jądra systemu Windows. Jej wartością zwrotną jest kod wyjścia danego procesu, ale tylko wtedy, gdy uchwyt do procesu jest prawidłowy. Po opuszczeniu bloku konstrukcji warunkowej zmienna `exit_code` jest bezużyteczna i dlatego nie potrzebujemy jej już w żadnym zasięgu.

Możliwość inicjalizacji zmiennych w blokach konstrukcji `if` okazuje się bardzo użyteczna w wielu sytuacjach, zwłaszcza podczas pracy z już przestarzałymi API, które używają parametrów danych wyjściowych.

> Staraj się maksymalnie ograniczać zakres w poleceniach inicjalizacyjnych konstrukcji `if` i `switch`. Dzięki temu tworzony kod będzie zwięźlejszy, łatwiejszy w odczycie, a podczas refaktoryzacji będzie można łatwiej nim operować.

Zalety stosowania nowych reguł inicjalizacji z użyciem składni opartej na nawiasach

Wraz z wydaniem standardu C++11 pojawiła się nowa składnia inicjalizatora, wykorzystująca nawias klamrowy {}. Jej przeznaczeniem było umożliwienie *agregowania* inicjalizacji, a także używania zwykłych wywołań konstruktora. Niestety zbyt łatwo można było wyrazić nie to, co trzeba, w przypadku połączenia nowej składni z typem zmiennej `auto`. W standardzie C++17 wprowadzono rozbudowany zestaw reguł inicjalizatora. W tej recepturze pokażę, jak prawidłowo zainicjalizować zmienne, wykorzystując do tego składnię wprowadzoną w standardzie C++17.

Jak to zrobić?

Zmienna jest inicjalizowana w jednym kroku. Stosując składnię inicjalizatora, mamy dwie odmienne sytuacje:

- Użycie opartej na nawiasach składni inicjalizacji *bez* określenia typu `auto`:

```
// Trzy identyczne sposoby zainicjalizowania zmiennej typu int
int x1 = 1;
int x2  {1};
int x3  (1);
std::vector<int> v1   {1, 2, 3}; // Wektor wraz z trzema liczbami całkowitymi: 1, 2, 3
std::vector<int> v2 = {1, 2, 3}; // To samo co powyżej
std::vector<int> v3   (10, 20);  // Wektor wraz z dziesięcioma liczbami całkowitymi,
                                 // z których każda ma wartość 20
```

- Użycie opartej na nawiasach składni inicjalizacji z określeniem typu `auto`:

```
auto v   {1};          // Zmienna v jest typu int
auto w   {1, 2};       // Błąd: w bezpośredniej inicjalizacji auto dozwolony
                       // jest tylko jeden element! (to jest nowość)
auto x = {1};          // Zmienna x ma postać std::initializer_list<int>
auto y = {1, 2};       // Zmienna y ma postać std::initializer_list<int>
auto z = {1, 2, 3.0}; // Błąd: nie można ustalić typu elementu
```

Jak to działa?

Bez ustalania typu `auto` działanie operatora `{}` nie powinno być zaskoczeniem, przynajmniej podczas inicjalizowania wartości zwykłych typów. Z kolei w trakcie inicjalizowania kontenerów, takich jak `std::vector`, `std::list` itd., ten inicjalizator spowoduje dopasowanie konstruktora `std::initializer_list` klasy kontenera. To się odbywa w zachłanny sposób, co oznacza brak możliwości dopasowania konstruktorów nieagregujących (konstruktor nieagregujący to zwykły konstruktor będący przeciwieństwem akceptującego listę inicjalizatora).

Na przykład `std::vector` oferuje konstruktor nieagregujący, który wypełnia dowolną liczbę elementów tą samą wartością: `std::vector<int> v (N, wartość)`. W przypadku polecenia `std::vector<int> v {N, wartość}` zostanie użyty konstruktor `initializer_list` inicjalizujący wektor wraz z dwoma elementami: `N` i `wartość`. To jest zachowanie, o którym należy wiedzieć.

Warto również pamiętać o pewnym drobiazgu dotyczącym operatora `{}` w porównaniu do wywołania konstruktora za pomocą zwykłego nawiasu `()`, a mianowicie braku niejawnej konwersji typu. W przypadku wywołań `int x (1.2);` i `int x = 1.2;` otrzymamy zmienną `x` zainicjalizowaną wraz z wartością `1`. Wartość zmiennoprzecinkowa została zaokrąglona w dół i skonwertowana na postać liczby całkowitej. Natomiast wywołanie `int x {1.2};` uniemożliwia kompilację z powodu próby *ścisłego* dopasowania typu konstruktora.

> Można się spierać, który z przedstawionych powyżej stylów inicjalizacji jest najlepszy.
>
> Fani składni inicjalizacji opartej na nawiasie klamrowym uważają, że takie rozwiązanie konkretnie wskazuje typ zmiennej inicjalizowanej za pomocą wywołania konstruktora, a tego rodzaju wiersz kodu nie przeprowadza ponownej inicjalizacji czegokolwiek. Ponadto użycie nawiasu klamrowego spowoduje wybór jedynie dopasowanego konstruktora, podczas gdy składnia inicjalizacji opartej na nawiasie zwykłym próbuje dopasować najbliższy konstruktor — w tym celu może nawet przeprowadzić konwersję typu.

Dodatkowa reguła wprowadzona w standardzie C++17 wpływa na inicjalizację wraz z określaniem typu `auto`. W C++11 nastąpi poprawne ustalenie typu zmiennej `auto x {123};` jako `std::initializer_list<int>` z tylko jednym elementem, ale rzadko o to chodziło. Natomiast w standardzie C++17 to samo polecenie powoduje, że zmienna `x` jest typu `int`.

Pamiętaj o następujących regułach:

- Wywołanie `auto zmienna {jeden_element};` określa `zmienną` jako takiego samego typu jak `jeden_element`.
- Wywołanie `auto zmienna {element1, element2, ...};` jest nieprawidłowe i uniemożliwia kompilację kodu.
- Wywołanie `auto zmienna = {element1, element2, ...};` określa typ `zmiennej` jako `std::initializer_list<T>`, gdzie `T` to taki sam typ, jaki mają wszystkie elementy listy.

Standard C++17 znacznie utrudnia przypadkowe zdefiniowanie inicjalizatora listy.

> Wypróbowanie przedstawionych fragmentów kodu z różnymi kompilatorami działającymi w trybach zgodności z C++11 i C++14 pokaże, że niektóre z nich faktycznie określają zmienną `x` w wywołaniu `auto x {123};` jako typu `int`, podczas gdy inne — jako typu `std::initializer_list<int>`. Tworzenie kodu w taki sposób może prowadzić do problemów z jego przenoszeniem.

Umożliwienie konstruktorowi automatycznego określenia typu klasy szablonu

Wiele klas w C++ jest zwykle specjalizowanych dla określonych typów, które mogą być łatwo określane na podstawie typu zmiennych umieszczonych w wywołaniach konstruktorów. Przed wydaniem C++17 to nie była ustandaryzowana funkcja. C++17 pozwala kompilatorowi na *automatyczne* określanie typu klasy szablonu na podstawie wywołania konstruktora.

Jak to zrobić?

Bardzo dobrym przykładem jest tworzenie egzemplarzy typu `std::pair` i `std::tuple`. Mogą być one specjalizowane i tworzone w jednym kroku:

```
std::pair  my_pair  (123, "abc");        // std::pair<int, const char*>
std::tuple my_tuple (123, 12.3, "abc"); // std::tuple<int, double, const char*>
```

Jak to działa?

Oto przykład definicji klasy z automatycznym określaniem typu szablonu:

```
template <typename T1, typename T2, typename T3>
class my_wrapper {
    T1 t1;
```

```
    T2 t2;
    T3 t3;

public:
    explicit my_wrapper(T1 t1_, T2 t2_, T3 t3_)
        : t1{t1_}, t2{t2_}, t3{t3_}
    {}

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

Przedstawiony powyżej fragment kodu to tylko jeszcze jedna klasa szablonu. W celu utworzenia jej egzemplarza wcześniej trzeba było używać następującego polecenia:

```
my_wrapper<int, double, const char *> wrapper {123, 1.23, "abc"};
```

Natomiast teraz można pominąć fragment dotyczący specjalizacji szablonu:

```
my_wrapper wrapper {123, 1.23, "abc"};
```

Przed wydaniem standardu C++17 takie podejście było możliwe jedynie poprzez *utworzenie funkcji pomocniczej*:

```
my_wrapper<T1, T2, T3> make_wrapper(T1 t1, T2 t2, T3 t3)
{
    return {t1, t2, t3};
}
```

Wykorzystując taką funkcję pomocniczą, można był osiągnąć efekt podobny do poprzedniego, czyli pominąć fragment dotyczący specjalizacji szablonu, jak tutaj:

```
auto wrapper (make_wrapper(123, 1.23, "abc"));
```

> Biblioteka STL jest dostarczana z wieloma funkcjami pomocniczymi, takimi jak `std::make_shared()`, `std::make_unique()`, `std::make_tuple()` itd. W specyfikacji C++17 większość z nich można uznać za zbędne. Oczywiście pozostały one w celu zapewnienia wstecznej zgodności.

Co dalej?

Przed chwilą dość dokładnie przedstawiłem *niejawne określanie typu szablonu*. Jednak w pewnych sytuacjach nie można opierać się na niejawnym określaniu typu. Spójrz na ten oto fragment kodu:

```
template <typename T>
struct sum {
    T value;

    template <typename ... Ts>
    sum(Ts&& ... values) : value{(values + ...)} {}
};
```

Struktura sum akceptuje dowolną liczbę parametrów i dodaje je do siebie za pomocą wyrażenia fold (w dalszej części rozdziału przedstawię nieco więcej informacji na temat takich wyrażeń). Obliczona suma będzie zapisana w zmiennej składowej o nazwie value. Mógłbyś w tym miejscu zapytać, jakiego typu jest T. Jeżeli nie chcesz wyraźnie zdefiniować typu T, będzie on zależał od typu wartości przekazanych konstruktorowi. W przypadku ciągów tekstowych typem T będzie std::string. Jeżeli konstruktor otrzyma liczby całkowite, typem będzie int. Natomiast w przypadku dostarczenia konstruktorowi liczb całkowitych, zmiennoprzecinkowych i podwójnej precyzji kompilator musi wybrać odpowiedni typ umożliwiający przechowywanie tych wartości bez utraty jakiejkolwiek informacji. Aby osiągnąć ten cel, można zastosować *podpowiedź niejawnego określania typu*:

```
template <typename ... Ts>
sum(Ts&& ... ts) -> sum<std::common_type_t<Ts...>>;
```

Podpowiedź niejawnego określania typu wskazuje kompilatorowi możliwość użycia egzemplarza std::common_type w celu ustalenia wspólnego typu obejmującego wszystkie otrzymane wartości. Spójrz na przykład użycia podpowiedzi niejawnego określania typu:

```
sum s          {1u, 2.0, 3, 4.0f};
sum string_sum {std::string{"abc"}, "def"};

std::cout << s.value          << '\n'
          << string_sum.value << '\n';
```

W pierwszym wierszu następuje utworzenie obiektu sum wraz z argumentami konstruktora typów unsigned, double, int i float. W wyniku użycia std::common_type_t kompilator wybiera double jako wspólny typ obejmujący wartości przekazane konstruktorowi, więc otrzymujemy egzemplarz sum<double>. W drugim wierszu przekazywany jest egzemplarz std::string i ciąg tekstowy w stylu C. Kierując się podpowiedzią niejawnego określania typu, kompilator przygotowuje egzemplarz typu sum<std::string>.

Po uruchomieniu powyższego fragmentu kodu nastąpi wygenerowanie wartości liczbowej 10 i ciągu tekstowego abcdef.

Użycie wyrażenia constexpr-if do uproszczenia decyzji podejmowanych podczas kompilacji

W kodzie szablonu bardzo często zachodzi potrzeba odmiennego wykonywania pewnych zadań w zależności od typu szablonu, dla którego jest on specjalizowany. Standard C++17 oferuje wyrażenie constexpr-if, które potrafi w takich sytuacjach *znacznie* uprościć kod.

Jak to zrobić?

W tej recepturze zaimplementuję niewielką funkcję pomocniczą szablonu klasy. Jest ona przeznaczona do pracy z różnymi specjalizacjami typu szablonu, ponieważ potrafi wybrać zupełnie inny fragment kodu w zależności od typu specjalizacji.

1. Pracę należy rozpocząć od utworzenia kodu niezależnego od specjalizacji. W omawianym przykładzie to jest prosta klasa umożliwiająca dodanie wartości typu `U` do wartości elementu składowego typu `T` za pomocą funkcji `add()`:

```
template <typename T>
class addable
{
    T val;
public:
    addable(T v) : val{v} {}
    template <typename U>
    T add(U x) const {
        return val + x;
    }
};
```

2. Przyjmuję założenie, że typem `T` jest `std::vector<cokolwiek>`, natomiast typem `U` jest `int`. Jaki powinien być wynik dodania liczby całkowitej do wektora? Powiedzmy, że chcę dodać tę liczbę całkowitą do każdego elementu wektora. Taką operację przeprowadzam za pomocą pętli:

```
template <typename U>
T add(U x)
{
    auto copy (val); // Pobranie kopii elementu składowego wektora
    for (auto &n : copy) {
        n += x;
    }
    return copy;
}
```

3. Następnym i zarazem ostatnim krokiem jest *połączenie* obu światów. Jeżeli `T` jest wektorem elementów `U`, wówczas należy zastosować wariant oparty na pętli, w przeciwnym razie wystarczy zaimplementować *zwykłą* operację dodawania:

```
template <typename U>
T add(U x) const {
    if constexpr (std::is_same_v<T, std::vector<U>>) {
        auto copy (val);
        for (auto &n : copy) {
            n += x;
        }
        return copy;
    } else {
        return val + x;
    }
}
```

4. Klasa jest teraz gotowa do użycia. Spójrz, jak elegancko potrafi współpracować z różnymi typami danych, takimi jak `int`, `float`, `std::vector<int>` i `std::vector<string>`:

```
addable<int>{1}.add(2);                  // Wynik wynosi 3
addable<float>{1.0}.add(2);              // Wynik wynosi 3.0
addable<std::string>{"aa"}.add("bb"); // Wynik wynosi "aabb"

std::vector<int> v {1, 2, 3};
addable<std::vector<int>>{v}.add(10);
    // Wynik wynosi std::vector<int>{11, 12, 13}

std::vector<std::string> sv {"a", "b", "c"};
addable<std::vector<std::string>>{sv}.add(std::string{"z"});
    // Wynik wynosi {"az", "bz", "cz"}
```

Jak to działa?

Nowe wyrażenie `constexpr-if` działa tak samo jak zwykła konstrukcja warunkowa `if-else`. Różnica między nimi polega na tym, że w przypadku `constexpr-if` warunek będzie sprawdzony *podczas kompilacji*. Generowany przez kompilator kod uruchomieniowy programu nie będzie zawierał żadnych poleceń gałęzi `constexpr-if`. Innymi słowy: można pokusić się o stwierdzenie, że wyrażenie to działa podobnie jak makra preprocesora `#if` i `#else`. Jednak w przypadku tych makr kod nie musi być nawet doskonale sformatowany syntaktycznie. Natomiast wszystkie gałęzie w konstrukcji `constexpr-if` muszą być doskonale *sformatowane syntaktycznie*, choć jednocześnie gałęzie *nie* muszą być *syntaktycznie poprawne*.

W celu odróżnienia kodu odpowiedzialnego za dodanie wartości zmiennej `x` do wektora wykorzystam typ `std::is_same`. Wyrażenie `std::is_same<A, B>::value` przyjmuje boolowską wartość `true`, gdy `A` i `B` są tego samego typu. Warunek wykorzystany w tej recepturze ma postać `std::is_same<T, std::vector<U>>::value` i przyjmuje wartość `true`, nawet jeśli użytkownik specjalizuje klasę jako `T = std::vector<X>` i próbuje wywołać funkcję `add()` wraz z parametrem `U = X`.

Oczywiście pojedynczy blok `constexpr-if-else` może zawierać wiele poleceń. (Zwróć uwagę, że wartości `a` i `b` zależą od parametrów szablonu, a nie tylko od stałych w trakcie kompilacji).

```
if constexpr (a) {
    // Dowolna operacja
} else if constexpr (b) {
    // Inna dowolna operacja
} else {
    // Jeszcze inna dowolna operacja
}
```

W standardzie C++17 mamy wiele związanych z metaprogramowaniem sytuacji, w których utworzenie kodu i jego późniejszy odczyt będą łatwiejsze, jeśli użyjesz wyrażenia `constexpr-if`.

Co dalej?

Aby się przekonać, jak dużym usprawnieniem w języku C++ jest konstrukcja `constexpr-if`, spójrz na tę samą implementację przygotowaną jeszcze *przed* wprowadzeniem standardu C++17:

```
template <typename T>
class addable
{
    T val;

public:
    addable(T v) : val{v} {}
template <typename U>
    std::enable_if_t<!std::is_same<T, std::vector<U>>::value, T>
    add(U x) const { return val + x; }

    template <typename U>
    std::enable_if_t<std::is_same<T, std::vector<U>>::value,
                     std::vector<U>>
    add(U x) const {
        auto copy (val);
        for (auto &n : copy) {
            n += x;
        }
        return copy;
    }
};
```

Wprawdzie bez użycia wyrażenia `constexpr-if` powyższa klasa będzie obsługiwała wszystkie interesujące nas typy, ale jej kod jest niepotrzebnie skomplikowany. Jak to rozwiązanie działa w praktyce?

Sama implementacja *dwóch różnych* funkcji `add()` wygląda na prostą. Natomiast niezwykle skomplikowanie przedstawiają się deklaracje ich wartości zwrotnych wraz ze sztuczką w postaci wyrażenia takiego jak `std::enable_if_t<warunek, typ>`, które będzie podanego `typu`, jeśli `warunek` przyjmie wartość `true`. W przeciwnym razie `std::enable_if_t` nie określi niczego. Normalnie taka sytuacja jest uznawana za błąd, ale już wkrótce dowiesz się, dlaczego tutaj tak się nie dzieje.

W przypadku drugiej funkcji `add()` ten sam warunek jest używany w odwrotny sposób. Dlatego w danej sytuacji tylko jedna z dwóch istniejących implementacji może przyjąć wartość `true`.

Kiedy kompilator napotyka różne funkcje szablonu o tej samej nazwie i musi wybrać tylko jedną z nich, do gry wchodzi ważna reguła określana mianem **SFINAE** (ang. *substitution failure is not an error*), co oznacza: **niepowodzenie podczas podstawiania nie jest błędem**. W takim przypadku kompilator nie wygeneruje błędu, jeśli wartość zwrotna dowolnej z wymienionych funkcji nie będzie mogła być ustalona na podstawie nieprawidłowego wyrażenia szablonu (którym tutaj jest `std::enable_if`, ponieważ jego warunek przyjmuje wartość `false`). Kompilator po prostu szuka dalej i wypróbuje *inne* implementacje funkcji. Na tym polega sztuczka, dzięki której działa przedstawione rozwiązanie.

Mnóstwo z tym kłopotu. To doskonale pokazuje, że przedstawione zadanie można znacznie łatwiej wykonać dzięki użyciu kodu zgodnego ze standardem C++17.

Włączenie bibliotek w postaci samych nagłówków z użyciem osadzonych zmiennych

W języku C++ zawsze istniała możliwość deklarowania poszczególnych funkcji jako *osadzonych*. Standard C++17 pozwala deklarować *zmienne* jako osadzone. Dzięki temu można znacznie łatwiej zaimplementować biblioteki w postaci samych nagłówków, co wcześniej wymagało stosowania sztuczek.

Jak to zrobić?

W tej recepturze utworzę prostą klasę, która może działać w charakterze elementu składowego typowej biblioteki w postaci samych nagłówków. Celem jest dostarczenie statycznego elementu składowego i zainicjalizowanie go globalnie za pomocą słowa kluczowego `inline`. Takie podejście było niemożliwe przed wydaniem standardu C++17.

1. Klasa `process_monitor` powinna zawierać statyczne elementy składowe i jednocześnie być dostępna globalnie. Takie rozwiązanie wymagałoby podwójnego definiowania symboli.

```
// foo_lib.hpp
class process_monitor {
public:
    static const std::string standard_string
        {"pewien statyczny dostępny globalnie ciąg tekstowy"};
    };
process_monitor global_process_monitor;
```

2. Jeżeli teraz plik zawierający powyższą klasę będzie dołączony w wielu plikach *.cpp*, aby przeprowadzić ich kompilację i linkowanie, wówczas etap linkowania zakończy się niepowodzeniem. Rozwiązaniem problemu jest dodanie słowa kluczowego `inline`:

```
// foo_lib.hpp
class process_monitor {
public:
    static const inline std::string standard_string
        {"pewien statyczny dostępny globalnie ciąg tekstowy"};
};

inline process_monitor global_process_monitor;
```

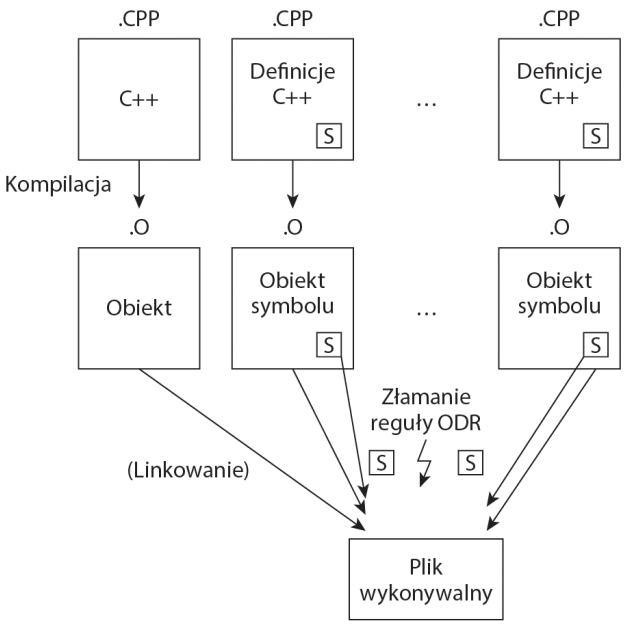
Voilà, o to chodziło!

Jak to działa?

Program w C++ często składa się z wielu plików kodu źródłowego (mają one rozszerzenie *.cpp* lub *.cc*). Najczęściej są one pojedynczo komplikowane na postać plików modułów lub obiektów (w takim przypadku rozszerzeniem pliku jest *.o*). Ostatnim krokiem jest linkowanie wszystkich plików modułów i obiektów w celu wygenerowania pojedynczego pliku wykonywalnego bądź biblioteki współdzielonej lub statycznej.

Jeżeli na etapie linkowania definicja danego symbolu zostanie znaleziona *wielokrotnie*, wówczas mamy do czynienia z błędem. Przyjmuję założenie o istnieniu funkcji z sygnaturą taką jak `int foo();`. Kiedy dwa moduły definiują tę samą funkcję, która z nich jest prawidłowa? Linker nie może wybierać losowo implementacji. Cóż, mógłby, ale jest mało prawdopodobne, aby takiego podejścia oczekiwał jakikolwiek programista.

Tradycyjnym sposobem dostarczania globalnie dostępnych funkcji jest ich *deklarowanie* w plikach nagłówkowych, które mogą być dołączane przez dowolny moduł C++ potrzebujący tych funkcji. Definicja każdej funkcji dostępnej globalnie będzie umieszczona *raz* w oddzielnym pliku modułu. Następnie te pliki będą linkowane wraz z modułami, w których mają być używane dane funkcje. To podejście jest określane mianem **reguły ODR** (ang. *one definition rule*), czyli **reguły pojedynczej definicji**. W sposób graficzny tę regułę pokazałem na rysunku 1.1.



Rysunek 1.1. Reguła pojedynczej definicji

Jeżeli to byłoby jedyne rozwiązanie, wówczas zabrakłoby możliwości dostarczania bibliotek w postaci samych nagłówków. Biblioteki okazują się niezwykle użyteczne, ponieważ mogą być dołączane w dowolnym pliku programu C++ za pomocą polecenia `#include`, po którego wykonaniu są natychmiast dostępne. W celu użycia biblioteki w postaci innej niż samego nagłówka programista musi zaadaptować skrypt kompilacji, aby linker dołączył moduły biblioteki do plików modułów programu. To jest szczególnie niewygodne w przypadku bibliotek zawierających niewielką liczbę małych funkcji.

Z pomocą przychodzi tutaj słowo kluczowe `inline`, które można zastosować do zdefiniowania wyjątku umożliwiającego użycie w różnych modułach *wielu* definicji tego samego symbolu. Jeżeli linker znajdzie wiele symboli o takiej samej sygnaturze, ale zadeklarowanych jako osadzone, wówczas wybierze pierwszy, uznając, że pozostałe symbole mają taką samą definicję. Zagwarantowanie takich samych definicji dla wszystkich osadzonych symboli jest w zasadzie *obietnicą* składaną przez programistę.

Powracam teraz do omawianej tutaj receptury — linker znajdzie symbol `process_monitor::standard_string` w każdym module importującym *foo_lib.hpp*. Bez słowa kluczowego `inline` nie będzie wiedział, który z nich powinien zostać użyty, więc spowoduje wygenerowanie błędu. To samo dotyczy również symbolu `global_process_monitor`. Który z nich jest prawidłowy?

Po zadeklarowaniu obu symboli za pomocą słowa kluczowego `inline` nastąpi akceptacja pierwszego wystąpienia każdego symbolu i *odrzucenie* wszystkich kolejnych wystąpień.

Przed wprowadzeniem standardu C++17 jedynym przejrzystym sposobem implementacji przedstawionego powyżej rozwiązania było dostarczenie symbolu za pomocą dodatkowego pliku modułu C++. To wymagało od użytkowników biblioteki dołączania tego pliku na etapie linkowania.

Słowo kluczowe `inline` tradycyjnie miało również *inną* funkcję — wskazywało kompilatorowi możliwość usunięcia funkcji poprzez pobranie jej implementacji i bezpośrednie wstawienie w miejscu wywołania tej funkcji. W ten sposób kod zawierał o jedno wywołanie funkcji mniej, co często mogło przekładać się na większą wydajność działania. Jeżeli funkcja jest niewielka, podzespół wynikowy również będzie mniejszy (przy założeniu, że liczba poleceń odpowiedzialnych za wywołanie tej funkcji, umieszczenie i pobranie jej ze stosu itd. jest większa niż faktyczny kod funkcji). Natomiast w przypadku, gdy osadzona funkcja jest duża, wielkość pliku binarnego ulegnie zwiększeniu i ostatecznie może się zdarzyć, że kod nie będzie działał szybciej. Dlatego kompilator traktuje słowo kluczowe `inline` jedynie jako wskazówkę i może eliminować wywołania funkcji poprzez osadzanie ich w kodzie. Czasami może też osadzać w kodzie pewne funkcje, nawet jeśli *nie* zostały przez programistę określone jako osadzone.

Co dalej?

Jednym z możliwych rozwiązań problemu przed wprowadzeniem standardu C++17 było użycie funkcji statycznej zwracającej odwołanie do obiektu `static`. Spójrz na fragment kodu:

```
class foo {
public:
    static std::string& standard_string() {
        static std::string s {"Dowolny standardowy ciąg tekstowy."};
        return s;
    }
};
```

W ten sposób można całkowicie prawidłowo dołączyć plik nagłówkowy w wielu modułach i nadal zapewnić sobie wszędzie dostęp do dokładnie tego samego egzemplarza. Jednak wspomniany obiekt *nie* był tworzony *natychmiast* po uruchomieniu programu, a dopiero po pierwszym wywołaniu funkcji getter. W niektórych przypadkach to może stanowić problem. Wyobraź sobie sytuację, w której konstruktor statycznego, globalnie dostępnego obiektu ma wykonać pewne ważne zadanie (podobnie jak w przypadku przykładowej biblioteki w tej recepturze) *w trakcie uruchamiania programu*. Jednak ze względu na to, że wywołanie następuje prawie na końcu działania programu, wykonanie zadania się nie udało.

Innym rozwiązaniem jest zdefiniowanie niebędącej szablonem klasy `foo` jako klasy szablonu, aby w ten sposób móc skorzystać z tych samych zalet, które oferują szablony.

Obu strategii można uniknąć w kodzie źródłowym zgodnym ze standardem C++17.

Implementowanie za pomocą wyrażeń fold przydatnych funkcji pomocniczych

Od chwili wydania standardu C++11 mamy pakiety parametrów wariadycznych, co pozwala zaimplementować funkcje akceptujące dowolną liczbę parametrów. Czasami te parametry są łączone w jednym wyrażeniu w celu pobrania wyniku wykonania funkcji. To zadanie stało się znacznie łatwiejsze w C++17 dzięki pojawieniu się wyrażeń fold.

Jak to zrobić?

Przystępuję do zaimplementowania funkcji pobierającej dowolną liczbę parametrów i zwracającej ich sumę.

1. Pracę rozpoczynam od zdefiniowania sygnatury funkcji:

```
template <typename ... Ts>
      auto sum(Ts ... ts);
```

2. W tym momencie mam pakiet parametrów `ts`, funkcja powinna rozwinąć wszystkie parametry i obliczyć ich sumę za pomocą wyrażenia fold. Jeżeli w celu zastosowania wszystkich wartości pakietu parametrów zostanie użyty operator (w omawianym przykładzie to operator +) wraz z wielokropkiem, wówczas wyrażenie należy ująć w nawias:

```
template <typename ... Ts>
auto sum(Ts ... ts)
{
    return (ts + ...);
}
```

3. W tym momencie można użyć następującego wywołania:

```
int the_sum {sum(1, 2, 3, 4, 5)}; // Wartość: 15
```

4. Przedstawione rozwiązanie działa nie tylko z liczbami całkowitymi, ale również z każdym innym typem implementującym funkcję `operator+()`, na przykład `std::string`:

```
std::string a {"Witaj, "};
std::string b {"świecie!"};
std::cout << sum(a, b) << '\n'; // Dane wyjściowe: Witaj, świecie!
```

Jak to działa?

W przedstawionym rozwiązaniu po prostu zastosowałem dla parametrów funkcji rekurencyjny operator dwuargumentowy `+`. Takie działanie jest określane mianem **foldingu**. Standard C++17 oferuje obsługę **wyrażeń fold**, dzięki którym tę samą ideę można wyrazić za pomocą mniejszej ilości kodu.

Przedstawiony rodzaj wyrażenia jest nazywany **foldingiem jednoargumentowym**. Standard C++17 oferuje obsługę foldingu pakietów parametrów wraz z następującymi operatorami dwuargumentowymi: `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `=`, `<`, `>`, `<<`, `>>`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<=`, `>>=`, `==`, `!=`, `<=`, `>=`, `&&`, `||`, `,`, `.*`, `->*`.

Warto w tym miejscu wspomnieć, że w omawianym tutaj przykładowym fragmencie kodu nie ma żadnego znaczenia, który z dwóch zapisów zostanie użyty — `(ts + ...)` czy `(... + ts)`. Jednak istnieje różnica, która może mieć znaczenie w innych sytuacjach: jeśli wielokropek znajduje się *po prawej* stronie operatora, wówczas mamy do czynienia z prawym *foldingiem*, natomiast wielokropek po lewej stronie operatora oznacza istnienie *lewego* foldingu.

W omawianym przykładzie lewy folding zostanie rozwinięty do postaci `1 + (2 + (3 + (4 + 5)))`, natomiast prawy do postaci `(((1 + 2) + 3) + 4) + 5`. W zależności od użytego operatora to może mieć znaczenie. W przypadku dodawania liczb nie ma żadnego znaczenia.

Co dalej?

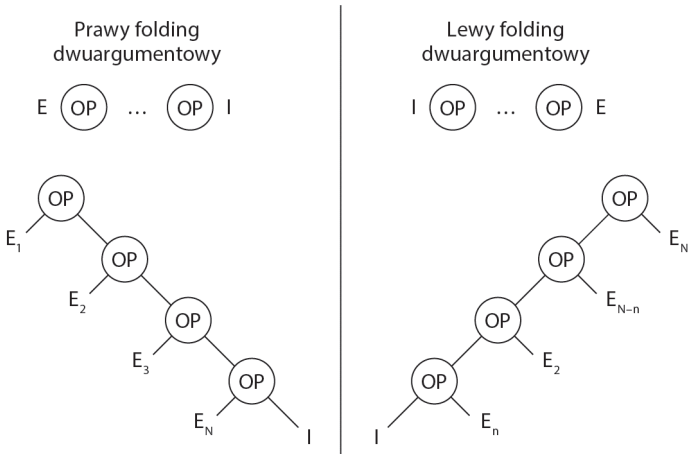
W przypadku wywołania funkcji `sum()` *bez* parametrów parametr wariadyczny nie zawiera żadnych wartości, które mogłyby zostać poddane foldingowi. W większości operatorów taka sytuacja spowoduje wygenerowanie błędu (choć nie we wszystkich, do tego jeszcze powrócę w niniejszym rozdziale). Trzeba wówczas zdecydować, czy błąd ma pozostać, czy ma zostać zwrócona pusta suma. Oczywiste jest, że suma niczego wynosi `0`.

Oto jak to można zrobić:

```
template <typename ... Ts>
auto sum(Ts ... ts)
{
    return (ts + ... + 0);
}
```

W ten sposób wynik wywołania funkcji `sum()` wynosi `0`, natomiast wywołanie `sum(1, 2, 3)` zostaje rozwinięte do postaci `(1 + (2 + (3 + 0)))`. Takie wyrażenie wraz z wartością początkową jest określane mianem **foldingu dwuargumentowego**.

W tym przypadku zapis również nie ma znaczenia — `(ts + ... + 0)` lub `(0 + ... + ts)` — choć wpływa na rodzaj foldingu dwuargumentowego: może być *prawy* lub *lewy*. Spójrz na rysunek 1.2.



Rysunek 1.2. Przykłady lewego i prawego foldingu dwuargumentowego

Podczas użycia foldingu dwuargumentowego w celu zaimplementowania obsługi sytuacji, w której nie zostanie podany parametr, element *identyczności* jest najczęściej ważny. W omawianym przykładzie dodanie `0` do dowolnej liczby niczego nie zmienia, więc `0` staje się elementem identyczności. Z powodu tej właściwości za pomocą operatora `+` lub `-` można używać `0` w dowolnym wyrażeniu fold. W przypadku wywołania funkcji bez parametrów dodanie `0` powoduje otrzymanie wyniku wynoszącego `0`. Z matematycznego punktu widzenia to jest poprawne. Natomiast z perspektywy implementacji na podstawie wymagań konieczne jest zdefiniowanie, co należy uznać z poprawne.

Ta sama reguła ma zastosowanie w przypadku mnożenia. Tutaj elementem identyczności jest `1`:

```
template <typename ... Ts>
auto product(Ts ... ts)
{
    return (ts * ... * 1);
}
```

Wynikiem wywołania funkcji `product(2, 3)` jest 6, natomiast wynik wywołania tej samej funkcji, ale bez parametrów wynosi 1.

Operatory logiczne **i** (`&&`) oraz **lub** (`||`) są dostarczane wraz z *wbudowanymi* elementami identyczności. Folding pustego pakietu parametrów dla operatora `&&` powoduje wygenerowanie wartości `true`, natomiast w przypadku operatora `||` będzie to wartość `false`.

Kolejny operator, który domyślnie stosuje pewne wyrażenie w przypadku użycia operatora bez parametrów, to przecinek. Jeżeli użyjesz go bez parametrów, domyślnie wywoła `void()`.

Przejdę teraz do kilku innych niewielkich funkcji pomocniczych, które można zaimplementować za pomocą omówionej funkcjonalności.

Dopasowanie zakresu względem poszczególnych elementów

Oto funkcja, która informuje, czy pewien zakres zawiera *co najmniej* jedną z wartości podanych jako parametr wariadyczny:

```
template <typename R, typename ... Ts>
auto matches(const R& range, Ts ... ts)
{
    return (std::count(std::begin(range), std::end(range), ts) + ...);
}
```

Ta funkcja pomocnicza wykorzystuje funkcję `std::count()` biblioteki STL i pobiera trzy parametry. Dwa pierwsze to iteratory *początkowy* i *końcowy* pewnego zakresu, trzeci zaś zawiera *wartość* porównywaną ze wszystkimi elementami we wskazanym zakresie. Metoda `std::count()` zwraca liczbę elementów w zakresie, które są identyczne z elementem podanym w trzecim parametrze wywołania `std::count()`.

W naszym wyrażeniu fold zawsze znajdują się te same iteratory *początkowy* i *końcowy* zakresu przekazywanego funkcji `std::count()`. Natomiast jako trzeciego parametru używamy po jednej wartości pochodzącej z pakietu parametru wariadycznego. Ostatecznie funkcja sumuje wyniki otrzymane dla poszczególnych wartości parametru wariadycznego i przekazuje je komponentowi wywołującemu.

Omawianej tutaj funkcji pomocniczej można użyć w następujący sposób:

```
std::vector<int> v {1, 2, 3, 4, 5};

matches(v,          2, 5);          // Wynik: 2
matches(v,          100, 200);      // Wynik: 0
matches("abcdefg", 'x', 'y', 'z'); // Wynik: 0
matches("abcdefg", 'a', 'd', 'f'); // Wynik: 3
```

Jak widzisz, funkcja pomocnicza `matches()` jest dość wszechstronna — można ją wywołać bezpośrednio dla wektorów, a nawet ciągów tekstowych. Działa również z metodami inicjalizacyjnymi list, czyli egzemplarzami `std::list`, `std::array`, `std::set` itd.

Sprawdzenie, czy operacja wstawienia wielu elementów do zbioru zakończyła się pomyślnie

Utworzę teraz funkcję pomocniczą wstawiającą dowolną liczbę parametrów wariadycznych do zbioru `std::set` i zwracającą informacje o tym, czy wstawienie wszystkich elementów zakończyło się *pomyślnie*:

```
template <typename T, typename ... Ts>
bool insert_all(T &set, Ts ... ts)
{
    return (set.insert(ts).second && ...);
}
```

W jaki sposób działa powyższy fragment kodu? Funkcja `insert()` zbioru `std::set` ma następującą sygnaturę:

```
std::pair<iterator, bool> insert(const typ_wartości& wartość);
```

Zgodnie z dokumentacją, w trakcie operacji wstawiania elementu, wartością zwrotną funkcji `insert()` jest para iterator i zmienna boolowska. Jeżeli wstawienie się powiodło, zmienna boolowska przyjmie wartość `true`, a iterator będzie wskazywał *nowy element* w zbiorze. W przeciwnym razie iterator wskazuje *istniejący* element, który może *kolidować* z elementem przeznaczonym do wstawienia.

Po operacji wstawienia omawiana tutaj funkcja pomocnicza uzyskuje dostęp do właściwości `.second`, która jest po prostu zmienną boolowską odzwierciedlającą sukces lub niepowodzenie. Jeżeli wszystkie operacje wstawienia doprowadziły do przypisania wartości `true` zmiennej boolowskiej w każdej parze wyniku działania funkcji, oznacza to, że wstawienie elementów zakończyło się powodzeniem. Wyrażenie fold za pomocą operatora `&&` łączy wyniki wszystkich operacji wstawienia i zwraca ostateczny wynik.

Omawianej tutaj funkcji pomocniczej można użyć w następujący sposób:

```
std::set<int> my_set {1, 2, 3};

insert_all(my_set, 4, 5, 6); // Wartością zwrotną jest true
insert_all(my_set, 7, 8, 2); // Wartością zwrotną jest false, ponieważ element 2 powoduje kolizję z istniejącym
```

Jeżeli spróbujesz wstawić na przykład trzy elementy, a drugi z nich nie może zostać wstawiony, wówczas wyrażenie `&& ...` skróci wykonywanie operacji i wstrzyma przetwarzanie pozostałych elementów:

```
std::set<int> my_set {1, 2, 3};

insert_all(my_set, 4, 2, 5); // Wartością zwrotną jest false
// Teraz zbiór zawiera elementy {1, 2, 3, 4}, ale bez 5!
```

Sprawdzenie, czy wszystkie parametry znajdują się w danym zakresie

Skoro można ustalić, czy *jedna* zmienna znajduje się w określonym zakresie, za pomocą wyrażenia fold to samo można sprawdzić dla wielu zmiennych:

```
template <typename T, typename ... Ts>
bool within(T min, T max, Ts ...ts)
{
    return ((min <= ts && ts <= max) && ...);
}
```

Wyrażenie (`min <= ts && ts <= max`) sprawdza, czy poszczególne wartości parametru wariadycznego znajdują się w zakresie definiowanym przez `min` i `max` (łącznie z `min` i `max`). Zdecydowałem się na użycie operatora `&&`, aby zmniejszyć liczbę wyników boolowskich do jednego. Jeżeli wszystkie wyniki cząstkowe przyjęły wartość `true`, wynik końcowy również będzie miał wartość `true`.

Oto omawiana funkcja pomocnicza w akcji:

```
within( 10,  20,  1, 15, 30);     // --> false
within( 10,  20,  11, 12, 13);    // --> true
within(5.0, 5.5,  5.1, 5.2, 5.3) // --> true
```

Ta funkcja pomocnicza jest niezwykle wszechstronna, ponieważ jedynym ograniczeniem nakładanym na typ jest możliwość *porównania* za pomocą operatora `<=`. To wymaganie jest spełnione między innymi przez egzemplarz `std::string`, jak pokazałem w następującym fragmencie kodu.

```
std::string aaa {"aaa"};
std::string bcd {"bcd"};
std::string def {"def"};
std::string zzz {"zzz"};

within(aaa, zzz,  bcd, def); // --> true
within(aaa, def,  bcd, zzz); // --> false
```

Umieszczenie wielu elementów w wektorze

Istnieje również możliwość utworzenia funkcji pomocniczej nie zmniejszającej liczby wyników, ale przetwarzającej wiele akcji tego samego rodzaju. Przykładem może być tutaj wstawianie elementów do kontenera `std::vector`, co nie powoduje zwrotu żadnego wyniku — wywołanie `std::vector::insert()` sygnalizuje błąd poprzez zgłoszenie wyjątku.

```
template <typename T, typename ... Ts>
void insert_all(std::vector<T> &vec, Ts ... ts)
{
    (vec.push_back(ts), ...);
}

int main()
```

```
{
    std::vector<int> v {1, 2, 3};
    insert_all(v, 4, 5, 6);
}
```

Zwróć uwagę na użycie operatora `,` w celu rozwinięcia parametru wariadycznego na poszczególne wywołania `vec.push_back(...)` bez zastosowania foldingu dla rzeczywistego wyniku. Prezentowana tutaj funkcja działa również w przypadku *pustego* parametru wariadycznego, ponieważ operator `,` ma niejawnie zdefiniowany element identyczności, `void()`, oznaczający *nie rób nic*.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ź

C++17 i STL. Elegancka klasyka i potężne możliwości.

C++ pozwala zarówno na tworzenie interfejsów wysokiego poziomu, jak i na pisanie kodu działającego na niskim poziomie. Sprawdza się, gdy trzeba zapewnić wysoką wydajność i niskie obciążenie. Język ten jest konsekwentnie rozwijany: jego kolejne specyfikacje, C++14 i C++17, przyniosły wiele znakomitych udoskonaleń. Aby w pełni wykorzystać ten potencjał, należy używać C++ łącznie z biblioteką STL. Jest to standardowa biblioteka języka, dzięki której C++ jest idealny do implementowania oprogramowania o wysokiej jakości i dużej wydajności. Zalety C++ sprawiają, że jest wykorzystywany niemal w każdej dziedzinie. Niestety, wielu programistów nie używa STL.

Dzięki tej książce poznasz użyteczność biblioteki standardowej (STL) w C++17 w praktyce, co pozwoli Ci na tworzenie efektywniejszego i w pełni przenośnego kodu źródłowego. Najpierw poznasz nowe funkcje języka, co pozwoli Ci na zrozumienie reguł rządzących C++, oraz funkcje biblioteki standardowej i sposób jej działania. Podczas pracy nad praktycznymi i łatwymi do wykorzystania recepturami poznasz podstawowe koncepcje STL, takie jak kontener, algorytm, klasa narzędziowa, wyrażenie lambda, iterator i wiele innych. Dowiesz się, jak działają najnowsze funkcje wprowadzone w standardzie C++17. Dzięki temu zaoszczędzisz czas i wysiłek podczas programowania, a Twój kod stanie się prostszy i zdecydowanie bardziej elegancki.

W książce między innymi:

- nowości w standardzie C++17
- kontenery STL i koncepcja iteratorów
- wyrażenia lambda i zaawansowane algorytmy biblioteki STL
- ciągi tekstowe, strumienie wejścia-wyjścia i wyrażenia regularne
- programowanie równoległe i współbieżność

Jacek Galowicz do niedawna implementował sterowniki jądra w C i C++. Tworzył też grafiki 3D i bazy danych, a także zajmował się komunikacją sieciową. Aktualnie pracuje w firmach Intel oraz FireEye, w których programuje systemy operacyjne o dużej wydajności i wysokim poziomie bezpieczeństwa, wykorzystując wirtualizację Intel x86. Jego ogromną pasją są nowoczesne implementacje C++ na niskim poziomie i łączenie w kodzie wysokiej wydajności z eleganckim stylem.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
WWW.SZKOLENIA.HELION.PL
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-4501-0
9 788328 345010
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 79,00 zł
Packt