B Dodatek: Nowe cechy języka wprowadzone do standardu C++14

To jest dodatek do książki: Jerzy Grębosz *„Opus magnum C++11”* (wyd. Helion).

Jest on sporządzony w formie pliku PDF, zatem może być sukcesywnie uaktualniany.

Ta wersja jest z dnia 27 marca 2018.

Ewentualną uaktualnioną wersję tego tekstu możesz sprowadzić sobie ze strony WWW poświęconej książce Opus magnum C++:

https://www.ifj.edu.pl/private/grebosz/opus.html

Na tej stronie znajdziesz też bieżącą erratę i kody źródłowe programów z książki.

C++14 to nowsza wersja standardu języka C++. Praktycznie rzecz biorąc jest tylko drobnym rozszerzeniem standardu C++11. Znawcy przedmiotu żartują, że do C++14 weszło kilka tych zagadnień, które spóźniły się na odjazd standardu C++11. W tym rozdziale poznamy nowości, które pojawiły się w C++14. Niektóre z nich zapowiadaliśmy już w poprzednich rozdziałach *„Opusu”*.

B.1 Zapis dwójkowy stałych dosłownych

Zacznijmy od przypomnienia. W tekście programu często występuje konieczność napisania jakichś stałych wartości. To tak zwane stałe dosłowne. Liczbową stałą dosłowną najczęściej zapisujemy posługując się zwykłym dziesiątkowym systemem liczenia. Czasem jednak wygodniej napisać ją szesnastkowo (czyli heksadecymalnie). Możliwy jest też zapis ósemkowy (czyli oktalny).

Przez domniemanie kompilator uznaje, że liczbowa stała dosłowna jest podana w zapisie dziesiątkowym. Jeśli jednak widzi, że stałą dosłowną rozpoczyna przedrostek 0x (zero i x), to rozumie, że zastosowaliśmy zapis szesnastkowy. Jeśli przedrostkiem jest samo zero, to rozumie, że ta liczba zapisana została ósemkowo. Oto przykład wyrażenia, w którym występują trzy stałe dosłowne, a każda zapisana w inny sposób.

```
int zmienna =  250 +  0xff + 0177 ;          // czyli to samo co dziesiątkowo 250 + 255 + 127
```

Standard C++14 dodaje do tego jeszcze jedną możliwość.

Stałą dosłowną możemy napisać także w postaci liczby dwójkowej (binarnej). O tym, że zastosowaliśmy taki zapis, informujemy kompilator poprzedzając daną liczbę przedrostkiem 0b lub 0B. Łatwo zapamiętać: litera B to oczywiście skrót od „binarnie".

```
int a = 0b101;                // 0b101 to dziesiątkowo 5
```

Tutaj dwójkowa stała dosłowna użyta została w wyrażeniu inicjalizującym, ale oczywiście może wystąpić także w innych instrukcjach programu. Na przykład w warunku instrukcji if.

```
if(x &  0b10000) ...          // Jeśli w zmiennej x jest ustawiony bit 4, to...
```

Przypominam, że bity numerujemy od prawej do lewej zaczynając od zera.

Zapis dwójkowy ma jednak pewną niedogodność: im liczba jest dłuższa, tym bardziej nieczytelna. Przy zapisie 0b1101100001010110101 można od tych zer i jedynek dostać oczopląsu. Jeśli programista popełni błąd w tak zapisanej liczbie, to tego błędu łatwo nie zauważy. Pewnie dlatego w dawniejszych wersjach C++ nie było zapisu stałych dosłownych w postaci dwójkowej. W podobnych sytuacjach zamiast zapisu binarnego stosowaliśmy zapis szesnastkowy.

Co takiego się wydarzyło, że teraz (w C++14) nieczytelność zapisu binarnego już nam nie przeszkadza? O tym opowie następny paragraf.

B.2 Separatory cyfr w stałych dosłownych

Spójrz szybko na liczbę 500000000 i wypowiedz na głos jej wartość. No co, zacząłeś od żmudnego liczenia tych zer i może nawet się pogubiłeś. A teraz to samo zadanie w nieco zmienionej wersji. „Wypowiedz na głos wartość liczby: 500 000 000". Teraz jasne, to pięćset milionów. Dlaczego teraz jest łatwiej? Pomogło podzielenie cyfr tej liczby na grupy 3-cyfrowe. Taki zapis człowiek przeczyta dużo łatwiej.

Ucieszy Cię więc pewnie wiadomość, że:

C++14 w zapisie stałych dosłownych pozwala programiście zastosować podobne grupowanie cyfr. Robimy to nie za pomocą spacji, lecz za pomocą znaku apostrof.

```
int zmienna = 500'000'000;
```

Apostrofy oddzielają od siebie (separują) wybrane grupy cyfr. Separatory te wolno nam postawić *wewnątrz* zapisu liczby, w wybranym miejscu.

- Ważne jest słowo *wewnątrz*; apostrof nie może stać ani na samym początku, ani na samym końcu tej liczby.
- Nie można też stawiać dwóch separatorów bezpośrednio obok siebie.

Poza tym mamy pełną swobodę. Możemy na przykład oddzielać cyfry grzecznie w grupach po trzy cyfry (jak powyżej), ale równie dobrze możemy podzielić je na jakieś „nierówne" grupy.

```
int zmienna = 5'0'00'0000'0;
```

Co na te separatory powie kompilator? Dla kompilatora separator cyfr jest jakby „niewidzialny", po prostu go ignoruje. Jednak dla nas, programistów, separator może być ogromnym ułatwieniem przy czytaniu. Są jeszcze dwie dobre wiadomości.

Separatorem cyfr możemy posłużyć się nie tylko w zapisie liczb całkowitych. Możemy go użyć także w przypadku liczb zmiennoprzecinkowych.

```
double x = 987'321.333'444;
double y = 1.000'00'9;
```

Separatora cyfr możemy użyć w liczbach zapisanych w postaci dziesiątkowej, szesnastkowej ósemkowej, a nawet dwójkowej.

```
int x = 0xffff'fffc;
int y = 0b1010'1111'0111;
```

W tej ostatniej instrukcji widzimy stałą dosłowną w postaci dwójkowej, a w niej zastosowany jest separator cyfr. Patrząc na ten zapis, łatwo zrozumieć dlaczego standard C++14 pozwala nam już na zapis stałych dosłownych w postaci dwójkowej: kompilator wie, że dzięki separatorowi możemy taki zapis uczynić przejrzystszym.

Dzięki obecności separatorów w powyższej stałej dosłownej, łatwiej na przykład odpowiedzieć na pytanie czy w tej zmiennej y (powyżej) bit 8 jest ustawiony czy nie. No co, jest czy nie?

Skoro separatorem pogrupowaliśmy w powyższym zapisie bity po cztery, to łatwo znaleźć bit nr 8. Jak widać, ma on wartość 0. (Numerujemy je od zera!).

Dowiedzieliśmy się właśnie, że separator cyfr stosujemy tylko w zapisie stałej dosłownej w tekście źródłowym programu. A co na to kompilator? Gdy kompilator pracuje nad naszym programem, to te apostrofy separujące po prostu pomija i nasz wygodny zapis zamienia sobie w zwykły zapis (taki bez separatorów).

Zapamiętaj, że strumienie we/wy nie honorują separatora...

...więc jeśli zechcesz wypisać na ekranie tak zapisaną stałą dosłowną, to nie zdziw się, że na ekranie tych separatorów-apostrofów już nie będzie.

```
cout << 20'324'222;          // na ekranie pojawi się: 20324222
```

Powtarzam: separatorów cyfr możemy używać ***tylko w tekście źródłowym programu***. Nie możemy więc ich użyć podczas pracy gotowego programu, np. przy wprowadzaniu do programu jakiejś danej instrukcją:

```
int n;
cin >> n;
```

Gdybyś w takiej sytuacji napisał na klawaturze 50'987, to strumień cin uzna, że apostrof kończy zapis liczby. W zmiennej n znajdzie się więc wartość 50. Natomiast apostrof i te następujące po nim cyfry ('987) będą czekały w buforze na ewentualną następną operację wczytywania.

Jeśli pomyślałeś, że to trochę szkoda, że ani cout, ani cin nie honorują separatorów, to mam jeszcze gorszą wiadomość: one nie honorują także zapisu dwójkowego. Głowa do góry! W następnych paragrafach zobaczysz, jak można te ograniczenia obejść.

B.2.1 Wypisywanie liczb w postaci binarnej

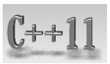
Poznaliśmy niedawno sposób zapisywania stałych dosłownych w postaci dwójkowej (binarnej). Jeśli taką stałą dosłowną przypiszemy do jakiegoś obiektu, a ten obiekt zechcemy wypisać na ekranie, to przez domniemanie wartość tego obiektu zostanie wypisana w postaci liczby dziesiętnej. Jeśli nam to nie odpowiada, to używając

odpowiedniego manipulatora możemy polecić strumieniowi, aby wypisał to samo w postaci liczby ósemkowej lub szesnastkowej.

```
int a = 0B1011101;
cout << showbase << a << endl;          // wypisze: 93
cout << oct << a << endl;               // wypisze: 0135
cout << hex << a << endl;               // wypisze: 0x5d
```

Jednak strumienie we/wy nie oferują (jak na razie) manipulatora, który by sprawił, że liczba zostanie wypisana w postaci dwójkowej.

Co jednak zrobić, gdybyśmy chcieli wypisać liczbę w postaci dwójkowej? W paragrafie o funkcjach rekurencyjnych omawialiśmy krótką funkcję, która potrafiła wypisać w postaci dwójkowej przysłaną do niej liczbę.



Jest jednak jeszcze inny, nowocześniejszy sposób. Biblioteka standardowa C++11 oferuje ciekawy szablon klas o nazwie bitset< >. To coś w rodzaju n-elementowej tablicy obiektów typu bool. Obiekt tej klasy (szablonowej) potrafi pracować na zestawie n bitów. (Liczba n jest parametrem szablonu). Każdy z elementów tej tablicy może przechować informację o konkretnym, pojedynczym bicie.

Jak posłużyć się klasą *bitset* w celu wypisania liczby w postaci dwójkowej?

Wystarczy umieścić tę liczbę w obiekcie wspomnianej klasy (szablonowej) bitset< >, a następnie wysłać ten obiekt strumieniowi. To wszystko, bo w klasie bitset jest przeładowany operator<<, który wypisze wszystkie bity jeden za drugim. W rezultacie zobaczymy na ekranie zapis dwójkowy.

Najpierw zastrzeżenie. Klasa bitset nie jest żadną nowinką C++14, bo jest ona dostępna co najmniej od C++11. Mimo to myślę, że dobrze umieścić ten przykład tutaj, w tym rozdziale, blisko zagadnienia użycia stałych dosłownych dwójkowych, bo tu pewnie zaczniesz poszukiwania, gdy będzie Cię nurtować problem „jak wypisać liczbę dwójkową...”

Oto przykład, w którym zobaczymy, jak to zrobić na kilka sposobów.

```
#include <iostream>
#include <bitset>            // deklaracja szablonu bibliotecznego std::bitset
using namespace std;
//*****************************************************************************************************************
int main()
{
    unsigned short int k = 93;
    bitset<16>  bity {k};                                                          ❶
    cout << bity                                                                   ❷
         << "   Wypisanie pomocniczego obiektu bity " << endl;
    // To samo w jednej instrukcji
    cout << bitset<16>{k}                                                          ❸
         << "   Tylko jedna instrukcja (obiekt chwilowy)" << endl;

    cout << bitset<12>{k}                                                          ❹
         << "         Krocej, bo za pomoca 12 bitow" << endl;

    cout << bitset<sizeof(k) * 8>{k}                                               ❺
         << "   Obliczona liczba bitow potrzebnych dla obiektu k"<< endl;
```

```
    cout << "Tak mozemy (sami!) dopisac separatory cyfr" << endl;
    for(int i = bity.size() - 1 ; i >= 0 ; --i)                          ❻
    {
        cout << bity[i];              // wypisanie pojedynczego bitu     ❼
        if(!(i % 4)  &&  (i > 0) )    // decyzja czy teraz separator     ❽
        {
            cout << "'" ;              // wypisanie separatora (apostrof' )
        }
    }
    cout << endl;
}
```

Na ekranie zobaczymy:

```
0000000001011101   Wypisanie pomocniczego obiektu bity                ❷
0000000001011101   Tylko jedna instrukcja (obiekt chwilowy)           ❸
000001011101        Krócej, bo za pomoca 12 bitow                     ❹
0000000001011101   Obliczona liczba bitow potrzebnych dla obiektu k
Tak mozemy (sami!) dopisac separatory cyfr
0000'0000'0101'1101
```

Omówienie

❶ Ta instrukcja to definicja pomocniczego obiektu o nazwie bity. Jest to obiekt klasy szablonowej bitset<16>. Parametr aktualny 16 oznacza, że obiekt tej klasy nadaje się do pracy z liczbami kodowanymi na co najwyżej 16 bitach. (Wybrałem 16, żeby wypis liczby na ekranie był krótszy, 16-miejscowy) Obiekt bity to w pewnym sensie 16-elementowa tablica obiektów typu bool. Każdy jej element pamiętał będzie stan jednego bitu.

Jak widzimy w definicji ❶, obiekt bity jest od razu inicjalizowany wartością zmiennej k. Co robi on z podaną mu wartością? Przygląda się jej i sprawdza stan kolejnych jej bitów, wstawiając true lub false do odpowiednich elementów swojej 16-elementowej tablicy. Jeśli obiekt ten wyślemy strumieniowi cout ❷, to strumień wypisze na ekranie kolejno wszystkie bity (od najstarszego do najmłodszego). Dzięki temu zobaczymy naszą wartość wypisaną w postaci liczby dwójkowej.

Jak widać, wypisanie wartości w postaci liczby dwójkowej zrealizowaliśmy tu za pomocą dwóch instrukcji ❶ i ❷. Najpierw definiowaliśmy pomocniczy obiekt inicjalizowany żądaną wartością, a następnie ten obiekt wypisywaliśmy na ekranie.

To samo można zrealizować w jednej instrukcji

❸ Oto ona. Widzimy tu, że do instrukcji wypisującej została zapakowana definicja obiektu chwilowego klasy bitset<16>. Obiekt ten jest od razu inicjalizowany wartością k. Tenże obiekt chwilowy zostaje wypisany na ekranie strumieniem cout.

Obiekt chwilowy może być inicjalizowany nie tylko wartością zmiennej typu int – jak u nas. Równie dobrze moglibyśmy tam postawić stałą dosłowną. Na przykład jeśli postawimy liczbę 274, to zostanie ona wypisana w postaci dwójkowej.

```
cout << bitset<16> {274};          // wypisane: 0000000100010010
```

Drobna uwaga kosmetyczna. Gdy spojrzysz na ekran, to zobaczysz, że wypisana wartość została wypisana za pomocą 16 zer lub jedynek. Mimo że liczba jest niewielka, to jednak musimy patrzeć na te początkowe nieznaczące zera. Czy to nie przeszkadza? Najczęściej ma to sens, bo bardzo rzadko używa się tej techniki, by podziwiać jakąś

liczbę w zapisie dwójkowym, gdzie początkowe zera są rzeczywiście zbędne i przeszkadzają. Najczęściej wypisywana wartość jest treścią jakiegoś słowa, którego wszystkie poszczególne bity mają jakieś szczególne znaczenie. W takich sytuacjach jest ważne wypisywanie wszystkich bitów, nawet zawierających „początkowe" zera.

❹ Jeśli jednak naprawdę nie chcesz aż tylu tych początkowych zer, to zastosuj klasę szablonową z odpowiednio mniejszym parametrem. Oto sytuacja, gdy roboczy obiekt jest klasy bitset<12>. Na ekranie możesz zobaczyć, że teraz wypis jest krótszy, 12-bitowy.

Możesz zapytać co by było, gdyby liczba okazała się większa i wymagała w swoim zapisie więcej, niż obecne 12 bitów.

Odpowiedź: pozostałe „najstarsze" bity, które nie mieszczą się w początkowych 12 bitach, zostaną pominięte. Nie zostaną więc wypisane. Czasem nam to bardzo odpowiada.

Jak uniknąć takiego pominięcia, czyli jak sobie zagwarantować wypisanie wszystkich bitów? Jest prosty sposób: parametrem aktualnym klasy szablonowej może być wyrażenie, które samo obliczy, ile bitów potrzebuje wybrany do wypisywania typ obiektu. Jak się pisze takie wyrażenie? Jeśli mamy obiekt o nazwie k, to operator sizeof(k) mówi nam z ilu ***bajtów*** składa się obiekt k. Ponieważ bajt to 8 bitów, zatem liczbę potrzebnych ***bitów*** określa wyrażenie sizeof(k) * 8. To właśnie stawiamy jako parametr szablonu. Praktyczną realizację tego sposobu widzimy w punkcie ❺.

Jak wypisać separatory w liczbie binarnej?

Pokazany sposób wypisywania jest dobry, bo automatyczny, ale niestety nie pozwala on skorzystać z najnowszego krzyku mody, czyli separatora cyfr. Jak pamiętamy, separatora można użyć tylko w zapisie stałej dosłownej, którą umieszczamy w tekście źródłowym programu. Powiedzmy brutalnie: w świecie strumieni we/wy ten separator po prostu nie istnieje.

Jeśli trochę Ci żal, to jest wyjście. Owe separatory możemy dopisać „ręcznie" w prosty sposób. Otóż klasa bitset< > ma przeładowany operator[ ], który daje nam dostęp do każdego wybranego pojedynczego bitu. Skoro tak, to możemy po kolei wypisywać na ekranie wszystkie pojedyncze bity, a przy okazji umieszczać między nimi własne separatory. (Separatorem może to być na przykład spacja lub apostrof).

❻ Oto jak to robimy. Jest tu pętla for przebiegającą po wszystkich bitach obiektu klasy szablonowej bitset<16>. Zwykle nasze pętle przebiegają „rosnąco", czyli od indeksu 0 w górę. Tymczasem sposób pisania liczby (nawet na papierze) jest taki, że najpierw pisze się cyfry najstarsze rangą, a potem coraz młodsze. Skoro tak, to nasza pętla powinna zacząć od bitu najstarszego (tutaj: 15) po czym przebiec 14, 13, ... aż do 1, 0.

❼ W ciele pętli widzimy instrukcję wypisującą na ekranie pojedynczy bit.

❽ Następnie zastanawiamy się czy po tym bicie ma wystąpić separator czy nie. Umawiamy się, że separatory powinny grupować bity w czwórki. „Czwórka" się kończy, gdy wyrażenie !(i % 4) ma wartość true. Ponieważ z powodów estetycznych nie chcemy, by separator wystąpił po (ostatnim) bicie 0, dlatego dodajemy dodatkowy warunek (i > 0). Jeśli koniunkcja obu tych warunków jest spełniona, to separator zostaje wypisany. Na ekranie możesz podziwiać rezultat.

B.2.2 Wczytywanie liczb dwójkowych strumieniem wejściowym

Znowu przypomnę brutalną prawdę: w krainie strumieni wejścia/wyjścia nie istnieją ani separatory cyfr, ani manipulatory pozwalające na wczytywanie liczb dwójkowych. Gdybyśmy mimo to chcieli dać użytkownikowi naszego programu możliwość podania liczby dwójkowej, to sami musimy się zająć odebraniem takiej danej. Oczywistym rozwiązaniem jest przyjęcie takiej liczby w postaci tekstu (stringu). Następnie należy taki tekst przeanalizować i zamienić go na liczbę.

Zobaczymy teraz przykład, w którym będą dwa sposoby realizacji tego zadania.

```
#include <iostream>
#include <string>
#include <bitset>              // deklaracja szablonu bibliotecznego bitset
using namespace std;
//*****************************************************************************************************************
int main()
{
    // Przyjmowanie liczby w systemie dwójkowym z klawiatury

    // Sposób 1. Przyjęcie liczby jako tekstu ————————————————————
    cout << "Podaj liczbe w systemie dwojkowym: ";
    string tekst;                                   // pomocniczy string
    cin >> tekst;                                                          ❶
    bitset<8* sizeof(unsigned long)>  bbb(tekst);                          ❷
    auto liczba = bbb.to_ulong();                                          ❸
    cout << "Przyjeta liczba dwojkowa to dziesiatkowo " << liczba << endl; ❹

    // Sposób 2. Przyjęcie zapisu bezpośrednio do obiektu klasy bitset ——————————
    cout << "Podaj liczbe w systemie dwojkowym: ";
    bitset<8*sizeof(int)>      tmp;                                        ❺
    cin >> tmp;                                                            ❻
    cout << "rezultat jako zestaw bitow: " << tmp << endl;
    int g = tmp.to_ulong();                                                ❼
    cout << "Liczba w obiekcie g wypisana dziesiatkowo = " << g << endl;
}
```

Na ekranie zobaczymy:

```
Podaj liczbe w systemie dwojkowym: 111
Przyjeta liczba dwojkowa to dziesiątkowo 7
Podaj liczbe w systemie dwojkowym: 1001
rezultat jako zestaw bitów: 00000000000000000000000000001001
Liczba w obiekcie g wypisana dziesiatkowo = 9
```

Porozmawiajmy, jak działają oba te sposoby

Sposób pierwszy. Przyjęcie liczby jako tekstu

❶ Prosimy użytkownika programu, aby na klawiaturze napisał liczbę w postaci dwójkowej. To, co napisze, odbieramy w postaci tekstu (składającego się z zer i jedynek) i umieszczamy w pomocniczym obiekcie o nazwie tekst.

❷ Definiujemy obiekt bbb, który jest obiektem klasy szablonowej bitset<64>.

*Jak widać z nazwy, obiekt ten jest zdolny pracować z wartościami zapisanymi dwójkowo za pomocą maksimum 64 cyfr. Wartość 64, czyli (8 * sizeof(unsigned long), jest oczywiście z dużym zapasem, ale to nie przeszkadza.*

Tworzony obiekt bbb jest od razu inicjalizowany tekstem! To tak można? Można, bo:

> klasa szablonowa bitset<64> ma między innymi konstruktor przyjmujący obiekt klasy std::string (lub C-string pokazywany wskaźnikiem char*). Ów konstruktor przygląda się treści przysłanego mu tekstu (czyli zestawowi zer i jedynek) i na tej podstawie ustawia wartości poszczególnych elementów swojej wewnętrznej tablicy bitów. W ten sposób informacja została zapamiętana w obiekcie bbb.

❸ Mając ją, wywołujemy na rzecz obiektu bbb jedną z funkcji składowych jego klasy bitset<...>. Funkcja nazywa się to_ulong. Co ona robi? Patrzy na ustawienie wewnętrznej tablicy „bitów" i rozpoznaje je jako zapis dwójkowy, po czym (jak sama nazwa wskazuje) zwraca rezultat będący odpowiednią liczbą typu unsigned long. Tę liczbę możemy zapamiętać w zmiennej liczba.

Tak oto wczytaliśmy podaną z klawiatury liczbę w zapisie dwójkowym i umieściliśmy ją w zmiennej typu unsigned long. Zadanie wykonane. Na dowód wypisujemy na ekranie wartość zmiennej liczba ❹.

Sposób drugi, czyli krótsza wersja tego, co powyżej

Przedtem wczytywaliśmy zapis dwójkowy do pomocniczego obiektu klasy string, a następnie jego treścią inicjalizowaliśmy obiekt klasy szablonowej bitset. Teraz zrezygnujemy z pomocniczego stringu i wczytamy bezpośrednio do obiektu klasy bitset.

❺ Oto definicja obiektu klasy bitset<64>. W następnej instrukcji ❻,

> strumieniem cin wczytujmy do tego obiektu liczbę dwójkową. Jest to możliwe dzięki mądrości twórców szablonu bitset, którzy wyposażyli ten szablon w operator>> przeładowany właśnie na taką okoliczność.

Jak działa ten operator? Przyjmuje z klawiatury zapis składający się z zer lub jedynek. Jeśli natknie się na jakiś inny znak, uzna że w tym miejscu wczytywanie liczby się skończyło. Przerwie też wczytywanie gdy tych zer i jedynek będzie więcej niż określa liczba będąca parametrem szablonu (tu: 64).

❼ Mamy więc wczytany ten zapis do obiektu klasy bitset (podobnie jak w ❸). Teraz więc wywołujemy funkcję składową to_ulong, która zamienia treść obiektu tmp na liczbę. Tę liczbę zapamiętujemy w zmiennej o nazwie g.

W jednym z ćwiczeń na końcu rozdziału poproszę Cię o takie uzupełnienie sposobu przyjmowania liczby dwójkowej, aby użytkownik mógł napisać na klawiaturze liczbę dwójkową, w której dla swej wygody umieścił separatory cyfr (apostrofy).

Ćwiczenie

I W paragrafie B.2.2 zobaczyliśmy sposób pozwalający wczytać z klawiatury liczbę w zapisie dwójkowym. Unowocześnij ten sposób tak, by pozwalał użytkownikowi (dla wygody) umieścić w zapisie separatory cyfr (apostrofy).

B.3 Kompilator rozpoznaje typ rezultatu funkcji

W rozdziale o wyrażeniach lambda zobaczyliśmy, że jeśli sami nie zadeklarujemy typu rezultatu zwracanego przez funkcję (wyrażenie) lambda, to **kompilator spróbuje rozpoznać go sam**. Okazało się to bardzo wygodne.

Może właśnie dlatego wprowadzono do C++14 udogodnienie, polegające na tym, że również i w przypadku zwykłych funkcji kompilator może sam rozpoznać typ rezultatu definiowanej właśnie funkcji. Jak to rozpozna? Spojrzy na instrukcję return znajdującą się w ciele naszej funkcji i określi typ wyrażenia, które przy tym return postawiliśmy. Uzna, że typ rezultatu funkcji ma być właśnie taki.

Jeśli chcemy poprosić kompilator o taką przysługę w stosunku do danej funkcji, należy w deklaracji (definicji) funkcji umieścić słowo auto, w miejscu, gdzie zwykle umieszczamy określenie typu rezultatu.

```
auto fun(int n, double x)                                    ❶
{
        return  n + x;
}
```

Czy czegoś Ci to nie przypomina? W paragrafie 6.5 poznaliśmy tzw. alternatywną deklarację funkcji. Tam też w podobnej sytuacji stawialiśmy słowo auto. Różnica jest taka, że wtedy na końcu tamtej deklaracji należało jednak postawić dodatkowo ->double, czyli „opóźnioną" specyfikację typu rezultatu.

```
auto   funkcja(typ_arg1, typ_arg2 ) -> typ_rezultatu;
```

W C++14 nie musisz robić nawet tego! Wystarczy słowo auto w miejscu określenia typu rezultatu funkcji, a kompilator, patrząc na ciało funkcji, sam spróbuje wydedukować o jaki chodzi typ.

W przypadku funkcji fun ❶ przy słowie return stoi wyrażenie (n + z). Kompilator pomyśli tak:

Skoro n jest typu int, zaś x jest typu double, to wyrażenie (n + z) jest typu double.

Rozpoznawszy typ tego wyrażenia, kompilator uzna, że właśnie taki ma być typ rezultatu funkcji fun. Słowo auto w definicji tej funkcji, kompilator zastąpi więc słowem double.

Zapytasz może: *„A jeśli w naszej funkcji jest więcej niż jedna instrukcja return?"*.

```
auto fun_max(int n, double x)                                ❷
{
    if(n > x) return  n;
    else return x;
}
```

Kompilator przyjrzy się wszystkim instrukcjom return w tej funkcji.

- Jeśli w każdej z nich typ wyrażenia-rezultatu jest taki sam, to świetnie.
- Jeśli jednak kompilator zauważy, że w tej funkcji nie wszystkie wyrażenia stojące w instrukcjach return są tego samego typu, to zaprotestuje. Słusznie uzna, że w tej sytuacji nie może jednoznacznie wywnioskować o jaki typ chodzi. Zażąda więc, abyśmy w deklaracji wyraźnie napisali jaki ma być typ rezultatu tej funkcji.

Taka właśnie niejednoznaczność wystąpiła w powyższej funkcji fun_max ❷. W pierwszej instrukcji return kompilator rozpoznał wyrażenie typu int, a w drugiej – rozpoznał wyrażenie typu double. Odpowiedział mi więc takim komunikatem o błędzie:

```
error: insconsistent deduction for auto return type: int and then double
```

czyli po polsku:

```
błąd: niespójna dedukcja typu rezultatu funkcji oznaczonego jako auto:
najpierw rozpoznany jako int, a później jako double.
```

> *Zwracam uwagę, że samo umieszczenie różnych typów wyrażeń (przy instrukcjach* return*) – nie było błędem. Wolno nam tak zrobić. Możliwe, że celowo umieściliśmy przy instrukcjach* return *wyrażenia nieco innego typu, słusznie licząc na to, że i tak nastąpi zwykła konwersja na typ wyszczególniony jako typ rezultatu tej funkcji. Zatem nie to jest błędem.*

Błąd polega na tym, że w takiej niejednoznacznej sytuacji nie możemy zlecać kompilatorowi, aby to on rozpoznawał jaki ma być typ rezultatu funkcji. Przecież dojdzie on do sprzecznych wniosków! Wiedząc o tym, powinniśmy zrezygnować z automatycznego rozpoznawania i samemu grzecznie określić typ rezultatu tej funkcji.

❦

Dowiedzieliśmy się właśnie, że kompilator może sam rozpoznać typ rezultatu funkcji na podstawie wyglądu instrukcji return będących w ciele tej funkcji. Skoro „w ciele", to znaczy, że musi to być *definicja* funkcji. Sama deklaracja nie wystarcza. Można jednak słowo auto postawić w samej deklaracji funkcji i nie jest to błędem. Jeśli jednak po takiej tylko deklaracji kompilator zobaczy w programie wywołanie tej funkcji, to zaprotestuje. Przecież jeszcze nie miał szansy się wykazać i rozpoznać typu rezultatu tej funkcji, więc taka deklaracja jest w pewnym sensie niepełna.

> Wywołanie takiej funkcji w programie możemy postawić dopiero w miejscu, gdzie kompilator już zdążył zapoznać się z definicją (czyli ciałem) tej funkcji, więc określił już typ jej rezultatu.

Dla dociekliwych

Może być jednak sytuacja pośrednia. Wyobraź sobie, że mamy definicję funkcji napisaną tym sposobem, czyli w pierwszej linijce definicji, zamiast typu rezultatu stoi słowo zastępcze auto. Kompilator się jednak nie niepokoi, bo przecież za chwilę zobaczy jakąś instrukcję return, która pomoże mu określić prawdziwy typ rezultatu funkcji. Tymczasem dostaje cios w plecy: funkcja okazuje się być rekurencyjna (§6.19, str. 228) czyli nagle wywołuje samą siebie. Jak biedny kompilator może teraz sprawdzić poprawność tego (rekurencyjnego) wywołania, skoro jeszcze żadnej instrukcji return nie napotkał!

Jest na to rada. Trzeba wtedy napisać definicję tej funkcji tak, żeby kompilator, przed tą newralgiczną linijką (rekurencyjnego) wywołania, zobaczył choćby jedną instrukcję return.

```
auto  zgadywanka(int n)
{
    if(0 > 55) return '*';      // Co prawda warunek nigdy nie spełniony, ale to i tak nam
                                //  wystarcza, bo dzięki temu kompilator widzi tu
                                // instrukcję return (i ocenia typ stojącego przy niej wyrażenia).

    if(n > 4) zgadywanka(n − 1);      // wywołanie rekurencyjne
```

```
    return 'a';                    // zwykła, „oficjalna” instrukcja return
}
```

B.4 Deklaracja typu rezultatu decltype(auto)

W C++14 pojawia się ciekawa możliwość deklarowania typu rezultatu funkcji za pomocą słów decltype(auto). Sprawa jest bardzo prosta, ale aby docenić korzyść z takiego sposobu deklaracji, dobrze by było przypomnieć sobie paragraf 22.15. Był to jednak paragraf dla ambitnych wtajemniczonych, zatem jeśli w trakcie czytania „*Opusu*” go opuściłeś, to w zasadzie mógłbyś opuścić i ten.

Jak pamiętamy, C++11 pozwala nam na tak zwaną „alternatywną” formę deklaracji funkcji.

```
auto  nazwa_funkcji(argumenty)  -> typ_rezultatu;
```

Polega ona na tym, że w tym miejscu, gdzie zwykle deklarujemy typ rezultatu funkcji, możemy postawić słowo zastępcze auto, a potem (na końcu deklaracji) stawiamy symbol -> i określenie typu rezultatu tej funkcji. W poprzednim paragrafie dowiedzieliśmy się, że C++14 pozwala nam na jeszcze wygodniejszą deklarację funkcji. Mianowicie w ogóle nie jest wymagane określenie typu rezultatu funkcji. Wystarczy, że postawimy w tym miejscu zastępcze słowo auto. Będzie to sygnał dla kompilatora, aby sam sobie rozpoznał jaki jest typ rezultatu danej funkcji. (Może to rozpoznać na podstawie wyglądu instrukcji return obecnych w jej ciele).

> Teraz zobaczymy, że jeśli w przy rzeczonej instrukcji return stoi wywołanie innej funkcji zwracającej coś przez referencję, to kompilator może nas zaskoczyć swoją nadgorliwością.

Oto taka sytuacja. Wyobraź sobie, że mamy dwie funkcje:

```
string      funkcjaOBJ();      ❶
string&     funkcjaREF();      ❷
```

Jak widać, pierwsza z nich ❶ zwraca obiekt klasy string przez wartość. Druga ❷, zwraca referencję do obiektu klasy string, czyli string&.

Wyobraź sobie, że z jakiegoś powodu musisz zdefiniować funkcje, które są niczym innym, jak tylko opakowaniem (futerałem) każdej z nich.

```
string      funkcja_opakowanie_OBJ() { return funkcjaOBJ(); }      ❸
string&     funkcja_opakowanie_REF() { return funkcjaREF(); }      ❹
```

Każda z tych funkcji „opakowujących” wygląda tak, że w swoim ciele po prostu wywołuje swojego „nieopakowanego” odpowiednika. Jego rezultat zwraca jako swój rezultat. Nic w tym nadzwyczajnego. Tak mogliśmy robić do tej pory.

Gdy użyjemy słowa *auto* możemy być zaskoczeni

Wyobraź sobie, że w swoim zachwycie nad C++14 postanowiliśmy, że w definicjach tych obu funkcji opakowujących określimy typ rezultatu słowem auto, co oznacza: „kompilatorze, wykaż się inteligencją i sam rozpoznaj jaki ma być typ rezultatu danej funkcji”. Zamiast definicji ❸ i ❹ napiszemy więc tak:

```
auto    funkcja_opakowanie_OBJ()  { return funkcjaOBJ(); }   ❺
auto    funkcja_opakowanie_REF()  { return funkcjaREF(); }   ❻
```

Okazuje się, że w przypadku tej drugiej funkcji ❻, kompilator uzna, że typem rezultatu funkcji opakowującej ma być string, a nie string&. Postąpi tak zgodnie z zasadami, o których wspomnieliśmy w §3.18. Dziwisz się? No to spójrz na takie instrukcje:

```
int   obiekt_m      = 55;                                          ❼
int & referencja    = obiekt_m;
auto coto           = referencja;  // coto – jest typu int (a nie typu int&)   ❽
```

❽ W tej ostatniej instrukcji widzimy definicję obiektu o nazwie coto.

> Ponieważ w miejscu określenia typu tego obiektu stoi słowo zastępcze auto, więc kompilator spojrzy na wyrażenie inicjalizujące. Jest tam przezwisko (referencja) obiektu_m. Uzna więc, że inicjalizatorem jest (znany pod przezwiskiem) obiekt obiekt_m. Kompilator sięgnie więc do definicji tego obiektu ❼ i od razu zobaczy, że obiekt_m jest typu int. Uzna więc, że definiowany obiekt o nazwie coto ma być obiektem typu int.

Jak widać, (nad)gorliwy kompilator, widząc w wyrażeniu inicjalizującym referencję obiektu typu int, uznał że nikt rozsądny nie tworzy referencji z referencji. Niby racja, ale w przypadku naszej funkcji opakowującej liczyliśmy jednak na tę referencję. Chcieliśmy przecież, aby nasza funkcja opakowująca ❻ miała „dokładnie taki sam typ rezultatu" jak ta funkcja opakowana ❷.

decltype(auto) zmienia sposób w jaki kompilator rozpoznaje typ

Standard C++14 daje nam prosty sposób na rozwiązanie tej niemożliwości „przeniesienia" referencji.

```
decltype(auto)  funkcja_opakowanie_OBJ()
  { return funkcjaOBJ(); }                                        ❾
decltype(auto)  funkcja_opakowanie_REF()
  { return funkcjaREF(); }      // rezultat będzie typu: string&   ❿
```

Jak widać, zamiast słowa zastępczego auto, postawiliśmy teraz wyrażenie decltype(auto). Sprawi ono, że typ rezultatu tej funkcji „opakowującej" będzie dokładnie taki sam, jak typ funkcji stojącej przy instrukcji return.

Ja to sobie zapamiętuję tak: słowo decltype(...) można po polsku przetłumaczyć jako: *typ deklarowany*, czyli dokładnie taki, jaki jest w deklaracji wyrażenia (...). Natomiast auto – oznacza „sam rozpoznaj kompilatorze co tu ma stać".

Złączenie tych obu słów decltype(auto) oznacza takie polecenie: kompilatorze,

auto – najpierw poszukaj w tej funkcji jakie wyrażenie stoi przy return,

decltype – następnie uznaj, że chodzi o dokładnie taki typ, który występuje w deklaracji tego wyrażenia.

Jeśli na przykład przy return stoi wywołanie innej funkcji, to uznaj, że chodzi o dokładnie ten typ, który jest napisany w deklaracji tej innej (opakowanej) funkcji.

W naszym przypadku konkretnych dwóch instrukcji ❾ i ❿ oznacza to, że:

- dla pierwszej funkcji ❾ kompilator uzna, że rezultat ma być typu string (jak poprzednio),
- ale dla drugiej funkcji ❿ kompilator uzna, że typem rezultatu ma być string&, czyli to, o czym marzyliśmy.

Szczęście jest nawet pełniejsze niż przypuszczasz. Gdyby w deklaracji tej opakowanej funkcji typ rezultatu miał jeszcze dodatkowo przydomek const lub volatile, to taki przydomek by się zachował i przeniósł na typ rezultatu funkcji „opakowującej".

Jeśli już ochłonąłeś ze szczęścia, to może naszła Cię refleksja: *„Po co tak cudować i kazać kompilatorowi zgadywać typ rezultatu? Czy nie prościej było w tym przypadku wysilić się trochę i samodzielnie określić typ rezultatu tej funkcji jako string&? W końcu to tylko 7 znaków do wystukania na klawiaturze"* .

Oczywiście że w takiej sytuacji byłoby lepiej. Definicja tej funkcji byłaby wtedy prostsza i jaśniejsza, a to w programie jest zaletą.

> Opisane tu rozwiązanie sprawdzi się dopiero w sytuacji definiowania szablonu funkcji, gdzie nie wiemy jak w przypadku danej specjalizacji określić typ rezultatu funkcji. (Zobaczymy to w następnym paragrafie).

Możliwe, że pomyślałeś teraz tak:

> *Skoro mam to inteligentne narzędzie decltype(auto), to teraz już nigdy nie użyję samotnego słowa auto w typie rezultatu funkcji i zawsze będę używać tego decltype(auto), nawet w przypadku alternatywnej deklaracji funkcji.*

Nie rób tak! W przypadku alternatywnej deklaracji funkcji nie używaj decltype(auto), bo przecież tam, po strzałce ->, jasno napisałeś kompilatorowi jaki ma być typ żądanego rezultatu funkcji. Nie wydawaj więc kompilatorowi sprzecznego polecenia by szukał typu w inny sposób.

Nie używaj *decltype(auto)* w alternatywnej deklaracji funkcji. Tym bardziej, że standard C++17 będzie tego oficjalnie zabraniał.

B.4.1 Przykład zastosowania konstrukcji *decltype(auto)* w szablonie funkcji

W poprzednim paragrafie zobaczyliśmy jak za pomocą decltype(auto) możemy poprosić kompilator o to, by rozpoznał potrzebny nam typ rezultatu funkcji – i żeby ten typ był dokładnie taki, jaki jest w deklaracji wyrażenia stojącego przy instrukcji return.

Zobaczyliśmy to w sytuacji użycia w zwykłej funkcji, ale prawdę mówiąc nikt tak chyba nie robi. Przecież łatwiej i pewniej samemu zdecydować jaki ma być typ rezultatu funkcji i wpisać to do tekstu programu. Zapytasz więc: *No to po co ta heca z konstrukcją decltype(auto)?*

Ten zapis decltype(auto) przydaje się gdy piszemy szablon funkcji fs, w którym typ rezultatu konkretnej funkcji szablonowej fs<T> zależy od aktualnych parametrów szablonu.

Na przykład:

- jeśli parametrem T będzie w danej specjalizacji aktualnie typ X, to rezultat funkcji szablonowej fs<X> ma być np. typu string,
- a jeśli będzie to specjalizacja fs<Y>, to typem jej rezulatu ma być np. string&.

Takie właśnie, **szablonowe** zastosowanie *decltype(auto)* zobaczymy w programie przykładowym

Będzie w nim szablon tak zwanej funkcji opakowującej o nazwie opakowany_tekst< >.

```
#include <iostream>
#include <string>
using namespace std;
//*************************************************************************************************************
// Szablon funkcji, której typ rezultatu zależy od rodzaju klasy, będącej parametrem T
template <typename T>                                                                          ❶
decltype(auto)  opakowany_tekst(T skad)
{
    return   skad.podaj_tekst();                                                               ❷
}
//*************************************************************************************************************
string   astronom { "Kopernik" };                                                              ❸
/////////////////////////////////////////////////////////////////////////////////////////////////////////////
struct Tinformator                                                                             ❹
{
    string  podaj_tekst() {
        return astronom;
    }
};
/////////////////////////////////////////////////////////////////////////////////////////////////////////////
struct Tredaktor                                                                               ❺
{
    string& podaj_tekst() {
        return astronom;
    }
};
//*************************************************************************************************************
int main()
{
    Tinformator inf;                                                                           ❻
    cout << "Za pomoca informatora ----------\n" ;
    cout << opakowany_tekst(inf) << endl;                                                      ❼
    opakowany_tekst(inf) = "Kepler";        // Nie zmienia oryginału, tylko chwilową kopię  ❽
    cout << opakowany_tekst(inf) << endl;                                                      ❾

    Tredaktor red;                                                                             ❿
    cout << "Za pomoca redaktora ----------\n" ;
    cout << opakowany_tekst(red) << endl;                                                      ⓫
    opakowany_tekst(red) = "Edvin Hubble";    // potrafi zmieniać oryginał                   ⓬
    cout << opakowany_tekst(red) << endl;                                                      ⓭

    opakowany_tekst(red) [6] = 'X';           // potrafi zmieniać oryginał                   ⓮
    cout << opakowany_tekst(red) << endl;                                                      ⓯
}
```

Wygląd ekranu po wykonaniu programu

```
Za pomoca informatora ----------
Kopernik                                  ❼
Kopernik                                  ❾
Za pomoca redaktora ----------
Kopernik                                  ⓫
Edvin Hubble                              ⓭
Edvin Xubble                              ⓯
```

Porozmawiajmy o tym programie. Jego istotą jest szablon funkcji...

...o nazwie opakowany_tekst, a konkretnie to, jak w tym szablonie zadeklarować typ rezultatu wytwarzanych z niego funkcji szablonowych.

❶ Jest to szablon o jednym parametrze będącym typem. Co ma robić funkcja szablonowa wyprodukowana z tego szablonu? To można odczytać z ciała tego szablonu.

```
template <typename T>
decltype(auto)  opakowany_tekst(T skad)
{
      return   skad.podaj_tekst();
}
```

Jak widać, szablon ten ma produkować funkcje szablonowe, które jako argument przyjmują obiekt klasy T i na jego rzecz wywołują jego funkcję składową T::podaj.tekst().

Wywołanie to stoi przy słowie return, więc możemy powiedzieć, że wywołanie funkcji podaj_tekst zostało opakowane w funkcję opakowany_tekst.

A teraz najważniejsze. Jaki typ rezultatu powinien być napisany w deklaracji tej funkcji szablonowej opakowany_tekst? Umówmy się, że ma to być taki sam typ, jak typ zwracany przez funkcje składową podaj_tekst. Taki sam, czyli jaki?

Tego się nie da powiedzieć, bo przecież funkcja ta jest funkcją składową typu T będącego *formalnym* parametrem szablonu. Zależnie od tego, jaki to typ będzie *aktualnym* parametrem szablonu, typ rezultatu jego funkcji składowej może być taki lub inny. Nie da się więc tego teraz jasno powiedzieć.

I tu właśnie przydaje się nasza konstrukcja decltype(auto).

> W szablonie funkcji ❶ widzimy, że typ rezultatu tej funkcji szablonowej oznaczony jest zastępczym słowem decltype(auto).

Dzięki temu kompilator otrzymuje informację, aby w trakcie specjalizacji naszego szablonu funkcji opakowany_tekst<T> sam rozpoznał typ rezultatu funkcji składowej T::podaj_tekst, i dokładnie to wpisał w miejsce słów decltype(auto).

To w zasadzie wszystko. Reszta wyjaśnień jest tylko po to, żeby Ci udowodnić, że to naprawdę działa.

Zobaczymy więc teraz specjalizację tego szablonu funkcji dla dwóch nieco odmiennych typów. Łączy je jedno: oba muszą mieć funkcję składową o nazwie podaj_tekst.

❹ Oto klasa (struktura) o nazwie *Tinformator*

Dla uproszczenia jej jedynym składnikiem jest funkcja podaj_tekst. W funkcji tej widzimy, że przy słowie return stoi nazwa globalnego (dla prostoty przykładu) obiektu klasy string o nazwie astronom ❸. Z deklaracji funkcji podaj_tekst w tej klasie widzisz, że funkcja zwraca obiekt astronom przez wartość, czyli przez kopię. (Innymi słowy, rezultat jest typu string).

> *Stąd zresztą pochodzi nazwa tej klasy Tinformator. Informator tylko informuje, ale nie daje szansy na zmianę oryginalnej informacji.*

❺ Na nieco innej zasadzie działa druga klasa o nazwie *Tredaktor*

Ta klasa pozwala oryginalny tekst zmieniać (redagować). Widzimy w tej klasie niemal identyczną funkcję składową podaj_tekst. Jedyną różnicą jest to, że ta funkcja zwraca

obiekt klasy string przez referencję (string&). Jak wiadomo, referencja daje nam dostęp do oryginału.

Zajrzyjmy do funkcji *main*, gdzie skorzystamy z tej klasy szablonowej.

❻ Oto definicja obiektu klasy Tinformator. Ma on nazwę inf.

❼ Wywołanie szablonowej funkcji opakowany tekst. Argumentem tej funkcji jest obiekt klasy Tinformator, zatem

> tutaj kompilator dowiaduje się, że jest potrzebna specjalizowana funkcja szablonowa opakowany_tekst<Tinformator>.

Na ekranie zostaje wypisana treść obiektu astronom, czyli „Kopernik".

❽ Oto kolejne wywołanie funkcji szablonowej opakowany_tekst<Tinformator>. Co to wywołanie robi konkretnie? Pomyśl chwilę, ale nie daj się zwieść wyglądem tej instrukcji. Jak wiesz, rezultatem funkcji szablonowej opakowany_tekst<Tinformator> jest obiekt klasy string. To stoi więc po lewej stronie znaku przypisania. Do tego obiektu przypisujemy to, co po prawej stronie, czyli nową treść "Kepler". No co, wiesz już?...

❾ Aby się przekonać czy to przypisanie zadziałało, w następnej linii jest instrukcja wypisująca na ekranie treść zwróconą przez kolejne wywołanie funkcji opakowany_tekst<Tinformator>(inf). Spójrz na ekran ❾. Zadziałało?

Nie. Nadal „Kopernik"... Czy rozumiesz dlaczego treść astronoma się nie zmieniła?

Zauważ, że w instrukcji ❽ zwrócony został string przez wartość, czyli przez kopię. Zatem instrukcja ta zmieniała się na taką:

chwilowy_obiekt_string = "Kepler"; ❽

Zmiana została więc zrobiona w chwilowej kopii, która już w następnej instrukcji stała się niepotrzebna.

Kolejna instrukcja (❾) zrobiła jeszcze nowszą chwilową kopię (niezmienionego) oryginału i to ją wypisała na ekranie ("Kopernik" ❾).

Zwracam uwagę, że to „niepowodzenie" w przypisaniu Keplera nie jest błędem, klasa Tinformator została wymyślona właśnie w taki sposób, żeby uniemożliwić zmianę oryginału. Celowo funkcja składowa Tinformator::podaj_tekst zwracała rezultat przez kopię, a konstrukcja decltype(auto) wiernie przeniosła ten tak zaplanowany typ rezultatu na funkcję szablonową opakowany_tekst<Tinformator>.

Nieśmiało zwrócę uwagę, że fakt, iż w tym przypadku „decltype(auto) wiernie przeniosła typ rezulatu string na funkcję opakowującą" to żadne osiągnięcie. Zwykłe, gołe auto, w przypadku typu string, potrafiłoby zrobić to samo.

Nie potrafiłoby jednak dokonać tej sztuki w przypadku typu: *referencja do string* (czyli string&). O tym porozmawiamy teraz.

Specjalizacja na rzecz klasy *Tredaktor* jest bardziej wymagająca

❿ Definiujemy obiekt klasy Tredaktor. Jak pamiętamy, w tej klasie jest funkcja składowa podaj_tekst zwracająca referencję string&.

⓫ Wywołujemy funkcję szablonową opakowany_tekst specjalizowaną na okoliczność typu Tredaktor. Widząc to wywołanie, kompilator rozumie, że ma teraz wyprodukować tę specjalizację. Jak ma wyglądać deklaracja tej specjalizowanej funkcji?

Skoro w funkcji szablonowej „opakowującej" określiliśmy typ rezultatu jako decltype(auto), to kompilator, widząc to słowo auto, zrozumie że określenia nie będzie. Sam ma rozpoznać, co stoi przy instrukcji return. (A stoi tam wywołanie funkcji Tredaktor::podaj_tekst).

> Ponieważ wystąpiło słowo decltype, to kompilator rozumie, że rozpoznany typ rezultatu ma być dokładnie taki sam, jaki jest w deklaracji funkcji Tredaktor::podaj_tekst (czyli typ string&).

Sukces! To dzięki decltype(auto) taki właśnie typ (bez ryzyka „zgubienia" referencji) kompilator przeniesie na określenie tej specjalizacji funkcji opakowany tekst.

Wróćmy do tekstu programu, do instrukcji ⓫. Sprawia ona, że na ekranie wypisana zostaje treść obiektu astronom. Proste wypisanie to żadna rewelacja.

⓬ Oto instrukcja, w której dokonujemy prawdziwej modyfikacji treści obiektu astronom. Tę instrukcję ⓬ możemy inaczej zapisać tak:

```
opakowany_tekst(red) = "Edvin Hubble";
```

co możemy w pseudokodzie zapisać jako:

```
referencja astronoma = "Edvin Hubble";
astronom = "Edvin Hubble";                    //   czyli inaczej taka instrukcja
```

Po tej operacji zobaczmy, co jest obecnie treścią astronoma.

⓭ Oto wypisanie na ekranie bieżącej treści obiektu astronom. Jak widać, zmiana nastąpiła.

> Czy to takie wielkie osiągnięcie? Tak, bo gdybyśmy w szablonie zamiast decltype(auto) napisali po prostu auto, to zmiana treści oryginalnego obiektu by nie nastąpiła. W instrukcji ⓬ pracowalibyśmy tylko na chwilowej kopii i to w tej kopii dokonalibyśmy zmiany. Tutaj zaś, w instrukcji ⓭, wypisalibyśmy na ekranie niezmieniony oryginał.

⓮ A teraz ciekawostka. Ta dziwnie wyglądająca instrukcja polega na podmianie w astronomie jednego znaku. Tego, który ma indeks [6]. W pseudokodzie ta operacja wygląda tak:

```
opakowany_tekst(r) [6] = 'X';
referencja astronoma[6] = 'X';          //   pseudokod
astronom[6] = 'X';                      //   czyli inaczej...
```

⓯ Czy ta podmiana jednej litery zadziałała na oryginale? Przekonajmy się. Wypisujemy na ekranie bieżącą treść obiektu astronom i, jak widać, w nazwisku odkrywcy innych galaktyk, litera H została zamieniona na X.

Przypomnijmy wniosek z tego paragrafu:

Dzięki temu, że w funkcji szablonowej opakowany_tekst opisaliśmy typ rezultatu słowami decltype(auto) (a nie „gołym" auto), funkcja ta jest zdolna poprawnie pracować nawet z klasami, które mają funkcje składową podaj_tekst zwracającą obiekt klasy string przez referencję czyli (string&). Użycie decltype(auto) sprawia, że ta referencja nie zostaje „zgubiona", co nam groziło gdyby użyć samego słowa auto.

Zastrzeżenie. Wprawdzie opisane tu narzędzie decltype(auto) wydaje się uniwersalne, ale jego twórcy wyraźnie podkreślają, że:

> decltype(auto) ma służyć do określania typu rezultatu funkcji. Nie powinno się go używać np. w definicjach zwykłych lokalnych obiektów.

B.5 Szablony zmiennych

Omawiając w *„Opusie”* szablony, dowiedzieliśmy się, że mogą istnieć: szablony funkcji, szablony klasy, a nawet szablony aliasu (czyli dodatkowej nazwy typu). C++14 daje nam jeszcze jedną możliwość: szablon definicji zmiennej.

> Szablon zmiennej o nazwie *nnn* mówi kompilatorowi jak (w razie potrzeby) ma nam wyprodukować definicję zmiennej o nazwie *nnn<T>* (gdzie T oznacza parametr szablonu).

Innymi słowy, mając taki szablon definicji zmiennej:

```
template <typename T>
T zmienna;
```

można w programie korzystać na przykład z:

- obiektu typu int o nazwie zmienna<int>,
- obiektu typu double o nazwie zmienna<double>,
- obiektu typu char o nazwie zmienna<char>

i tak dalej...

Jak działa taki szablon do produkcji zmiennej szablonowej?

Gdy kompilator zauważa, że w naszym programie korzystamy na przykład z obiektu o nazwie zmienna<double>, a definicji tak nazwanego obiektu jeszcze nie było, to sięga do szablonu zmienna i na jego podstawie dodaje do programu (szablonową) definicję obiektu o nazwie zmienna<double>.

Wiem, co pomyślałeś: *–Dlaczego po prostu sami nie napiszemy definicji takiego obiektu?* Rzeczywiście, najczęściej tak właśnie robimy. Natomiast...

> Szablon zmiennej stosujemy zwykle po to, by korzystały z niego inne szablony. Na przykład szablony funkcji.

Żeby nam się łatwiej rozmawiało, zejdźmy na poziom konkretnego przykładu

Wyobraź sobie, że pracujesz nad funkcją szablonową, która ma służyć do obliczenia pola powierzchni koła o zadanym promieniu. Niech argumentem tej funkcji będzie promień koła. Parametrem tego szablonu funkcji ma być *typ* argumentu przynoszącego informację o promieniu koła.

```
template <typename T>
T pole_kola_o_promieniu(T promień);                    // deklaracja szablonu funkcji
```

Dzięki parametrowi T szablon ten będzie zdolny do wyprodukowania rodziny funkcji, które potrafią liczyć pole koła o promieniu zadanym wartością typu double, albo wartością typu float, albo nawet wartością typu int.

Jak będzie się odbywać takie obliczanie? To pamiętasz ze szkoły. Mając zadany promień r nasza funkcja oblicza pole powierzchni koła według prostego wzoru: r^2

```
template <typename T>
T pole_kola_o_promieniu(T r);                          // definicja szablonu funkcji
{
        return (3.1415926535897932385 * r * r);
}
```

Niby sprawa jest bardzo prosta. Jeśli promień r zadany jest w postaci liczby typu double to r * r po pomnożeniu przez wartość (podaną jako stała dosłowna typu double) da rezultat typu double. Tę obliczoną wartość funkcja ma zwrócić jako rezultat. Sprawa załatwiona.

Zwróć uwagę, że pisząc definicję szablonu zdecydowaliśmy, że funkcja szablonowa ma zwracać rezultat takiego samego typu jak typ zadanego promienia (T), czyli w tym przypadku ma być to typ double. Nie ma problemu, bo takiego właśnie typu jest wartość wyrażenia (3.14159265358979323 85 * r * r). Wspaniale.

Niestety nie zawsze. Co będzie, gdy ktoś wywoła naszą funkcję szablonową podając promień koła w postaci wartości typu float, czyli na przykład tak:

```
pole_kola_o_promieniu(1.5f);                    // wywołanie funkcji szablonowej
```

Skoro argument wywołania jest teraz typu float, to funkcja szablonowa zostaje specjalizowana dla parametru typu float. Co prawda promień podniesiony do kwadratu to nadal wartość typu float, ale przecież to zostanie pomnożone przez stałą dosłowną typu double (3.14159265358979323 85), więc wynik mnożenia będzie typu double.

Tymczasem nasz szablon wymaga, by funkcja szablonowa pole_kola_o_promieniu<float> zwracała rezultat typu float (czyli tego samego typu, co typ zadanego promienia koła). Skoro tak, to kompilator musi dodatkowo zamienić tę właśnie obliczoną wartość typu double na wartość typu float. Zauważ: najpierw zamiana z typu float na double, a potem z powrotem z double na float. To marnotrawstwo.

Dlaczego do tej pierwszej zmiany typu w ogóle doszło? Dlatego że wartość liczby pi była innego typu (double) niż typ promienia koła (float).

> Temu marnotrawstwu można zaradzić, jeśli przygotuje się w programie kilka różnych wersji wartości liczby pi. Osobną w wersji double, osobną w wersji float. Może nawet wersję long double.

Otóż dzięki C++14 nie musimy takich różnych wariantów liczby pi definiować ręcznie. Zrobi to za nas kompilator, jeśli dostarczymy mu taki szablon zmiennej:

```
template <typename T>
T pi = T( 3.14159265358979323 85 );
```

Jest to szablon definicji zmiennej typu T o nazwie pi. Dzięki niemu będziemy mogli posługiwać się w programie na przykład obiektami pi<double>, pi<float>, pi<long double>, pi<int> itd.

Jak widać, w tym szablonie jest definicja zmiennej, mającej od razu inicjalizację. Co jest inicjalizatorem? Długa stała dosłowna 3.14159265358979323 85 rzutowana na typ T.

Oznacza to, że jeśli parametrem aktualnym szablonu T będzie na przykład float, to definicja pi<float> przyjmie postać:

```
float     pi<float> = float(3.14159265358979323 85);        // specjalizacja dla float
```

w przypadku innych specjalizacji:

```
double    pi<double> = double(3.14159265358979323 85);      // specjalizacja dla double
int       pi<int> = int(3.14159265358979323 85);            // specjalizacja dla int
```

Drobna uwaga: oczywiście wiesz, że liczba jest stałą i do tego taką „odwieczną”, czyli w zasadzie w naszym szablonie można by dodatkowo postawić przydomek constexpr. Nie stawiam go jednak tutaj celowo, bo zauważyłem, że w wielu artykułach

internetowych przyjmuje się za oczywistość, iż szablony, o których teraz rozmawiamy, służą do produkcji stałych constexpr.

> Wiedz, że w standardzie C++14 nie ma takiego ograniczenia.

Szablon zmiennej może mieć przydomek constexpr, ale nie musi. Zatem szablon zmiennej może Ci nawet służyć do produkcji *prawdziwych* zmiennych. Takich, do których będziesz czasem coś wpisywał, a czasem coś z nich odczytywał.

Zobaczmy teraz krótki program, w którym wystąpi omawiany szablon zmiennej *pi*

Przykład ten ma pokazać Ci:

1. Jak się definiuje szablon zmiennej.
2. Jak się z niego korzysta w zwykłych, nieszablonowych sytuacjach.
3. Jak się z niego korzysta z wnętrza innego szablonu (z szablonu funkcji).
4. Dodatkowo zobaczymy ciekawą rzecz. Nasz szablon zmiennej pi może mieć nawet specjalizację zdefiniowaną „ręcznie" przez użytkownika.

```
#include <iostream>
#include <string>
using namespace std;
//*************************************************************************************************************
template <typename T>                    // szablon zmiennej pi                                    ❶
constexpr  T pi = T( 3.1415926535897932385 );
//*************************************************************************************************************
// możliwa nawet taka zwariowana specjalizacja powyższego szablonu zmiennej
template < >                                                                                       ❷
std::string     pi<std::string>  ("LICZBA PI");
//*************************************************************************************************************
template <typename T>                    // szablon funkcji
T pole_kola_o_promieniu(T r)                                                                       ❸
{
    return pi<T> * r * r;        // użyje  pi  specjalizowanego dla typu T                        ❹
}
//*************************************************************************************************************
int main()
{
        cout.precision(10);                                                                        ❺

        cout << "Wartosc obiektu pi<double> = " << pi<double> << endl;                             ❻
        cout << "Wartosc obiektu pi<float> = " << pi<float> << endl;                               ❼
        cout << "Wartosc obiektu pi<int> = " << pi<int> << endl;                                   ❽
        cout << "Wartosc obiektu pi<string> = " << pi<string> << endl;                             ❾

        cout << "Pole kola dla argumentow roznego typu\n";
        cout << pole_kola_o_promieniu(2.0) << " -> double " << endl;                               ❿
        cout << pole_kola_o_promieniu(2.0f) << " -> float" << endl;                                ⓫
        cout << pole_kola_o_promieniu(2) << " -> int " << endl;                                    ⓬
}
```

Po wykonaniu programu, na ekranie zobaczymy taki tekst:

```
Wartosc obiektu pi<double> = 3.141592654
Wartosc obiektu pi<float> = 3.141592741
Wartosc obiektu pi<int> = 3
```

```
Wartosc obiektu pi<string> = LICZBA PI                    ❾
Pole kola dla argumentow roznego typu
12.56637061 -> double                                     ❿
12.56637096 -> float                                      ⓫
12 -> int                                                 ⓬
```

Omówienie programu

❶ Oto szablon zmiennej. Szablon nazywa się pi, a będzie służył kompilatorowi do produkcji zmiennych o nazwach pi<T> gdzie T jest parametrem formalnym oznaczającym typ.

Korzystanie ze zmiennej szablonowej w zwykłych partiach programu

❻ Tu jest dowód, że nasz szablon działa nawet w zwykłych (nieszablonowych) wyrażeniach. Jak widać, instrukcją cout wypisujemy na ekranie wartość zmiennej szablonowej o nazwie pi<double>. W następnej instrukcji (❼) wypisujemy wartość zmiennej pi<float>. Są to oczywiście różne obiekty, przechowujące wartość liczby z właściwą sobie dokładnością.

> *Zwykle wartości zmiennoprzecinkowe wypisuje się z dokładnością 6 miejsc. Tymczasem wartości zapisane w obiektach pi<double> i pi<float> są tak do siebie zbliżone, że różnią się dopiero na dalszych miejscach po przecinku. Abyś mógł mimo wszystko tę różnicę zobaczyć, umieściłem wcześniej instrukcję ❺ sprawiającą, że strumień cout wypisze wartości zmiennoprzecinkowe za pomocą aż dziesięciu cyfr.*

❽ Jeśli poprosimy o w postaci liczby całkowitej typu int, czyli o wartość zapisaną w obiekcie pi<int>, to zobaczymy wartość zaokrągloną do 3. Nie pytaj mnie po co to komu...

Użytkownik „ręcznie” specjalizuje szablon zmiennej *pi*

❷ A teraz ciekawa rzecz. Jak wiesz, dla szablonu funkcji (albo szablonu klas) użytkownik może sam napisać NIE-szablonową specjalizację na okoliczność wybranych przez siebie konkretnych parametrów. Podobnie można postąpić wobec szablonu zmiennej.

> Użytkownik (czyli Ty) może sam napisać specjalizację szablonu zmiennej pi dla wybranego przez siebie typu.

Jeśli będzie na przykład dla typu std::string, to powstanie ręcznie zrobiona specjalizacja zmiennej pi<std::string>

Tu właśnie ❷ widzisz, jak prosto się to robi. Użytkownik zdecydował tutaj, że wartością obiektu typu pi<std::string> ma być tekst "LICZBA PI".

❾ A to jest instrukcja, w której z tej tekstowej wersji pi korzystamy. Wypisujemy na ekranie wartość zmiennej pi<string>. Jak widać, tekst został wypisany poprawnie, co dowodzi, że nawet taka specjalizacja szablonu zmiennej pi jest możliwa.

Korzystanie ze zmiennej szablonowej w „szablonowych partiach programu”

Do tej pory tylko zdefiniowaliśmy kilka (szablonowych) obiektów z rodziny pi<T>, ale korzyść z nich jest mała. Równie dobrze moglibyśmy je zdefiniować „na piechotę” w postaci pi_double, pi_float itd. Teraz zobaczymy do czego naprawdę ten nasz szablon może zostać wykorzystany.

Jak zapowiadałem, te całe sztuczki z szablonami zmiennych tak naprawdę przydają się w sytuacji, gdy z szablonowych zmiennych mamy skorzystać z wnętrza innych szablonów.

❸ Oto definicja szablonu funkcji pole_kola_o_promieniu. Jej argument r służy do przekazania funkcji informacji o długości promienia koła. Typ argumentu r jest parametrem tego szablonu funkcji. Dzięki temu promień koła może zostać zadany czasem za pomocą zmiennej typu double, czasem za pomocą typu float, a czasem np. typu long double.

Rolą szablonu funkcji, który tu definiujemy, jest wytwarzać funkcje umiejące obliczać pole koła o zadanym promieniu. Dodatkowo chcemy, aby (uwaga!) obliczone pole koła (czyli rezultat funkcji szablonowej) było wartością takiego samego typu jak typ zadanego promienia r.

❹ Praca tych funkcji szablonowych polega na prostym obliczaniu wyrażenia według wzoru r^2

Dodatkowe wymaganie jest takie, że do tego obliczenia powinniśmy użyć („zewnętrznej") wartości pi zapisanej w takim obiekcie (szablonowym), którego typ zgadza się z typem promienia r. Tak też się stanie, bo widzimy wyrażenie ❹:

```
return pi<T> * r * r;
```

a z tego wynika, że:

- specjalizacja funkcji pole_kola_o_promieniu<double> skorzysta z pi<double>,
- specjalizacja funkcji pole_kola_o_promieniu<float> skorzysta z pi<float>

i tak dalej.

Na ekranie możesz zobaczyć potwierdzenie, że tak rzeczywiście było. Są tu wydrukowane wartości obliczone dla promieni zadanych w postaci wartości

typu double – 10,
typu float – 11,
typu int – 12.

Jak widać to działa. Nasza funkcja szablonowa potrafi skorzystać z właściwego sobie wariantu szablonowej zmiennej pi.

❦

Zobaczyliśmy tu w dość przejrzysty sposób, jak się definiuje szablon zmiennej. To proste dwie linijki programu.

Na tym stwierdzeniu można by w zasadzie zakończyć omawianie tego zagadnienia. Wtedy jednak zadałbyś mi pytanie, które ludzie ciągle zadają na forach dyskusyjnych: *„po co właściwe są te szablony zmiennych"*. Do tej pory odniosłeś chyba wrażenie, że jest to raczej tylko jakieś sympatyczne udogodnienie.

Trudno byłoby Ci się zgodzić z twórcami tego narzędzia, gdy piszą, że **jest to jedna z najważniejszych innowacji wprowadzonych do C++14**.

Aby naprawdę zrozumieć co daje nam to nowe narzędzie, musimy porównać jak programiści musieli sobie radzić w czasach, gdy tego narzędzia nie było. Potem zobaczymy o ile łatwiej i przejrzyściej można ten sam efekt osiągnąć, używając szablonu zmiennej.

B.5.1 Jak to drzewiej bywało?

Wyobraź sobie, że pracujesz nad jakimś szablonem funkcji fun.

```
template <typename T>
T fun(T argument);
```

W ciele tego szablonu chciałbyś korzystać z jakiejś zewnętrznej zmiennej x. Na przykład po to, by w niej jakąś informację zapisać lub żeby coś z niej odczytać. Nie ma problemu, to znamy od dawna. Z ciała funkcji szablonowej fun, wolno nam się odnosić do zmiennych zewnętrznych (np. globalnych).

Niestety, w naszym zadaniu jest dodatkowe wymaganie utrudniające sprawę. Otóż typ zmiennej x (potrzebnej szablonowi fun) zależy od tego, dla jakich parametrów funkcja szablonowa fun została w danym przypadku specjalizowana. Słowem: typ zmiennej 'x' ma zależeć od aktualnych parametrów funkcji szablonowej fun.

> *Czyli na przykład: dla funkcji szablonowej fun<double> zmienna x ma być typu double. Dla funkcji szablonowej fun<char> ma być typu char. I tak dalej.*

Inaczej mówiąc, typ potrzebnej nam zmiennej x jest także określony parametrem szablonu. (Mówimy krócej: typ ten jest „parametryzowany").

Powiesz pewnie: *„Skoro tak, to wystarczy, żeby taka zmienna x była zdefiniowana w obrębie naszego szablonu funkcji fun. Wówczas przy jej definicji określimy, że jest ona tego samego typu co parametr szablonu, czyli typu T"*.

```
template <typename T>
T fun(T argument)
{
    T   x;          // definicja zmiennej x
    // ...
}
```

Rzeczywiście, tak byłoby najlepiej. Niestety, czasem tak nie można, bo z jakichś powodów zmienna x musi naprawdę leżeć na zewnątrz naszego szablonu fun.

Jak to wówczas zrealizować? Jak sprawić, że jeśli w programie kompilator zobaczy gdzieś wywołanie funkcji fun dla argumentu typu double to nie tylko, że wytworzy specjalizację funkcji fun<double>

> ale także wytworzy potrzebną jej „zewnętrzną" zmienną x typu double.

Innym razem, gdy kompilator zobaczy wywołanie funkcji fun dla argumentu typu float, to oprócz specjalizacji fun<float>,

> dodatkowo wytworzy dla niej zmienną x typu float.

Zobaczymy teraz jak z tym problemem radzili sobie programiści do tej pory. Zobaczymy dwa sposoby rozwiązania.

- Sposób pierwszy: za pomocą dodatkowego szablonu funkcji.
- Sposób drugi: za pomocą dodatkowego szablonu klasy. (Ten sposób zobaczymy w dwóch wariantach).

Załóżmy, że znów chodzi liczbę pi. Ma być ona różnego typu. Czasem będziemy potrzebowali jej w postaci liczby typu double, czasem w postaci liczby typu float i tak dalej. Nie zawsze chodzi o to, żeby taki „zewnętrzny" obiekt był rzeczywiście zmienną, czasem wystarczy, żeby była to stała. Jeśli wystarczy stała, to wtedy zadanie się upraszcza. Taką sytuację przedstawia sposób 1.

Sposób 1) – za pomocą dodatkowego szablonu funkcji

> Potrzebną nam parametryzowaną stałą możemy zrealizować jako szablon (innej) funkcji on nazwie pi. Funkcja taka powinna zwracać wartość liczby pi w postaci wartości określonego typu.

Oto jak prosto zdefiniować taki szablon funkcji:

```
template<typename T>
T pi() { return 3.1415926535897932385; }
```

Zwróć uwagę, że teraz mamy już w programie dwa szablony funkcji. Szablon funkcji fun<T> i szablon funkcji pi<T>. Wyobraź sobie, że gdzieś w ciele szablonu funkcji fun<T> umieściłeś wywołanie funkcji szablonowej: pi<T>().

Prześledźmy teraz jak na widok tego wywołania zachowa się kompilator. Kompilator przystępuje do pracy nad tym programem i...

✧ Jeśli zobaczy w nim wywołanie fun(9.9), to słusznie uzna, że skoro 9.9 jest argumentem typu double, to znaczy, że jest potrzebna specjalizacja fun<double>. Zabierze się więc do wytwarzania jej z szablonu fun.

✧ W trakcie pracy nad tą funkcją fun<double> dojdzie do tego miejsca w jej ciele, gdzie powinno stać teraz wywołanie funkcji pi<T>().

Skoro w obecnej specjalizacji funkcji fun<T> parametr aktualny T oznacza typ double, więc uzna, że w tym miejscu ma postawić wywołanie funkcji:

```
pi<double>()
```

Czy jest w programie taka funkcja? Kompilator wie, że takiej funkcji co prawda jeszcze nie ma, ale przecież zna już szablon do jej wytworzenia.

✧ Zatem z tego szablonu wytworzy potrzebną mu teraz specjalizację pi<double>. Dzięki temu wywołaniu potem, już w trakcie wykonywania programu, w tym miejscu funkcji fun<double> pojawi się wartość stałej pi będąca typu double.

> Wspaniale, prawda? Kompilator widząc, że zamierzamy skorzystać z funkcji szablonowej fun<double> sam uznał, że będzie jej dodatkowo potrzebna funkcja pi<double> i ją wytworzył.

Nie muszę chyba mówić, że bardzo podobnie będzie w sytuacji, gdy gdzieś indziej w programie kompilator zobaczy wywołanie funkcji fun(9.9f). Wytworzy wtedy dodatkowo funkcję szablonową pi<float>.

Podsumowanie sposobu 1)

Zaleta: Sposób jest bardzo łatwy. Wymaga od nas tylko dodatkowego zdefiniowania prostego szablonu funkcji.

Wada 1. Tym prostym sposobem tym możemy się posłużyć w przypadku, gdy ten zewnętrzny obiekt dostarczać ma nam stałą wartość. W przypadku gdyby to miała być prawdziwa zmienna, czyli taka, do której chcemy coś zapisywać, a potem z niej odczytywać, to ten sposób się nie nadaje.

Wada 2. Dezorientujące nawiasy. Jeśli nawet nazwa tego dodatkowego szablonu funkcji dobrze i zgrabnie określa nazwę obiektu, to i tak, aby z tego obiektu skorzystać użyć zapisu z dodatkowymi nawiasami wywołania funkcji:

```
pi<T>()          // Gdy wywołanie jest z:
                 //     – wnętrza innego szablonu o parametrze T
```

```
pi<double>()        //  – z ciała zwykłej, nieszablonowej funkcji
pi<float>()         //  – z ciała zwykłej, nieszablonowej funkcji
```

Trochę dziwne te nawiasy w przypadku obiektu, prawda? Nie jest to eleganckie. Ktoś może powiedzieć: *„To ma być (zmienny/stały) obiekt? Przecież widać, że to funkcja"* .

Aby tego nieszczęsnego nawiasu uniknąć, możemy tę całą sprawę rozwiązać w inny sposób.

Sposób 2. Zmienna szablonowa zrealizowana jako składnik-dana dodatkowego szablonu klasy

Potrzebną szablonowi fun zewnętrzną zmienną możemy zrealizować jako składnik odrębnego szablonu klasy. Będzie on parametryzowanego typu, czyli jego typ będzie wynikać z parametru jej szablonu.

Zrobimy to jeszcze sprytniej. Zwykle, aby móc skorzystać ze składnika klasy musimy najpierw mieć obiekt tej klasy. Zwykle, ale nie zawsze. Jeśli składnik-dana będzie statyczny, to będzie istniał nawet wtedy, gdy żaden obiekt tej klasy nie istnieje. (Taka jest przecież natura składników statycznych).

Zaraz to wszystko zobaczymy w przykładowym programie. Zbudujemy prosty szablon klasy, który będzie miał tylko jeden składnik statyczny. To ten składnik będzie naszą zmienną do przechowywania liczby pi wybranego typu.

```
#include <iostream>
#include <string>
using namespace std;
//*************************************************************************************************************
// Sposób 1)  pi zrealizowane jako szablon funkcji
template<typename T>
T pi1() { return 3.1415926535897932385; }                                                          ❶
//*************************************************************************************************************
// Sposób 2a)  pi za pomocą szablonu klasy ze składnikiem statycznym
// umieszczonym poza ciałem klasy.
template<typename T>
struct  pi2a                                                                                       ❷
{
        static T  wartosc;  // patrz. Opus, str. 572 (tom I)
};
// a na zewnątrz szablonu klasy jest definicja tego składnika
template<typename T>
T pi2a<T>::wartosc = 3.1415926535897932385;                                                        ❸
//////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Sposób 2b)  pi zrealizowane jako szablon klasy (struktury) ze składnikiem statycznym
// inicjalizowanym "w klasie".
template<typename T>
struct  pi2b                                                                                       ❹
{
        static constexpr T  wartosc = 3.1415926535897932385;
};
//*************************************************************************************************************
// Sposób 3), najlepszy: za pomocą szablonu zmiennej.  (C++14)
template <typename T>
T pi = T(3.1415926535897932385);                                                                   ❺
```

```
//*************************************************************************************************************
// Szablon funkcji, w którym z tego pi musimy korzystać
template<typename T>                                                                              ❻
void fun(T arg)
{
    cout << "Obliczenia w funkcji szablonowej \n";
    T wynikA = arg * pi1<T>() ;                                                                   ❼
    cout << "\tSposobem 1. \tWynikA to " << wynikA << endl;

    T wynikB = arg * pi2a<T>::wartosc;                                                            ❽
    cout << "\tSposobem 2a. \tWynikB to " << wynikB << endl;

    T wynikC = arg * pi2b<T>::wartosc;                                                            ❾
    cout << "\tSposobem 2b. \tWynikC to " << wynikC << endl;

    T wynikD = arg * pi<T>;                                                                       ❿
    cout << "\tSposobem 3. \tWynikD to " << wynikD << endl;
}
//*************************************************************************************************************
int main()
{
    fun(1.0);           // fun<double>                                                            ⓫
    fun(1.0f);          // fun<float>                                                             ⓬
    fun(1);             // fun<int>                                                               ⓭
}
```

Wykonanie programu spowoduje, że na ekranie wypisany zostanie tekst:

```
Obliczenia w funkcji szablonowej
    Sposobem 1.     WynikA to 3.14159
    Sposobem 2a.    WynikB to 3.14159
    Sposobem 2b.    WynikC to 3.14159
    Sposobem 3.     WynikD to 3.14159
Obliczenia w funkcji szablonowej
    Sposobem 1.     WynikA to 3.14159
    Sposobem 2a.    WynikB to 3.14159
    Sposobem 2b.    WynikC to 3.14159
    Sposobem 3.     WynikD to 3.14159
Obliczenia w funkcji szablonowej
    Sposobem 1.     WynikA to 3
    Sposobem 2a.    WynikB to 3
    Sposobem 2b.    WynikC to 3
    Sposobem 3.     WynikD to 3
```

W tym programie zebrane są różne sposoby rozwiązania naszego problemu

❻ Oto szablon funkcji o nazwie fun, w którym potrzebujemy skorzystać z (parametryzowanej) zmiennej, znajdującej się gdzieś poza tym szablonem. W celach dydaktycznych w ciele tego szablonu zrobimy to kilkakrotnie, za każdym razem innym sposobem.

Sposób 1: za pomocą dodatkowego szablonu funkcji

❼ Jest to wyrażenie, będące prostym mnożeniem, w którym potrzebna jest nam wartość liczby . W wyrażeniu tym korzystamy z liczby uzyskanej omówionym wcześniej sposobem 1, czyli tę wartość (odpowiedniego typu) dostarcza funkcja szablonowa.

❶ Oto definicja szablonu takiej funkcji dostarczającej nam liczbę pi.

Zauważ, że w nazwie tej funkcji umieściłem teraz dodatkowo jedynkę (pi1), co ma nam przypominać, że to realizacja sposobem nr 1.

Patrząc na ciało tego szablonu funkcji widzimy, że powstałe z niego funkcje szablonowe zwracają stałe dosłowne reprezentujące wartość liczby pi różnego typu. Ich typ (czyli typ rezultatu funkcji szablonowej) jest określony parametrem aktualnym szablonu.

Dokładniej omówiliśmy już tę sytuację poprzednio.

Wiemy, że ten sposób z dodatkowym szablonem funkcji ma dwie wady. Pierwsza wada to fakt, że sposób ten nadaje się właściwie tylko do dostarczania stałych wartości (taką jest liczba pi). Druga wada to ten dezorientujący nawias wywołania funkcji, który musimy stawiać w wyrażeniu wywołującym ❼.

Sposób 2a

Zobaczmy teraz (stosowany do niedawna) inny sposób rozwiązania problemu tworzenia zmiennych parametryzowanego typu.

❷ Oto definicja pomocniczego szablonu klasy. Szablon ten ma nazwę pi2a, co ma nam przypominać, że jest to realizacja pi sposobem 2. (Dlaczego w tej nazwie jest dodatkowo litera a, wyjaśni się za chwilę).

Jest to bardzo prosty szablon klasy, ma tylko jeden składnik o nazwie wartosc. Typ tego składnika jest określony jako T, czyli składnik ten ma być typu takiego jak konkretny parametr aktualny szablonu klasy. Dodatkowo składnik ten ma przydomek static.

Pamiętasz chyba, że jeśli składnik-dana jest statyczny, to w klasie jest tylko jego deklaracja, a definicję tego składnika musimy umieścić gdzieś poza ciałem klasy (szablonu klasy).

Jeśli nie pamiętasz, to rozmawialiśmy o tym w „Opusie" w paragrafie 37.9.2 pt. „Składniki statyczne w szablonie klas".

❸ Tu właśnie umieszczamy definicję statycznego składnika szablonu. Jak widać, składnik wartosc jest inicjalizowany wartością liczby .

To tyle przygotowań. Zajrzyjmy ponownie do funkcji main.

⓫ Jest tu wywołanie funkcji fun dla argumentu typu double, a to znaczy, że niniejszym kompilator dowiaduje się o konieczności zaistnienia funkcji szablonowej fun<double>. Jeśli takiej funkcji nie zrobił wcześniej, to teraz sięgnie do szablonu ❻ i zacznie jego specjalizację dla typu double. W trakcie tej pracy natknie się na...

❽ ...proste mnożenie, w którym chcemy skorzystać z wartości pi zrealizowanej sposobem 2a. Będzie to wyrażenie pi2a<T>::wartosc. Kompilator wie, że w specjalizacji funkcji fun, którą teraz wytwarza, parametr T oznacza teraz typ double, zatem stojące w niej wyrażenie pi2a<T>::wartosc oznacza obecnie pi2a<double>::wartosc. *„Takiej specjalizacji klasy szablonowej jeszcze w programie nie ma"* – pomyśli kompilator – *„ale to nic, skoro znam szablon służący do jej wyprodukowania (to szablon ❷)"*.

Kompilator weźmie się więc teraz do pracy i dodatkowo tę klasę szablonową pi2a<double> zdefiniuje.

W ramach tej pracy powstanie definicja składnika statycznego ❸ o nazwie pi2a<double>::wartosc. Ten składnik statyczny będzie typu double. Zostanie inicjalizo-

wany wartością 3.1415926535897932385. Na tym skończy się ta „dodatkowa" praca kompilatora.

Mając już definicję tej (dodatkowej) klasy pi2a<double> kompilator może wrócić do pracy nad instrukcją ❽ znajdującą się w funkcji szablonowej fun<double>. W obecnej sytuacji instrukcja ta będzie miała taką postać:

```
double wynikB = arg * pi2a<double>::wartosc;
```

Przypominam: to napisane tłustym drukiem wyrażenie oznacza tę potrzebną nam parametryzowaną zmienną zdefiniowaną gdzieś na zewnątrz funkcji szablonowej fun<double>. Sprytny kompilator automatycznie rozpoznał, że (w bieżącej sytuacji) powinna ona być typu double, i do jakiej (szablonowej) powinna należeć klasy (pi2a<double>), po czym dla nas tę szablonową klasę automatycznie wygenerował.

Podsumujmy zalety i wady tego sposobu 2a

W porównaniu ze sposobem 1, sposób 2a ma tę **zaletę**, że pozwala nawet wytworzyć zmienną (a nie tylko stałą). To znaczy, że do tej zmiennej moglibyśmy nawet wpisywać nową wartość, chociażby tak:

```
pi2a<T>::wartosc = 44.4;
```

W naszym programie tego nie zrobimy, ale zapamiętaj, że ten sposób 2a nadaje się nie tylko do wytwarzania dowolnych *stałych* parametryzowanych, ale nawet parametryzowanych *zmiennych*. To już coś!

A teraz **pierwsza wada**. Poprzednio narzekaliśmy, że w wyrażeniu ❼ oznaczającym liczbę pi musieliśmy po nazwie szablonu (pi1<T>) musieliśmy dopisać nawiasy pi1<T>(). Jak jest teraz? Jeszcze gorzej. Musimy teraz dopisać nazwę składnika, czyli ::wartosc. Oczywiście można by tę nazwę wymyślić krótszą, np. (::w), ale tak czy owak ten dopisek zaciemnia sprawę.

Druga wada tego sposobu realizacji to fakt, że oprócz definicji szablonu klasy pi2a ❷ musieliśmy też napisać (gdzieś poza ciałem szablonu) szablonową definicję jego składnika statycznego ❸. Więcej pisania.

Tę ostatnią wadę można usunąć stosując sposób 2b, opisany poniżej.

Sposób 2b – jak 2a, ale składnik statyczny od razu inicjalizowany jest „w klasie"

Wspomnianą przed chwilą wadę można czasem usunąć, bo (jak pamiętamy z C++11) w pewnych warunkach wolno nam opuścić pisanie definicji składnika statycznego poza ciałem klasy, jeśli ten składnik statyczny ma być stały (constexpr). U nas chodzi o stałą liczbę , więc stałość nas zadowala.

Oto jak sposób 2b zrealizować w naszym przykładowym programie:

❹ Definiujemy (jak poprzednio) dodatkowy szablon klasy o nazwie pi2b. Jest w nim tylko jeden składnik statyczny i to taki, że – uwaga – wolno go inicjalizować sposobem „w klasie". Skoro „w klasie", to znaczy, że nie musimy dodatkowo pisać definicji tego składnika poza ciałem szablonu klasy.

> *Przypominam, że bliższe szczegóły ten temat opisuje „Opus magnum C++11", paragraf 16.18.1 zatytułowany „Deklaracja składnika statycznego mająca inicjalizację 'w klasie'". Jednak nie radzę teraz zaprzątać sobie głowy tamtymi szczegółami.*

Mając już ten „prostszy” dodatkowy szablon klasy pi2b możemy zobaczyć jak się go używa.

Zatem jeśli kompilator wytwarza właśnie funkcję szablonową fun<double>, to dojdzie do instrukcji ❾. W niej zobaczy wyrażenie pi2b<T>::wartosc, więc uzna, że teraz potrzebna jest mu klasa szablonowa pi2b<double>. Jeśli takiej klasy w programie jeszcze nie ma, to ją teraz wytworzy z szablonu ❹. Wobec tego będzie możliwe wyrażenie pi2b<double>::wartosc dostarczające wartości liczby typu double.

Jak widać, sposób ten pozwolił nam uniknąć konieczności pisania definicji składnika statycznego poza klasą. Nadal jednak pozostała ta wada polegająca na tym, że w wyrażeniu korzystającym z tak zrealizowanej liczby pi musimy stawiać ten dopisek ::wartosc.

Innymi słowy, w starych sposobach 1, 2a, 2b występują dopiski, których chcielibyśmy unikać. To z powodu tych kłopotliwych dopisków wprowadzono do C++14 szablony zmiennych.

Sposób 3, najlepszy: szablon zmiennej (C++14)

C++14 Ponieważ o tym rozmawialiśmy już na początku, teraz tylko demonstracja.

❺ To jest szablon zmiennej, który jest prostym rozwiązaniem naszych kłopotów. Szablon nazywa się pi.

✧ Jeśli chcemy skorzystać ze zmiennej szablonowej ze zwykłej (nieszablonowej) części programu, to piszemy odpowiednią nazwę na przykład pi<double>, pi<float> lub pi<int>.

✧ Jeśli chcemy z niej skorzystać z wnętrza innego szablonu o parametrze T, to napiszemy po prostu pi<T>. To właśnie widzimy w instrukcji ❿.
Kompilator, wiedząc co w danej specjalizacji funkcji szablonowej fun oznacza symbol T, zamieni wówczas zapis pi<T> na właściwy w danej specjalizacji (czyli np. pi<double>).

❦

W naszym programie zobaczyliśmy cztery możliwe realizacje problemu zmiennej szablonowej. Żeby wszystkie mogły zaistnieć w tym samym programie, musiały mieć różne nazwy. Niestety te dłuższe nazwy w rodzaju pi2b mogły trochę zaciemnić sprawę. Aby więc teraz uczciwie porównać zapis wyrażeń w przypadku tych wszystkich sposobów, zobaczmy je teraz zebrane w tabelce, gdzie za każdym razem nazywają się tak samo: pi.

Sposób	Zmienna szablonowa zrealizowana jako	Użycie jej w szablonie
1	szablon funkcji	pi<T>()
2a	składnik statyczny w szablonu klasy	pi<T>::składnik
2b	składnik statyczny w szabl. klasy (dla uproszczenia inicjalizowany w-klasie)	pi<T>::składnik
3	**nowoczesny (C++14) szablon zmiennej**	pi<T>

Zapamiętaj:

W standardzie C++11, chcąc mieć w programie zmienną parametryzowaną, musieliśmy się zgodzić na te dziwne zapisy:

- z dodatkowym nawiasem wywołania funkcji **()** (sposób 1)
- lub z operatorem zakresu i nazwą składnika statycznego **::składnik** (sposoby 2a, 2b).

Szablon zmiennej wprowadzono do C++14 właśnie dlatego, żebyśmy nie musieli dodawać żadnego tych dopisków. Aby zapis zmiennej szablonowej wyglądał po prostu tak: *nazwa_zmiennej<parametr>*

Pomyślałeś może: *–Ech, czy naprawdę było warto dla tak skromnej korzyści?* Zobacz jak prosta i oczywista jest definicja szablonu zmiennej. Tylko dwie linijki (❺). A co do korzyści? Porozmawiamy o tym w następnym paragrafie.

B.5.2 Ciekawe zastosowanie: sprawdzenie cech charakteru danego typu

Zobaczyliśmy zastosowanie nowego narzędzia (zwanego szablonem zmiennej) do tworzenia parametryzowanej liczby . Przykład ten jest bardzo często wykorzystywany do wyjaśniania, co to są szablony zmiennych. Wydaje mi się jednak, że to zastosowanie jest raczej śmieszne, i że nikt nie stosuje go na serio. Spróbuję pokazać Ci sytuację, w której szablon zmiennej przydaje się naprawdę.

Wiesz od dawna, że jeśli mamy jakiś szablon (np. szablon funkcji fun<T>) to nie znaczy, że możemy go użyć do specjalizacji na rzecz każdego możliwego typu T. Są typy, dla których ten szablon nie tylko nie będzie miał sensu, ale nawet zawarte w nim instrukcje wywołają błąd kompilacji. Za sens lub bezsens tego typu sytuacji odpowiada programista sięgający po dany szablon. Powinien on wiedzieć do jakich typów T dany szablon się nadaje, a do jakich nie.

Teraz porozmawiamy o tym, jak zrobić, żeby szablon sam potrafił sprawdzić, czy dany typ T się dla niego nadaje. Zdziwisz się pewnie: „*–To szablony nagle tak zmądrzały?*" Nie, nie zmądrzały. Po prostu C++14 dał programistom możliwość sporządzenia takiego narzędzia, za pomocą którego ich szablon może wykryć, czy dany parametr T jest tym „nadającym się". Do sporządzenia tego „wykrywacza" użyjemy szablonu zmiennej. Najpierw zrobimy to w wersji niedoskonałej.

W uproszczeniu zasada sprawdzania będzie taka:

- Zdefiniujemy szablon zmiennej zet<T> będącej typu bool. Przez domniemanie, dla wszystkich możliwych typów T ta zmienna będzie inicjalizowana wartością false.
- Następnie zrobimy ręcznie specjalizację użytkownika tego szablonu zmiennej. Będzie to specjalizacja dla konkretnie *wybranego typu*. Ta szablonowa zmienna nazywać się będzie zet<*wybrany_Typ*> i inicjalizujemy ją wartością true.

Oto realizacja tego pomysłu w krótkim programie.

```
#include <iostream>
#include <string>
using namespace std;
//*****************************************************************************************************************
```

```
template <typename T>                // Ogólny szablon                                          ❶
bool          zet = false;           // zmienna szablonowa dla każdego typu T ma wartość false
//*************************************************************************************************************
template < >                         // Specjalizacja powyższego szablonu oznacza,
bool          zet<string>  = true;   // że gdy T jest typem string – to "się nadaje"            ❷
//*************************************************************************************************************
template <typename T>
void fun_wariant_tekstowy(T arg)
{
    cout << arg[2] << endl;          // wydruk konkretnej litery tekstu (tej o indeksie 2)
}
//*************************************************************************************************************
template <typename T>
void fun_wariant_nie_tekstowy(T arg)
{
    T rez = 2 * arg;                 // zakładamy, że arg to liczba, więc można ją mnożyć
    cout << rez << endl;
}
//*************************************************************************************************************
template <typename T>
void funkcja_robocza(T argument)                                                             ❸
{
    if(zet<T> == true)                                                                       ❹
    {
        cout << "Typ T oznacza obecnie std::string" << endl;
        // cout << "Litera [2] tekstu = " << argument[2] << endl;        //    !!!            ❺
    } else {
        cout << "Typ T to NIE jest std::string " << endl;
        // auto rezultat = 2 * argument;                                //    !!!            ❻
    }
}
//*************************************************************************************************************
int main()
{
        std::string  tekst("Napis");

        cout << "Wywolanie funkcji_roboczej<string>\n";
        funkcja_robocza(tekst);                                                              ❼

        cout << "Wywolanie funkcji_roboczej<double>\n";
        funkcja_robocza(3.3);                                                                ❽
}
```

Na ekranie zobaczymy:

```
Wywolanie funkcji_roboczej<string>
    Typ T oznacza obecnie std::string
Wywolanie funkcji_roboczej<double>
    Typ T NIE jest typu std::string
```

Porozmawiajmy bliżej o tym dziwnym programie

❸ Załóżmy, że Twoim zadaniem jest napisanie szablonu funkcji o nazwie funkcja_robocza. Szablon funkcji, to narzędzie do wytwarzania funkcji szablonowych funkcja_robocza<T> dla wybranego typu T. Jednak nie każdy możliwy typ T nadaje się dla danego szablonu . Kto potrafi to ocenić? Najlepiej zna się na tym twórca tego szablonu – czyli Ty.

> Chodzi więc o to, abyś w swoim szablonie funkcja_robocza<T> potrafił sprawdzać czy dany typ T jest odpowiedni dla tego konkretnego szablonu, czy też się do niego nie nadaje.
>
> Jeśli dany typ jest odpowiedni, to możesz wykonywać zamierzoną operację przeznaczoną dla tego typu danych.
>
> Jeśli jednak dany typ się nie nadaje, to szablon ten powinien wykonać inną, jakąś „neutralną" operację. (Choćby nawet było to wypisanie na ekranie ostrzegającego tekstu).

Takie jest marzenie. Jak je zrealizować?

Potrzebujemy narzędzia informującego nas: nadaje się/nie nadaje się. Zrobimy je przy pomocy dodatkowego szablonu zmiennej (na który pozwala nam standard C++14).

❶ Oto szablon zmiennej o nazwie zet. Ma on jeden parametr formalny T oznaczający *typ*, zatem ten szablon zmiennej zet może być specjalizowany dla dowolnych typów, na przykład double, char, int itd. W takiej sytuacji kompilator wytworzy zmienne szablonowe o nazwach zet<double>, zet<char>, zet<int> itd.
Oto (zapisana w pseudokodzie) specjalizacja dla typu double:

```
bool   zet<double> = false;                //   dla parametru double
```

Jest to definicja zmiennej typu bool o nazwie zet<double>, która zostaje inicjalizowana wartością false.

Spójrz jeszcze raz na inicjalizator w szablonie ❶. Sam widzisz, że jaki by nie był parametr aktualny tego szablonu, to i tak wytworzona tu zmienna szablonowa typu bool zostanie „na dzień dobry" inicjalizowana wartością false.

A teraz skup się. Szablon – to była produkcja seryjna. Teraz będzie rękodzieło.

❷ Oto robimy „ręcznie" dla tego szablonu **specjalizację użytkownika** na okoliczność, gdy konkretnym wybranym typem T jest std::string. Ta nasza specjalizacja zmiennej zet różni się od wszystkich innych „szablonowych" tym, że jest inicjalizowana wartością true. Tu, w punkcie ❷, widzisz jak się definiuje taką specjalizację użytkownika. Odpowiada to w praktyce takiemu zapisowi (w pseudokodzie):

```
bool   zet<string> = true;          //   dla parametru string
```

Jak widać, jest to definicja zmiennej typu bool o nazwie zet<string>, która to zmienna jest inicjalizowana wartością true.

Podsumujmy. Do tej pory zaistniała:

1. Specjalizacja szablonu zet, która dla **dowolnego** parametru T wytwarza zmienną typu bool o nazwie zet<T>. Taka zmienna szablonowa zawsze będzie inicjalizowana wartością false.
2. Ręcznie zrobiona specjalizacja szablonu zet, na okoliczność, gdy parametrem T jest aktualnie typ string. Ta specjalizacja wytwarza zmienną typu bool o nazwie zet<string>. Dla odróżnienia zmiennej tej nadana zostaje wyjątkowo wartość: true.

Co to nam dało? Jeśli teraz, w obrębie funkcji szablonowej funkcja_robocza<T>, umieścisz instrukcję if sprawdzającą wartość wyrażenia zet<T> to taka instrukcja pozwoli Ci przekonać się (w trakcie pracy programu) czy obecnie wykonywana jest funkcja szablonowa funkcja_robocza<string>, czy też może jakaś inna funkcja wyprodukowana z tego szablonu funkcja_robocza< *każdy inny typ* >.

❹ Oto ta instrukcja sprawdzająca. Znajduje się oczywiście w ciele szablonu funkcji funkcja_robocza. Jak widać, jest to zwykła instrukcja if, która sprawdza wartość przechowywaną w szablonowej zmiennej zet<T>. Powtarzam: Wszystkie wersje tej szablonowej zmiennej zet<T> będą miały wartość false, oprócz sytuacji, gdy parametr T jest naprawdę typem string, bo ta wersja zmiennej szablonowej ma wartość true.

Brutalnie zapytam: A kiedyż to się tak szczęśliwie stanie, że funkcja_robocza będzie w wersji dla typu string? Co musi zaistnieć w naszym programie, żeby tak było?

Odpowiedź jest prosta: gdy w naszym programie wystąpi gdzieś wywołanie funkcji funkcja_robocza z argumentem będącym stringiem. Tak się u nas dzieje w ramach funkcji main.

❼ Oto takie wywołanie. Gdy kompilator je zobaczy, to z szablonu ❸ wytworzy specjalizację funkcja_robocza<string>.

W tej wersji sprawdzenie warunku ❹ okaże się prawdziwe, bo zet<string> ma rzeczywiście wartość true. W rezultacie na ekranie możesz zobaczyć tekst świadczący, że funkcja poprawnie rozpoznała, że to jej ulubiony typ.

❽ A oto przykład negatywny: wywołanie funkcji_roboczej z argumentem 3.3 czyli argumentem typu double. Na widok tego wywołania kompilator wytworzy z szablonu wersję funkcja_robocza<double>. Ta wersja też sprawdzi warunek ❹. Skoro zet<double> ma w tym przypadku wartość false, więc funkcja wypisze na ekranie tekst, mówiący że taki aktualny typ T (czyli double) jej nie odpowiada. Jak widać, to sprawdzanie typu działa!

No, niby sukces, ale zauważ, że oprócz wypisania na ekranie radosnych tekstów, nie zrobiliśmy żadnych operacji.

Spróbujmy ten szablon funkcja_robocza trochę wzbogacić. Na przykład tak, że jeśli to argument jest typu string (czyli pewnie zawiera tekst), to odczytamy jedną z liter tego tekstu (na przykład literę o indeksie [2]). Oto jak spróbujemy zdefiniować tę nowszą wersję szablonu

```
template <typename T>
void funkcja_robocza(T argument)                                    ❸
{
    if(zet<T> == true)                                              ❹
    {
        cout << "Typ T oznacza obecnie std::string" << endl;
        cout << "Litera [2] tekstu = " << argument[2] << endl;   //  !!!   ❺
    } else {
        cout << "Typ T to NIE jest std::string " << endl;
        auto rezultat = 2 * argument;                           //  !!!   ❻
    }
}
```

Niestety, teraz programu nie da się skompilować. Będą dwa błędy kompilacji. Jak do nich dochodzi? Pracując nad wywołaniem ❼ kompilator wytworzy funkcję szablonową funkcja_robocza<string> i zacznie ją kompilować. W trakcie tej pracy...

1. Kompilator powie, że dla funkcji szablonowej funkcja_robocza(T) zrealizowanej właśnie przez niego w wersji dla parametru T = std::string – widzi błąd w instrukcji ❻ używającej wyrażenia (2 * argument). Błąd polegają na tym, że nie wolno użyć operatora mnożenia dla pomnożenia argumentów (int i string).

Mówiąc po prostu, kompilator poucza nas, że stringów nie można pomnożyć.

Gdy kompilator będzie kompilował dalej, to w swej pracy nad programem dojdzie w funkcji main do instrukcji ❽, czyli wywołania: funkcja_robocza(3.3). Jest to wywołanie funkcji szablonowej w wersji, której jeszcze nie ma, czyli w wersji funkcja_robocza<double>. Kompilator więc ją wytworzy (według szablonu ❸), po czym zacznie ją kompilować. Wtedy natknie się na kolejny błąd.

2. Kompilator powie, że dla funkcji szablonowej funkcja_robocza(T) zrealizowanej w wersji dla parametru T = double, widzi (w ❺) błędne wyrażenie: argument[2]. Błąd polega na tym, że nie można używać operatora indeksowania tablicy wobec pojedynczego obiektu typu double (dla którego teraz specjalizowana została ta funkcja).

> *No jak to!* – zawołasz oburzony. *–Przecież to pierwsze wyrażenie ❺ postawiłem tej części instrukcji if, gdzie już wykryliśmy, że argument jest typu string, zatem wolno nam za pomocą operatora [ ] sięgnąć do konkretnej litery stringu.*
>
> *Natomiast drugie wyrażenie(❻) jest w tej części instrukcji if-else, gdzie wiemy, że na pewno nie mamy do czynienia z typem string. Skoro nie, to chodzi o obiekt typu double wobec tego wolno nam wartość tego typu mnożyć przez 2.*

Niestety, takie rozumowanie jest błędne. To nie jest tak, że pół tej funkcji zostało specjalizowane dla typu string, a drugie pół (else) dla pozostałych typów. Tak to nie działa, sorry... Całe ciało szablonu funkcji zostało raz w całości specjalizowane dla typu string, a drugi raz – w całości dla typu double.

Zatem *wszystkie* instrukcje ciała szablonu funkcji_roboczej muszą być tak napisane, że

- ❖ w przypadku wersji funkcja_robocza<string> będą składniowo poprawne w pracy z obiektami klasy string,

i równocześnie

- ❖ w przypadku wersji funkcja_robocza<double> będą składniowo poprawne w pracy z obiektami klasy double.

> *–Ależ to niemożliwe! Tak się nie da!*

Masz rację. Tak się nie da...

> *–No to po co robiliśmy tę całą szopkę z if(zet<T>)... ?*

To było poprawne. Wykryliśmy typ, dla którego powstała specjalizacja i oznajmiliśmy to wypisując odpowiedni tekst na ekranie. Mogliśmy nawet zrobić coś więcej, bo mogliśmy nawet wypisać na ekranie ten argument:

```
cout << argument ;
```

Jest to możliwe, bo ta instrukcja wypisująca ma tę samą składnię w obu przypadkach. Wygląda tak samo dla wypisania liczby typu double jak i dla obiektu typu string.

Natomiast jeśli jakieś wyrażenie możliwe tylko dla jednej z tych wersji, a błędne dla drugiej – to kompilator przy kompilacji tej drugiej wersji zgłosi błąd. Umarł w butach.

Wiem, co teraz pomyślałeś: *–Znam cię Jurku już na tyle, że doprowadziwszy mnie do skraju rozpaczy, wyjmiesz zaraz jakiegoś królika z kapelusza i pokażesz jak ten problem można mimo wszystko rozwiązać.*

B.5.3 Lubię, nie lubię...

Na czym polegał nasz problem? Na tym, że w funkcji szablonowej potrafiliśmy ocenić, z jakim parametrem mamy do czynienia (nadaje się, czy też się nie nadaje), ale ta ocena obywała się na podstawie *wartości* zapisanej w danej zmiennej szablonowej. Wartości,

czyli czegoś, co jest znane dopiero w trakcie wykonania programu. Tymczasem błędy składni zostały wykryte dużo wcześniej, bo jeszcze w czasie kompilacji. Trzeba więc wymyślić inny sposób.

Spójrzmy jeszcze raz na dotychczasowy szablon zmiennej zet i na stojącą zaraz za nim specjalizację (użytkownika) napisaną dla parametru typu string.

```
template <typename T>                  // Szablon zmiennej                    ❶
bool     zet = false;

template < >                           // Specjalizacja użytkownika...        ❷
bool     zet<string>  = true;          // ...na  okoliczność parametru <string>
```

Jak się tym „narzędziem testującym" posługiwaliśmy? Gdy chcieliśmy wiedzieć czy wyrażenie zet<T> oznacza w konkretnym przypadku: zet<string>, czy może oznacza zet<cokolwiek_innego>, to w funkcji szablonowej w trakcie pracy programu sprawdzaliśmy wartość tego wyrażenia instrukcją if(zet<T>)... Rezultat true informował nas, że w danej wersji funkcji szablonowej T jest aktualnie typem string, a rezultat false – że każdym innym typem.

A może to był błąd? A może dałoby się rozpoznawać po czymś innym? Zastanów się... Czym innym (niż wartością true/false) mogłaby się różnić zmienna szablonowa zet<string> od zmiennej szablonowej np. zet<double>? Spójrz na powyższe definicje ❶ i ❷.

W jednym z poprzednich paragrafów rozmawialiśmy o tym, że konkretne realizacje specjalizacji zmiennej szablonowej (zrobionej przez użytkownika) powinna łączyć wspólna nazwa. Zgoda. Natomiast **wcale nie muszą być tego samego typu**. Przecież nawet w jednym z poprzednich przykładów mieliśmy różne wersje liczby pi, będące różnego typu (double, float, ...).

Już jesteśmy bardzo blisko rozwiązania. Umówmy się, że wszystkie możliwe typy świata T podzielimy na dwie grupy. Te, które lubimy, i te, których nie lubimy. A jeszcze dokładniej: te, które mają jakąś pożądaną przez nas cechę, i te, które takiej cechy nie mają.

Umówmy się, że:

- lubiane typy T będziemy oznaczać tak, że ich zmienna szablonowa zet<T> będzie typu signed int,
- natomiast w te nielubiane oznaczymy tak, że dla nich zet<T> będzie typu unsigned int.

Zmartwiłeś się pewnie jak to praktycznie zrobić? Niezwykle prosto. Skoro tych nielubianych jest zwykle bardzo dużo, to niech je wyprodukuje szablon.

```
template <typename T>
unsigned int    czy_lubiany_typ;          // wszystkich typów T (a priori) nie lubimy
```

Natomiast tych kilka, które lubimy, wyprodukujemy ręcznie jako specjalizację użytkownika. Proponuję by naszymi „ulubionymi typami" były te, które niosą informację tekstową, czyli typ string i typ char * (C-string).

Oto dwie specjalizacje napisane dla takiej sytuacji.

```
template< >
signed int   czy_lubiany_typ<const char*> ;          // lubimy C-stringi
```

```
template< >
signed int  czy_lubiany_typ<std::string>;          // lubimy std::stringi
```

Podkreślam, że:

- lubiane typy T rozpoznawane będą w taki sposób, że odpowiadająca im zmienna szablonowa czy_lubiany_typ<T> jest typu signed int,
- w przypadku nielubianych – odpowiadająca im zmienna szablonowa czy_lubiany_typ<T> jest typu unsigned int.

No dobrze, ale jak to sprawdzić w szablonie funkcja_robocza? Przecież nie można postawić instrukcji if do oceny typu! Oczywiście, że nie, zresztą po co? Przecież niedawno się przekonaliśmy, że to nic nam nie da, poza ewentualnym komunikatem na ekranie.

Rozwiązanie jest dużo lepsze. Wyobraź sobie, że gdzieś w ciele szablonu funkcji funkcja_robocza umieścimy taką instrukcję:

```
template <typename T>
void funkcja_robocza(T arg)                // szablon funkcji          ❸
{
    // ...
    fun_dwoista(arg, czy_lubiany_typ<T> );                              ❹
}
```

Jak to wywołanie zrozumie kompilator? To zależy kiedy będzie używał tego szablonu.

Typy lubiane

Jeśli będzie go używał dlatego, że gdzieś w programie zobaczył wcześniej wywołanie:

```
string zdanie = "tekst";
funkcja_robocza(zdanie);
```

to znaczy, że jest w trakcie wytwarzania funkcji szablonowej funkcja_robocza<string>. (według szablonu ❸). Innymi słowy jest to sytuacja, gdy kompilator wytwarza funkcję szablonową dla „lubianego" typu string.

W ciele tego szablonu kompilator widzi wywołanie fun_dwoistej ❹. (Gdzieś oczywiście musiała być jej deklaracja, ale to teraz nie ważne). Jakie są argumenty wywołania tej funkcji?

- Typ pierwszego argumentu jest oczywisty. Skoro to typ T, a jesteśmy w funkcji funkcja_robocza<string>, to znaczy, że T oznacza aktualnie typ string.
- A typ drugiego argumentu? To typ następującego wyrażenia: czy_lubiany_typ<T>. Skoro już ustaliliśmy, że T oznacza aktualnie string to znaczy, że chodzi o typ wyrażenia czy_lubiany_typ<string>. Jest to zmienna szablonowa, którą użytkownik sobie specjalizował tak, że ma być ona typu signed int.

Ustaliwszy to, możemy napisać, że chodzi tu o wywołanie funkcji o następujących dwóch argumentach:

```
void funkcja robocza(string, signed int);
```

Takie było wywołanie w przypadku „lubianego" typu string.

Drugi lubiany typ

Mamy w naszym programie zdefiniowany jeszcze drugi ulubiony typ: char *. Jeśli gdzieś w programie kompilator zauważy wywołanie

```
funkcja_robocza("abc");                    // C-string   (typ char*)
```

to przystąpi do generowania funkcji szablonowej funkcja_robocza<char *>. W ciele szablonu tej funkcji jest wyrażenie:

```
fun_dwoista(arg, czy_lubiany_typ<T> );                    ❹
```

Skoro parametr T oznacza obecnie char *, kompilator słusznie uzna, że chodzi o wywołanie funkcji:

```
void funkcja robocza(char *,  signed int);
```

Typy inne, czyli „nielubiane”

A teraz dla odmiany zobaczmy jak będzie jeśli kompilator zobaczy wywołanie funkcji funkcja_robocza z jakimś argumentem „nielubianego” typu:

```
funkcja_robocza(55.78);                    // wywołanie dla typu double
```

Kompilator przystąpi do wytworzenia funkcji szablonowej funkcja_robocza<double>. W jej ciele natknie się na wywołanie:

```
fun_dwoista(55.78, czy_lubiany_typ<T>);
```

Ponieważ kompilator już wie, że aktualnie parametr T oznacza typ double, więc rozumie, że zmienna szablonowa czy_lubiany_typ<T> oznacza teraz

```
czy_lubiany_typ<double>.
```

Czy jest taka specjalizacja użytkownika szablonu czy_lubiany_typ? Nie, nie ma, zatem obowiązuje wersja szablonowa, a ta jak łatwo sprawdzić jest typu unsigned int.

Oznacza to, że teraz, w przypadku „nielubianego” typu double mamy wywołanie funkcji o takiej deklaracji:

```
fun_dwoista(double, unsigned int);
```

Zbierzmy te dwie przykładowe sytuacje w tabeli.

Wywołanie funkcji roboczej	dla typu T	który uznajemy jako	spowoduje wywołanie funkcji dwoistej o takiej deklaracji
string s("tekst"); f_robocza(s);	std::string	*lubiany*	fun_dwoista(string, signed int);
f_robocza(33.44);	double	*nielubiany*	fun_dwoista(double, unsigned int);

Podsumujmy co się udało uzyskać:

Niniejszym więc udało nam się wewnątrz funkcji_roboczej rozdzielić zadanie między różne wersje fun_dwoistej, i to tak sprytnie, że jedno i to samo wywołanie fun_dwoistej dopasowane zostanie do różnych wersji tej funkcji. Co prawda wersje te mają tę samą nazwę, ale to zwykłe przeładowanie nazwy. Są to dwie odrębne funkcje, bo mają inne typy argumentów.

- W jednej z nich (tej dla typów „lubianych”) możemy swobodnie posługiwać się wyrażeniami charakterystycznymi dla pracy z tekstem (np. odniesienie się do wybranej litery tekstu).
- W drugiej, jakimiś wyrażeniami, które są neutralne, czyli nadają się do wszystkich „nielubianych” typów.

Realizację tego pomysłu zobaczymy zaraz w działającym przykładzie, ale najpierw kilka usprawnień.

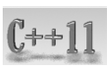
B.5.4 Usprawnienia i realizacja w programie

Pierwsze usprawnienie – typy *std::true_type* i *std::false_type*

Jak pamiętasz, rozróżnienie na sytuacje lubiane i nielubiane zrobiliśmy w ten sposób, że w sytuacji lubianej zmienna szablonowa była typu signed int, a w sytuacji nielubianej – unsigned int. Pytanie. Czy zamiast tego moglibyśmy wybrać typy char i typ long double? Tak! Byle były to tak różne typy, że kompilator potrafi rozsądzić do której to przeładowanej funkcji można dane wywołanie dopasować.

Zatem równie dobrze mógłbyś sobie sam zdefiniować jakieś takie proste typy, użyć je w szablonie zmiennej.

```
struct Typ_tak { };
struct Typ_nie { };
```



Żebyś tych prostych klas nie musiał definiować, twórcy biblioteki C++11 zrobili to za nas z myślą o szerokim zastosowaniu. Mamy więc dostępne dwa typy do oznaczania sytuacji tak/nie (prawda/fałsz, lubię/nie lubię)

- std::true_type
- std::false_type

Użyjemy ich w przykładzie w do naszych zmiennych szablonowych (zamiast tamtych signed int, unsigned int).

Te typy mają jeszcze jedną wygodną cechę. Ich twórcy wyposażyli je w operator konwersji na typ bool? Dzięki temu możemy obiekty tego typu stawiać w warunku instrukcji if.

Drugie usprawnienie
Funkcja dwoista niech będzie produkowana przez dwa odrębne szablony

Rozmawiając przedtem o sytuacjach, kiedy przychodzi nam pracować z typem lubianym i z typem nielubianym, doszliśmy do wniosku, że będą nam potrzebne dwie odrębne funkcje – osobna na sytuację lubianą, osobna na nielubianą.

Teraz zobaczmy, że tak łatwo nie będzie. Oto zestawienie potrzebnych nam funkcji w przypadku różnych typów

typ T	*potrzebna funkcja*
string	void fun_dwoista(string, **std::true_type**);
char*	void fun_dwoista(char*, **std::true_type**);
double	void fun_dwoista(double, std::false_type);
int	void fun_dwoista(int, std::false_type);
long long	void fun_dwoista(long long, std::false_type);

...i tak dalej

Potrzebujemy więc wiele funkcji, ale to nie jest problem, bo funkcje można podzielić na dwa rodzaje. Te lubiane mają się zachowywać określony sposób, a te nielubiane w inny. Sprawę załatwimy więc za pomocą dwóch szablonów funkcji. Obydwa będą nazywały się tak samo fun_dwoista, ale konfliktu nie będzie, bo będą produkować funkcje różniące się typem drugiego argumentu.

Zobaczymy to w programie.

```
#include <iostream>
#include <string>
using namespace std;
//*****************************************************************************************************************
template<typename T>
std::false_type   czy_lubiany_typ;                                                          ❶
//*****************************************************************************************************************
template< >
std::true_type   czy_lubiany_typ<const char*> ;                                             ❷
//*****************************************************************************************************************
template< >
std::true_type   czy_lubiany_typ<std::string>;                                              ❸
//*****************************************************************************************************************
// szablon funkcji dla typów nadających się (bo tekstowych)
template <typename T>
void fun_dwoista(T arg, std::true_type )                    // szablon A                    ❹
{
    cout <<  "\tF. dwoista wg szablonu A dla typow lubianych" << endl;
    std::string tekst (arg);
    cout << "\tTekst: >" << tekst << "<, \n\ta litera nr [2] to '" << arg[2] << "'"<< endl;
}
//*****************************************************************************************************************
// szablon funkcji dla typów nie nadających się (bo nie-tekstowych)
template <typename T>
void fun_dwoista(T arg, std::false_type)                    // szablon B                    ❺
{
    // nietekstowa wersja
    cout << "\tf. dwoista wg szablonu B dla typow NIE lubianych" << endl;
    T  tmp = arg;
    cout << "\tWydruk nie−tekstowego arg = " << tmp << endl;
}
//*****************************************************************************************************************
template <typename T>
void funkcja_robocza(T arg)                 // szablon funkcji                              ❻
{
     // najważniejsza instrukcja
    fun_dwoista(arg, czy_lubiany_typ<T>  );                                                 ❼

    // Poniżej  to tylko kosmetyka ——————————
    cout << "Czy ten typ T jest aprobowany? ";
    if(czy_lubiany_typ<T> )             // if(nazwa_zmiennej_szablonowej)                   ❽
    {
        cout << "TAK" << endl;
    } else {
        cout << "NIE" << endl;
    }
}
//*****************************************************************************************************************
int main()
{
    cout << "\nProba z liczba typu T −−−> int \n";
```

```
    funkcja_robocza(5);                                                    ❾

    cout << "\nProba z C-stringiem (typ const char*) \n";
    funkcja_robocza("abc");              // TAKI NUMER ???? --->          ❿

    cout << "\nProba z obj. kl. std::string \n";
    string s("Nowosci w C++14");
    funkcja_robocza(s) ;                                                   ⓫

    cout << "\nProba ze wskaznikiem int * \n";
    int roboczy = 4;
    int *wsk = &roboczy;
    funkcja_robocza(wsk);                                                  ⓬
}
```

Na ekranie zobaczymy:

```
Proba z liczba typu T ---> int
    f. dwoista wg szablonu B dla typow NIE lubianych                       9
    Wydruk nie-tekstowego arg = 5
Czy ten typ T jest aprobowany? NIE

Proba z C-stringiem (typ const char*)
    F. dwoista wg szablonu A dla typow lubianych
    Tekst: >abc<,
    a litera nr [2] to 'c'
Czy ten typ T jest aprobowany? TAK

Proba z obj. kl. std::string
    F. dwoista wg szablonu A dla typow lubianych
    Tekst: >Nowosci w C++14<,
    a litera nr [2] to 'w'
Czy ten typ T jest aprobowany? TAK

Proba ze wskaznikiem int *
    f. dwoista wg szablonu B dla typow NIE lubianych
    Wydruk nie-tekstowego arg = 0x7ffc1a25a67c
Czy ten typ T jest aprobowany? NIE
```

Omówienie ciekawszych miejsc programu

Na końcu programu widzimy funkcję main. Właściwie wszystko powyżej, to albo szablony, albo ich specjalizacje. Dlatego zacznijmy analizę od treści funkcji main.

❾ Oto wywołanie funkcji_roboczej z argumentem typu int. Gdy zobaczy to kompilator, powinien już znać albo deklarację takiej funkcji, albo definicję szablonu do wyprodukowania takiej funkcji.

❻ Oto on. Z tego szablonu kompilator powinien teraz wyprodukować specjalizację funkcja_robocza<int>, czyli innymi słowy funkcję szablonową void funkcja_robocza(int). Oto jakby ona wyglądała:

```
void funkcja_robocza(int arg)
{
    // najważniejsza instrukcja
    fun_dwoista(arg, czy_lubiany_typ<int> );                    ❼_int
```

```
    // Poniżej  to tylko kosmetyka ————————————
    cout << "\nCzy typ T jest aprobowany? ";
    if(czy_lubiany_typ<int> )  // if(nazwa_zmiennej_szablonowej)          ❽_int
    {
        cout << "TAK, ten typ aprobujemy." << endl;
    } else {
        cout << "NIE, tego typu nie lubimy." << endl;
    }
}
```

W tak wygenerowanej funkcji kompilator zobaczy wywołanie ❼ **_int.** Co to jest? Jest to wywołanie funkcji szablonowej o nazwie fun_dwoista. Takiej funkcji kompilator jeszcze nie spotkał, ale zna już nawet dwa szablony mogące funkcje o tej nazwie wyprodukować. To szablony ❹ i ❺. Który z nich się przyda w tym wypadku? O tym zadecydują argumenty wywołania. Odczytamy je z instrukcji ❼ **_int**.

a) Pierwszy argument (arg) jest typu int;

b) Jaki jest drugi argument wywołania dowiemy się gdy sprawdzimy jakiego typu jest wyrażenie czy_lubiany_typ<int>. Jest to zmienna szablonowa wyprodukowana z szablonu ❶. Jak widać, takie zmienne „na dzień dobry" są typu std::false_type.

Co prawda są potem specjalizacje tego szablonu ❶ na rzecz „ulubionych" typów const char oraz std::string – ale typ int do nich nie należy. Mamy teraz do czynienia z typem „nie-ulubionym", czyli zwykłym.*

W sumie instrukcja ❼ **_int** jest wywołaniem o argumentach typu:

```
fun_dwoista(int, std::false_type);
```

Funkcję szablonową o takiej nazwie i takich argumentach może wyprodukować w naszym programie tylko szablon ❺. Kompilator weźmie go na warsztat i powstanie mniej więcej taka definicja:

```
void fun_dwoista(int arg, std::false_type)          // specjalizacja szablonu B
{
    // nietekstowa wersja
    cout << "\tf. dwoista wg szablonu B dla typow NIE lubianych" << endl;
    int  tmp = arg;
    cout << "\tWydruk nie-tekstowego arg = " << tmp << endl;
}
```

Jak widać, nie ma tu żadnych wyrażeń charakterystycznych dla danych tekstowych. Jest bardzo ogólna operacja przypisania, a potem bardzo ogólna operacja wypisania na ekranie. Nie ma więc ryzyka błędów składni. Niniejszym więc na ekranie zostanie wypisany odpowiedni tekst. Sterowanie wróci w okolice instrukcji ❼ **_int**. Na tym w zasadzie praca funkcji_roboczej mogłaby się zakończyć. Oddelegowaliśmy przecież całą pracę do odpowiedniej wersji fun_dwoistej.

Chciałem jednak pokazać, że nawet w obecnej sytuacji można rozpoznać czy bieżący typ T jest tym „lubianym", czy „nielubianym".

❽ Spójrz na to miejsce w programie, albo jeszcze lepiej na miejsce ❽ **_int** w naszym tekście wyjaśniającym. Mamy tu instrukcję

```
if(czy_lubiany_typ<int> ) ...                                          ❽_int
```

Co jest wyrażeniem warunkowym w tej instrukcji if? Jest to wartość, która jest zapisana w zmiennej szablonowej czy_lubiany_typ<int>. A jaka tam jest wartość? Z szablonu ❶ możemy odczytać, że jest to coś typu std::false_type, ale nie widzimy tam żadnej inicjalizacji. Czy zatem w tej zmiennej szablonowej od urodzenia jest śmieć? Nie.

> *To, że nie widzimy inicjalizatora nie przesądza sprawy, przecież wystarczy, żeby inicjalizacji dokonał konstruktor domniemany klasy std::false_type.*

Musisz mi uwierzyć na słowo, że typ std::false_type jest inicjalizowany wartością false. W dodatku ta klasa std::false_type ma przeładowany operator bool(), dzięki czemu możemy obiekty tej klasy stawiać jako warunek w instrukcji if. (Tak jak tutaj, w ❽_int).

Podsumowując: w funkcji_roboczej też możemy rozpoznać czy mamy do czynienia z lubianym, czy nielubianym typem. Oczywiście nie możemy zrobić tutaj żadnych operacji swoistych dla takiego lub innego typu. Możemy tylko zrobić jakieś operacje wspólne dla obu sytuacji (lubianej, nielubianej). Na przykład operację wypisania komunikatu na ekranie, ale jeśli chodzi o rzeczy specyficzne dla tylko jednej z tych sytuacji, to w tym celu wymyśliliśmy dwa różne szablony fun_dwoista.

Zakończyliśmy omawianie wywołania ❾.

Sytuacja z typem „lubianym”

❿ Przejdźmy do wywołania funkcji_roboczej, w którym argumentem jest C-string. Chodzi o wywołanie funkcja_robocza("abc"). Kompilator teraz musi z szablonu ❻ wyprodukować specjalizację funkcja_robocza<const char*>, czyli innymi słowy funkcję szablonową funkcja_robocza(const char*); Oto jakby ona wyglądała:

```
void funkcja_robocza(const char* arg)
{
    // najważniejsza instrukcja
    fun_dwoista(arg, czy_lubiany_typ<const char*> );                    ❼_char*

    // Poniżej  to tylko kosmetyka ————————
    cout << "\nCzy ten typ T jest aprobowany? ";
    if(czy_lubiany_typ<const char*> ) // if(nazwa_zmiennej_szablonowej)   ❽_char*
    {
        cout << "TAK" << endl;
    } else {
        cout << "NIE" << endl;
    }
}
```

Po wytworzeniu tej funkcji, kompilator zacznie ją kompilować i natknie się na wywołanie ❼_char*. Jest to wywołanie (innej) funkcji szablonowej o takiej deklaracji

```
... fun_dwoista(cont char*, std::true_type);
```

Skąd wziąłem ten drugi typ argumentu?
Ano stąd, że zmienna szablonowa czy_lubiany_typ<const char*> ma w programie specjalizację użytkownika ❷ i tam jasno widać, że zmienna ta jest typu std::true_type.

Kompilator widzi więc tutaj (❼_char*) wywołanie funkcji, której wprawdzie jeszcze nie ma, ale może ją wyprodukować z szablonu A (❹). Tak też czyni. Powstaje funkcja

```
void fun_dwoista(const char* arg, std::true_type )          // wg szablonu A   ❹
{
      cout <<  "\tF. dwoista wg szablonu A dla typow lubianych" << endl;
      std::string tekst (arg);
      cout << "\tTekst: >" << tekst << "<, \n\ta litera nr [2] to '" << arg[2] << "'"<< endl;
}
```

Jak widać, w tej wersji szablonu (szablon A) jest poprawna składnia wyrażeń, które służą do wydobycia z tekstu litery o indeksie 2. Po wykonaniu tej części zadania, sterowanie wróci do funkcji_roboczej<const char*> i nastąpi wykonanie instrukcji

```
if(czy_lubiany_typ<const char*> )        // if(nazwa_zmiennej_szablonowej)  ❽ _char*
```

Nie potrzebuję dodawać, że ta instrukcja jest możliwa dlatego, że zmienna szablonowa czy_lubiany_typ<const char*> jest typu std::true_type, a konstruktor domniemany tej pożytecznej klasy inicjalizuje swoje obiekty wartością true. Dodatkowo, dla naszej wygody, obecny w tej klasie operator konwersji na typ bool sprawia, że możemy te zmienne szablonowe sprawdzać instrukcjami warunkowymi, jak tutaj, instrukcją if.

Inne typy

W naszym przykładzie widzimy kolejne wywołanie funkcji_roboczej.

⓫ Tym razem jest to wywołanie dla argumentu typu std::string. Kompilator wygeneruje więc funkcję szablonową funkcja_robocza<string>. W niej będzie wywołanie funkcji fun_dwoista(string, std::true_type). Drugi argument jest właśnie taki, bo definiując specjalizację czy_lubiany_typ<std::string> ❸ zdecydowaliśmy, że taka zmienna szablonowa jest typu std::true_type. W ten sposób poinformowaliśmy kompilator, że także typ std::string jest naszym ulubionym typem.

⓬ A to jest wywołanie dla typu, który jest wskaźnikiem do obiektów typu int, czyli wywołanie funkcji szablonowej funkcja_robocza<int*>. Skoro nie ma żadnej deklaracji, że użytkownik „lubi" ten typ int* – mówiąc serio: nie ma żadnej specjalizacji użytkownika zmiennej czy_lubiany_typ<int*> , zatem obowiązuje szablonowa wersja tej zmiennej, która „na dzień dobry" nie lubi... Nastąpi więc wywołanie funkcji:
fun_dwoista(int*, std::false_type);

Ćwiczenie

I Program z tego paragrafu przerób tak, by nie występowały w nim biblioteczne typy std::true_type i std::false_type. Ich rolę niech pełnią typy signed int oraz unsigned int.
Uwaga. Instrukcja if(czy_lubiany_typ<T>) w naszym programie jest możliwa dlatego, że obiekty bibliotecznego typu std::true_type i typu std::false_type są przez domniemanie inicjalizowane wartościami true i false. Zatem spraw, aby u Ciebie było podobnie, czyli żeby Twoje nowo wytworzone obiekty typu np. czy_lubiany_typ<T> były inicjalizowane wartością zero dla typów T „nie nadających się", a wartością 1 dla typów „nadających się".

B.6 Przeładowanie globalnych operatorów *new*, *new[ ]*, *delete*, *delete[ ]*

Kiedy rozmawialiśmy o przeładowaniu operatorów, podkreślałem że przeładowujemy je na użytek obiektów danej klasy. Dotyczy to również operatorów new, new[ ], delete,

delete[ ]. Są one jednak w pewnym sensie wyjątkowe, bo wolno nam także przeładować ich wersje globalne.

Co to oznacza? Oznacza to, że jeśli napiszemy swoją wersję tych operatorów, to ta nasza wersja będzie stosowana nawet do tworzenia/likwidowania zwykłych obiektów np. typu int, typu double – mówiąc ogólniej do typów wbudowanych. Tym samym przejmujemy kontrolę nad gospodarką pamięcią w całym naszym programie.

Po co się tym zajmować?

> Jedynym sensownym zastosowaniem, jakie mi przychodzi do głowy, to sporządzanie statystyki urodzin i zgonów obiektów naszego programu. Dzięki takiej statystyce możemy na przykład wyśledzić, gdzie przez pomyłkę rezerwujemy za dużo pamięci lub gdzie zapominamy o odwołaniu jakichś rezerwacji.

Niby to bardzo pożyteczna praca, ale zwykle nie musimy takiej całościowej statystyki robić samemu. Nasi starsi bracia programiści napisali do tych celów powszechnie dostępne programy i możemy z nich skorzystać.

> *W środowisku Linux możemy skorzystać z programu valgrind (jego opis łatwo znajdziesz w Internecie).*

Po co więc wyważać otwarte drzwi? Kto wie, może kiedyś chciałbyś napisać jeszcze lepszą wersję takiego programu diagnostycznego? Wtedy umiejętność przeładowania globalnych wersji operatorów new, new[ ], delete, delete[ ] bardzo Ci się przyda.

To jednak technika dla bardziej zaawansowanych. W zwykłych zastosowaniach nie ma potrzeby dokonywania takich przeładowań, więc przy pierwszym czytaniu, zdecydowanie radzę opuścić to zagadnienie.

„Obłędna! Stamtąd nikt nie wraca!"

Jeśli przeładowujesz globalny operator new, to niniejszym jego poprzednia, „oficjalna" wersja przestaje istnieć. Nie jest zasłonięta, tylko naprawdę przestaje istnieć. Skoro więc rezygnujesz z dotychczasowego new, to musisz znać jakiś inny sposób na powiedzenie Twojemu komputerowi, że właśnie potrzebujesz jakiegoś obszaru pamięci.

> Jednym takich sposobów jest wywołanie standardowej bibliotecznej funkcji malloc. (Zwrot tak uzyskanego obszaru pamięci robi się potem funkcją free).

Funkcja biblioteczna malloc to jest dobre rozwiązanie. Wywołamy ją w ciele naszego przeładowanego operatora new i dostaniemy obszar pamięci, o który nam chodzi.

Jest tu przy okazji mała, niemal kosmetyczna trudność. To nasze przeładowanie *globalnego* operatora new obowiązuje od samego początku pracy naszego programu. Spowoduje to pewien kłopot. Otóż znane nam strumienie cout oraz cin są obiektami swoich klas, a budowane są one na samym początku pracy naszego programu właśnie przy pomocy operatorów new. Gdybyśmy chcieli w ciele tego naszego przeładowanego *globalnego* operatora new umieścić instrukcję cout, wypisującą na ekranie jakiś tekst, to... będzie katastrofa. Dlaczego? Bo jeśli przeładujemy globalną wersję operatora new, wówczas obiekt cout będzie powstawał przy użyciu już tego nowego (naszego) globalnego operatora new. Zatem pierwszy raz nasz operator new zostanie wywołany do

pracy jeszcze zanim w naszym programie zaistnieje obiekt strumienia cout (bo jego budowa jeszcze się nie zakończyła). Co to oznacza w praktyce? To, iż w ciele tego operatora nie możemy umieścić instrukcji korzystających ze strumieni. Na przykład nie możemy tam umieścić takiego prostego wypisania tekstu:

```
cout << "To ja, globalny operator new" << endl;
```

Kompilator nie zabroni nam tej instrukcji, (bo nazwa cout jest najprawdopodobniej poprawnie zdeklarowana), ale w trakcie wykonania wstępnej fazy programu (przy tworzeniu przyszłego strumienia cout) nastąpi błąd.

> *To tak, jakby cout był wskaźnikiem pokazującym na razie byle gdzie, a my próbowalibyśmy korzystać z (na razie jeszcze nonsensownego) adresu. Pamiętasz paragraf „Strzał na oślep”?*

Czy zatem z wnętrza przeładowanego globalnego operatora *new* nie możemy dokonać żadnego wypisu na ekran?

Strumieniem cout rzeczywiście nie możemy, ale są przecież inne sposoby. Znają je zapewne programiści klasycznego C, gdzie wypisywanie na ekran odbywało się za pomocą wywoływania standardowych funkcji puts, printf (biblioteka stdio). Tymi starymi funkcjami wolno nam się posłużyć. Są to przecież funkcje, a nie obiekty.

```
puts("To ja globalny operator new");
int m = 7;
printf("Wartosc m = %d zostala wypisana", m);
```

To rozwiązuje problem. Zapamiętaj:

Jeśli wypisujemy coś w wnętrza globalnych operatorów *new*, *new[ ]*, to musimy użyć bibliotecznych funkcji *printf* lub *puts*. Nie można użyć strumienia *cout*.

Oczywiście w innych, „zwykłych” fragmentach naszego programu, wolno nam korzystać z cout według zwykłych zasad.

Jak przeładowanie operatorów *new*, *new[ ]*, *delete*, *delete[ ]* wygląda w praktyce?

Zaraz to zobaczymy w przykładowym programie. Najpierw jednak uwaga. W tym odcinku skupiamy się na nowinkach C++14, więc spieszę dodać, że ów standard C++14 wprowadził konieczność dodatkowego przeładowania operatora delete.

Takiego, w którym operator delete otrzymuje dodatkowy argument określającym rozmiar rezerwowanego obszaru pamięci.

```
void * operator new(std::size_t rozmiar);
void * operator new[ ](std::size_t rozmiar) ;
void operator delete  ( void* ptr )  noexcept;
void operator delete[ ]( void* ptr ) noexcept;
void operator delete  ( void* ptr, std::size_t rozmiar ) noexcept;      //   C++14
void operator delete[ ]( void* ptr, std::size_t rozmiar ) noexcept;     //   C++14
```

Pora na przykład, w którym zobaczymy przeładowanie powyższych operatorów w akcji.

```
#include <iostream>
#include <string>
using namespace std;
#include <stdio.h>                          // dla funkcji printf        ❶
// Przeładowanie globalnych operatorów
```

```
// Dla pojedynczych obiektów ----------------------------------------------
void * operator new(size_t rozmiar)                                          ❷
{
    printf("\tnew  pojedynczy obiekt, rozmiar %lu bajtow\n", (unsigned long) rozmiar);  ❸
    void *wsk = malloc(rozmiar);                                             ❹
    if(!wsk) throw std::bad_alloc();                                         ❺
    return wsk;                                                              ❻
}
//*************************************************************************************************************
void operator delete(void * wsk) noexcept                                    ❼
{
    printf("\tdelete pojedynczy obiekt\n");
    free(wsk);
}
//*************************************************************************************************************
// Nowość w C++14:  delete z dodatkowym argumentem określającym rozmiar
void operator delete (void * wsk, std::size_t rozmiar) noexcept              ❽
{
    printf("\tdelete (void * wsk, size_t rozmiar = %lu bajtow)\n", rozmiar);
    free(wsk);
}
//---DLA TABLIC ------------------------------------------------------
void * operator new[ ](std::size_t rozmiar)                                  ❾
{
    printf("\tnew[ ](size_t rozmiar = %lu bajtow)\n", rozmiar);
    void *wsk = malloc(rozmiar);
    if(!wsk) throw std::bad_alloc();
    return wsk;
}
//*************************************************************************************************************
void operator delete[ ](void * wsk)  noexcept  // (1 zwykłe)                ❿
{
   printf("\tdelete [ ] (void*);      bez arg. 'rozmiar'\n");
    free(wsk);
}
//*************************************************************************************************************
// Nowość w C++14:  delete[] z dodatkowym argumentem określającym rozmiar -----------
void operator delete [ ](void * wsk, unsigned long   rozmiar)    noexcept    ⓫
{
    printf("\tdelete [ ](void * wsk, size_t rozmiar = %lu bajtow)\n", rozmiar);
    free(wsk);
}
//*************************************************************************************************************
// Przykładowa klasa użytkownika
struct Tjonizator                                                            ⓬
{
    char znaki[50];

    Tjonizator()
    {
        cout << "\t\tKonstruktor Tjonizatora\n" ;
    }
    //-------------------------
    ~Tjonizator()                              // ma destruktor
```

```
    {
        cout << "\t\tdestruktor Tjonizatora \n" ;
    }
};
//*****************************************************************************************************************
int main()
{
    try {
        cout << "Proby z pojedynczymi obiektami----------\n";

        cout << "a) Pojedynczy obiekt (int ma rozmiar " << sizeof(int) << " bajtow)\n" ;

        int * wsk = new int;                                                    (13)
        delete wsk;                                                             (14)

        cout << "b) Obiekt klasy Tjonizator  (ma rozmiar " << sizeof(Tjonizator) << " bajtow)\n";
        Tjonizator * w = new Tjonizator;                                        (15)
        delete w;                                                               (16)

        cout << "\nProby z tablicami obiektów----------\n" ;

        cout << "c) Tablica int[100]\n";
        int *tablica = new int[10];                                             (17)
        delete [] tablica;                                                      (18)

        cout << "d) Tablica Tjonizator[4]\n";
        auto tab = new Tjonizator [3];                                          (19)
        delete []  tab;                                                         (20)
    }
    catch(bad_alloc)
    {
        cout << "Niepowodzenie w trakcie rezerwacji pamieci " << endl;
    }
}
```

Po wykonaniu programu na ekranie zobaczymy taki tekst:

```
Proby z pojedynczymi obiektami----------
a) Pojedynczy obiekt (int ma rozmiar 4 bajtow)
    new  pojedynczy obiekt, rozmiar 4 bajtow
    delete (void * wsk, size_t rozmiar = 4 bajtow)                              (14)
b) Obiekt klasy Tjonizator  (ma rozmiar 50 bajtow)
    new  pojedynczy obiekt, rozmiar 50 bajtow                                   (15)
        Konstruktor Tjonizatora
        destruktor Tjonizatora                                                  (16)
    delete (void * wsk, size_t rozmiar = 50 bajtow)

Proby z tablicami obiektów----------
c) Tablica int[100]
    new[ ](size_t rozmiar = 40 bajtow)
    delete [ ] (void*);    bez arg. 'rozmiar'                                   (18)
d) Tablica Tjonizator[4]
    new[ ](size_t rozmiar = 158 bajtow)                                         (19)
        Konstruktor Tjonizatora
        Konstruktor Tjonizatora
```

```
Konstruktor Tjonizatora
destruktor Tjonizatora
destruktor Tjonizatora
destruktor Tjonizatora
delete [ ](void * wsk, size_t rozmiar = 158 bajtow)
```

Porozmawiajmy jak przeładowuje się globalne *new, new[ ], delete, delete[ ]*

Rezerwacja (i likwidacja) pojedynczych obiektów.

❷ Oto przeładowanie globalnego operatora new. Kompilator sprawia, że operator ten zostanie wywołany z argumentem określającym ile bajtów ma mieć dana rezerwacja. W punkcie ⓭ widzimy użycie tego operatora do rezerwacji obiektu typu int. Kompilator wie, że w moim komputerze obiekty typu int są 4-bajtowe. Zatem ta właśnie liczba 4 zostanie wysłana operatorowi ❷.

❸ Pierwsza instrukcja w ciele naszego operatora wypisuje na ekranie informację. (To w celach szkoleniowych). Jak pamiętamy, tu w ciele globalnego operatora new, nie wolno nam posłużyć się strumieniem cout. Zamiast rozpaczać, używamy starożytnej funkcji printf z biblioteki stdio znanej jeszcze od czasów klasycznego języka C. Aby móc się nią posłużyć, na górze programu ❶ włączony został plik nagłówkowy z deklaracjami funkcji z tej biblioteki.

❹ Najważniejszą akcją w ciele operatora new jest wywołanie bibliotecznej funkcji malloc, która służy do rezerwacji pamięci o zadanym rozmiarze (podanym w bajtach).

- Jeśli rezerwacja się powiedzie, funkcja zwraca wskaźnik do początku zarezerwowanego właśnie obszaru pamięci. To właśnie postawimy poniżej przy słowie return jako rezultat pracy naszego operatora new.
- Gdyby z jakichś powodów rezerwacja się nie powiodła, to funkcja malloc odpowie wskaźnikiem do adresu zero. Taka sygnalizacja dawniej nam wystarczała, ale teraz jesteśmy nowocześni i polubiliśmy rzucanie wyjątków, zatem...

  Jeśli wolimy, żeby nasz operator new rzucał wyjątek, to wystarczy tu ❺ *sprawdzić, co jest we wskaźniku wsk. Jeśli wskazuje on na adres 0 (niepowodzenie), to wtedy instrukcją throw rzucamy wyjątek std::bad_alloc.*

⓭ Tym sposobem w funkcji main wytworzyliśmy (naszym globalnym operatorem new) obiekt typu int.

⓮ Gdy obiekt ten nie jest już nam potrzebny, możemy go skasować operatorem delete (także globalnym, przeładowanym przez nas).

❼ Oto jak go zdefiniowaliśmy. W pierwszej linijce jego definicji (która jest także jego deklaracją) widzimy słowo kluczowe noexcept, którym zapewniamy kompilator, że ta funkcja operatorowa nie rzuca żadnych wyjątków. (Jest taka zasada, że z operatorów delete nie powinno się rzucać wyjątków).
W ciele funkcji delete widzimy najpierw umieszczoną tam (w celach szkoleniowych) instrukcję wypisującą na ekranie. Bardziej z konsekwencji niż z obowiązku używam tu także bibliotecznej funkcji printf.

Najważniejszym zadaniem operatora delete jest zwolnienie rezerwacji. W tym celu wywołujemy tu funkcję biblioteczną free. Jako argument wysyłamy jej wskaźnik do początku obszaru pamięci, który chcemy zwolnić.

Standard C++14 zaleca, byśmy w przypadku przeładowania operatora delete przeładowali go w dwóch wersjach. Tej, co powyżej (❼), a także w wersji z dodatkowym argumentem określającym rozmiar zwalnianego obszaru.

❽ Oto definicja tej drugiej, siostrzanej wersji. Widzimy tutaj, że obok argumentu, będącego wskaźnikiem do początku zwalnianego obszaru, jest też argument przypominający jak duży był ten obszar (w bajtach).

Zauważ, że w funkcji main, gdzie korzystamy z tego operatora (❿), podaliśmy temu operatorowi delete tylko wskaźnik do początku zwalnianego obszaru. Kompilator w prezencie dodał jednak ten jeszcze jeden argumentu. Jest to ta sama wartość, którą przedtem przysłał do operatora new rezerwującego pamięć dla tego obiektu.

Ciała tego operatora ❽ nie muszę chyba objaśniać. Tutaj właściwie nie skorzystałem z tego dodatkowego argumentu rozmiar, ale gdybym robił tutaj statystykę zajętości pamięci, to ten rozmiar bardzo by się przydał. Działanie tego operatora możesz zobaczyć na ekranie 14.

Globalne *new* i *delete* wobec pojedynczego obiektu jakiejś klasy

Globalne operatory new i delete mogą posłużyć także do tworzenia obiektów klas użytkownika. Kompilator użyje ich pod warunkiem, że dana klasa nie ma własnych przeładowanych operatorów new i delete).

⓬ Oto definicja prostej klasy Tjonizator. Ma ona jeden składnik-daną, konstruktor i destruktor.

⓯ W funkcji main znajduje się instrukcja powołująca do istnienia obiekt klasy Tjonizator. Ponieważ klasa ta nie ma „swojego" operatora new, wobec tego kompilator sprawia, że uruchomiony zostaje globalny operator new (❷).

Jak widać na ekranie 15, kompilator sprawił także, że zaraz po operatorze new, zostaje też uruchomiony konstruktor klasy Tjonizator „meblujący" właśnie zarezerwowaną pamięć.

⓰ Oto instrukcja likwidująca właśnie wytworzony obiekt. Jak widać, kompilator zadba o to, by najpierw został uruchomiony destruktor tego obiektu, a dopiero po tym uruchamia naszą wersję globalnego operatora delete.

> *Zobaczyłeś tu użycie operatorów new i delete – w ich wersji globalnej. Może nie wierzysz, że to naprawdę przeładowanie wersji globalnej? Jeśli nie wierzysz, to możesz się łatwo przekonać dodając do występujących w main wszystkich wywołań operatorów new i delete dodatkowy operator zakresu, czyli pisząc ::new i ::delete. Działanie programu się nie zmieni. Nasze operatory są naprawdę globalne.*

Rezerwacja (i likwidacja) **tablic** obiektów typu fundamentalnego

Jak wiemy, operator new, służący do tworzenia pojedynczych obiektów, ma swego brata new[ ], służącego do tworzenia tablic obiektów. Podobnie jest w przypadku operatorów delete i delete[ ]. Również i te (globalne) wersje tych operatorów można przeładować. Zobaczymy teraz, jak zrobić i jak to działa w praktyce.

⓱ Oto znajdująca się w funkcji main instrukcja rezerwacji tablicy obiektów typu int. Kompilator sprawia, że wywołany zostaje nasz globalny operator new[ ] przeznaczony do rezerwacji tablic.

❾ Oto jego definicja. Najpierw „szkoleniowy" wypis na ekranie, przy użyciu bibliotecznej funkcji printf. Dalej wywołanie funkcji malloc rezerwującej pamięć. Gdyby rezerwacja nie powiodła się, funkcja ta zwróci wskaźnik do adresu zerowego. Ponieważ polubiliśmy rzucanie wyjątków, więc sprawdzamy ten wskaźnik i, jeśli jest rzeczywiście zerowy, rzucamy wyjątek typu std::bad_alloc. Jeśli rezerwacja się powiodła, to funkcja kończy pracę z rezultatem będącym wskaźnikiem do początku właśnie zarezerwowanej pamięci.

⓲ Gdy uznamy, że w naszym programie tak wytworzona tablica jest nam już niepotrzebna, wywołujemy globalny operator delete[ ].

> Standard C++14 wymaga, by przeładowując globalny operator delete[ ], zrobić to w dwóch wersjach: takiej „zwykłej" i takiej z dodatkowym argumentem określającym rozmiar zwalnianego obszaru pamięci.

⓾ Oto definicja tego „zwykłego", prostszego delete[ ]. Jak widać, w naszym programie jego ciało niewiele różni się od delete dla pojedynczych obiektów. Oczywiście gdybym rzeczywiście chciał skorzystać z możliwości prowadzenia statystyki zajętości pamięci, tobym umieścił tutaj instrukcje odnotowujące, że kasowana jest tablica.

⓫ Oto definicja drugiej wersji tego globalnego operatora delete[ ] – wymagana przez standard C++14. Jak widać, przyjmuje ona dodatkowy argument określający całościowy rozmiar likwidowanej tablicy. Zwróć uwagę, że to nie my wysyłamy ten argument temu operatorowi. Dodaje go nam w prezencie kompilator.
W ciele funkcji widzimy najważniejszą instrukcję: wywołanie funkcji bibliotecznej free, która oddaje systemowi operacyjnemu niepotrzebny nam już obszar pamięci.

Spójrz na ekran w okolice punktu ⓲. Zauważysz, że został wywołany operator delete[ ] ❿ (a nie ⓫). Dlaczego?

Standard pozwala, by kompilator wywołał tę prostszą wersję operatora delete w sytuacji:

- gdy jest to tablica obiektów typu wbudowanego (czyli niebędących obiektami jakiejś klasy)
- gdy są to obiektu typu, którego destrukcja jest banalna (bo po prostu ten typ nie ma destruktora),
- gdy kasowane obiekty są typu „jeszcze niedokończonego" (*ang. incomplete type*), czyli typu, który co prawda w danym miejscu w programie został już zadeklarowany (deklaracja zwiastująca), ale pełna definicja tego typu (klasy) jeszcze w programie się nie pojawiła.

Jak widać na ekranie, mój kompilator skwapliwie skorzystał z tej pierwszej możliwości, bo przecież typ int jest typem wbudowanym.

Rezerwacja (i likwidacja) tablic obiektów typu użytkownika

⓳ Dalej w funkcji main widzimy instrukcję, która ma za zadanie wytworzyć tablicę trzech obiektów klasy Tjonizator.
Ponieważ klasa nie ma swojego osobistego przeładowania operatora new[ ], zatem zostanie użyty globalny new[ ]. Jak wiemy, ten operator został w naszym programie przez nas przeładowany ❾, wobec tego teraz zostanie użyty. Na ekranie ⓳ widzimy ślad tego faktu w postaci wypisanego tekstu.

Dodatkowo następuje coś bardzo ważnego: po wykonaniu ciała tego operatora new[ ] (czyli samego aktu rezerwacji pamięci) kompilator uruchomi wielokrotnie konstruktor klasy Tjonizator, aby „umeblował" powstałe trzy elementy tablicy. Fakt uruchomienia tych konstruktorów potwierdzają teksty wypisane przez nie na ekranie.

⓴ Gdy tablica jest nam już niepotrzebna, wywołujemy operator delete[ ]. Na ekranie możesz zauważyć 20, że uruchomiona została ta wersja operatora, która ma dodatkowy argument określający rozmiar (czyli ⓫). To dlatego, że klasa Tjonizator ma destruktor, więc jej obiekty nie są likwidowane „banalnie". Gdybyśmy destruktor tej klasy usunęli, wówczas kompilator mógłby sobie uprościć sprawę i wywołać „prostszy" operator delete[ ] ❿.

❦

Zobaczyliśmy tu jak przeładowywać globalne wersje operatorów new, new[ ], delete i delete[ ]. Dowiedzieliśmy się także, że standard C++14 wymaga, aby jeśli już zdecydujemy się przeładować globalny operator delete, to trzeba to zrobić w dwóch wersjach: „prostej" i z dodatkowym argumentem określającym rozmiar zwalnianego obszaru. To samo dotyczy przeładowania globalnego operatora delete[ ].

Na zakończenie – powtórzę: prawie nigdy nie będzie Ci potrzebne przeładowanie tych operatorów. Sięgniesz po nie tylko w sytuacji, gdybyś chciał monitorować gospodarkę pamięcią w Twoim programie.

❦

B.7 Nowości C++14 w wyrażeniach lambda

O wyrażeniach lambda rozmawialiśmy w *Opus magnum* w rozdziale 30. Teraz zapoznamy się z dwoma ciekawymi cechami wprowadzonymi przez standard C++14. Są to:

1) uogólnione wyrażenia lambda,
2) wychwytywanie połączone z inicjalizacją.

Zanim zobaczymy te cechy w przykładowych programach, kilka zdań wyjaśnienia.

B.7.1 Przykład uogólnionego wyrażenia lambda

Mówiąc najprościej, w C++11 argumenty formalne wyrażenia lambda musiały być ściśle określonego typu. Tymczasem...

> Standard C++14 pozwala by typ argumentu formalnego naszego wyrażenia lambda był parametrem. Robimy to określając typ danego parametru słowem auto. Dzięki temu nasze wyrażenie lambda staje się jakby szablonem funkcji lambda o danej nazwie.

W poniższym programie zobaczysz jakie to proste.

```
#include <iostream>
using namespace std;
//**************************************************************************
int main()
{
    // wytworzenie zwykłego wyrażenia lambda C++11
    auto  czy_mniejsze_int = [ ] (int m, int k) { return m < k; };          ❶
```

```
    cout << boolalpha;                                                          ❷
    cout << "Czy (5 < 2) ? "  << czy_mniejsze_int(5, 2) << endl;                ❸
    cout << "Czy (5.1 < 5.9) ? "  << czy_mniejsze_int(5.1, 5.9) << endl; // niestety!  ❹

    // C++14
    // wytworzenie uogólnionego wyrażenia lambda
    auto  czy_mniejsze_uniwersalne = [ ] (auto m, auto k) { return m < k; };    ❺

    cout << "Czy (5 < 2) ? "  << czy_mniejsze_int(5, 2) << endl;                ❻
    cout << "Czy (5.1 < 5.9) ? "  << czy_mniejsze_int(5.1, 5.9) << endl;        ❼
    // to działa nawet na znakach
    cout << "Czy ('a' < 'b') ? "  << czy_mniejsze_uniwersalne('a', 'b') << endl; ❽
    cout << "Czy ('b' < 'a') ? "  << czy_mniejsze_uniwersalne('b', 'a') << endl;

    cout << "Wolno nawet nawet porownac wartosci roznych typow\n";
    cout << "Czy (3 < 3.14 ) ? "  << czy_mniejsze_uniwersalne(3, 3.14) << endl;  ❾

    cout << "Znak 'p' ma kod liczbowy: " << int('p') << endl;

    cout << "Czy ('p' < 111) ? "  << czy_mniejsze_uniwersalne('p', 111) << endl;  ❿
    cout << "Czy ('p' < 113.5) ? "  << czy_mniejsze_uniwersalne('p', 113.5) << endl;  ⓫
}
```

Na ekranie pojawi się taki tekst

```
Czy (5 < 2) ? false                    ❸
Czy (5.1 < 5.9) ? false                ❹
Czy (5 < 2) ? false                    ❻
Czy (5.1 < 5.9) ? false                ❼
Czy ('a' < 'b') ? true
Czy ('b' < 'a') ? false
Wolno nawet nawet porownac wartosci roznych typow
Czy (3 < 3.14 ) ? true                 ❾
Znak 'p' ma kod liczbowy: 112
Czy ('p' < 111) ? false                ❿
Czy ('p' < 113.5) ? true               ⓫
```

Najpierw przypomnienie

Jak pamiętamy z C++11, wyrażenie lambda najczęściej wpisujemy wprost w potrzebną nam instrukcję. Zostaje ono użyte przez tę funkcję, a potem staje się niepotrzebne. Gdybyśmy jednak chcieli skorzystać z danego wyrażenia lambda kilkakrotnie (w kilku innych instrukcjach), to możemy to zrobić nadając mu nazwę. W praktyce nadanie nazwy polega na tym, że definiujemy „obiekt lambda" o określonej nazwie i inicjalizujemy go wymyślonym przez siebie wyrażeniem lambda.

```
auto nazwa_obiektu_lambda = wyrażenie_lambda;
```

Abyśmy się nie musieli głowić jakiego typu jest nasze wyrażenie lambda stosujemy zastępcze słowo kluczowe auto. Nie będę rozwijał tego zagadnienia, bo o tej sprawie rozmawialiśmy w *Opusie* w §30.5.1. Koniec przypomnienia, teraz o nowościach.

❶ W naszym programie widzimy tak właśnie zdefiniowane wyrażenie lambda. Obiekt lambda, w którym będziemy je przechowywać, nazywamy długą, ale obrazową nazwą czy_mniejsze_int. Patrząc na ciało tego wyrażenia lambda, łatwo zauważysz, że otrzymuje ono dwa argumenty (dwie wartości typu int) i sprawdza czy pierwsza wartość jest mniejsza od drugiej. Jeśli tak, to zwraca wartość true, jeśli nie, to zwraca wartość false.

❸ Oto wywołanie tego wyrażenia lambda dla dwóch wartości typu int: 5 oraz 2. Na ekranie pojawia się odpowiedź false.

Odpowiedź jest „słowna" dzięki manipulatorowi boolalpha, którym powiedzieliśmy strumieniowi cout, że wolimy „słownie" ❷.

A co będzie, jeśli to samo wyrażenie lambda zastosujemy wobec dwóch wartości typu double?

❹ Oto taka sytuacja. Niestety nie zadziała to poprawnie. Skoro wartości 5.1 oraz 5.9 (czyli typu double) zostały wysłane wyrażeniu lambda obsługującemu wartości typu int, to nastąpiło ich obcięcie do typu int. Zatem wyrażenie lambda porównało czy (5 < 5). Wartością tego wyrażenia jest false, bo przecież 5 nie jest mniejsze od 5.

Wniosek? Nasze wyrażenie lambda nie nadaje się do porównywania wartości typu double. Nie jest uniwersalne.

Ten kłopot rozwiązuje możliwość wytworzenia uogólnionego wyrażenia lambda, na które pozwala nam standard C++14.

❺ Oto jego definicja. Zacytujmy:

```
auto  czy_mniejsze_uniwersalne = [] (auto m, auto k) { return m < k; };
```

Jak widać, zamiast określać typ argumentów m i k postawiliśmy tam słowa auto. Dzięki temu, stworzyliśmy jakby szablon wyrażeń lambda o danej nazwie. Takie uogólnione wyrażenie lambda można zastosować do wartości różnych innych typów. (Oczywiście takich, wobec których operator < ma sens). Oto kilka takich sytuacji.

❻ Użycie wyrażenia lambda czy_mniejsze_uniwersalne wobec argumentów typu (int, int). Działa, podobnie jak to poprzednie (❸).

❼ Użycie wobec argumentów typu (double, double). Działa poprawnie, czego poprzednia lambda (❹) nie potrafiła.

❽ Użycie wobec argumentów typu (char, char). Także działa poprawnie.

❾ Dwa argumenty naszego uogólnionego wyrażenia lambda wcale nie muszą być tego samego typu. Przecież każdy z nich ma swoje własne określenie auto. Oto więc użycie tego wyrażenia wobec pary argumentów (int, double). Jak widać, także i to działa poprawnie.

❿ A oto bardziej karkołomne zastosowanie. Porównanie argumentów (char, int), a potem ⓫ argumentów (char, double). (Patrząc na ekran i sprawdzając, że czy i jak to działa, pamiętaj że kod ASCII znaku 'p' to 112).

Pora na podsumowanie: Uogólnione wyrażenie lambda to narzędzie bardzo wygodne i bardzo proste w użyciu. Sprawia, że wyrażenie lambda, które właśnie tworzymy, może być bardziej uniwersalne, bo można je zastosować do wielu różnych typów argumentów.

> Ambitnym wyjaśniam, że cały mechanizm uogólnionego lambda polega na tym, że kompilator wyrażenie lambda zamienia nie na obiekt funkcyjny mający funkcję składową operator(), ale na obiekt funkcyjny mający *szablon funkcji składowej operator()*. W zależności jakie podajemy argumenty aktualne naszemu wyrażeniu lambda, kompilator produkuje taką lub inną przeładowaną wersję tego operatora().

Jak widać, cały spryt uogólnionego wyrażenia lambda realizowany jest za pomocą szablonu. Szablony zaś, jak to już kiedyś ustaliliśmy, to nie

- run-time polimorfizm,

  *innymi słowy **wielopostaciowość** zrealizowana już trakcie pracy programu (za pomocą funkcji wirtualnej),*

- lecz compile-time polimorfizm

  *inaczej: **wielopostaciowość** zrealizowana w trakcie kompilacji (za pomocą szablonu).*

Przyznam się, że nie lubię tych uogólnionych wyrażeń lambda nazywać polimorficznymi. Niepotrzebnie kojarzy się to z funkcjami wirtualnymi, a – co tu dużo mówić – ten mechanizm nie dorasta funkcjom wirtualnym do pięt.

B.7.2 Przykład wychwycenia połączonego z inicjalizacją

Jak wiadomo, wyrażenie lambda może wychwycić jakieś lokalne obiekty automatyczne dostępne w zakresie, w którym nastąpiło. Dzięki temu może z nich korzystać w swoim ciele.

C++14 daje nam dodatkową możliwość:

> Na liście wychwytywania możemy zdefiniować jakiś dodatkowy obiekt przydatny nam w ciele wyrażenia. W dodatku już tam, na liście wychwytywania możemy inicjalizować go jakimś wyrażeniem.

Mówiąc ściślej:

Jeśli na liście wychwytywania umieścimy nazwę z inicjalizacją:

nazwa = *wyrażenie_inicjalizacyjne*;

to tak, jakbyśmy zdefiniowali zmienną i inicjalizowali ją. Jej zakres, to zakres ciała wyrażenia lambda. Jej typ jest taki, jakby przed jej nazwą stało słowo auto, (czyli wynika on z typu wyrażenia inicjalizacyjnego).

Co ciekawe i bardzo wygodne – w wyrażeniu inicjalizującym tę zmienną, możemy wykorzystać nawet nazwy lokalnych (automatycznych) obiektów dostępnych w zakresie, w którym naszą lambdę tworzymy.

> Jeśli potrafisz sobie wyobrazić wyrażenie lambda jako obiekt funkcyjny, to przedstawiony tu mechanizm możesz rozumieć jako możliwość dodania do lambdy nowych danych składowych

Te i dalsze interesujące szczegóły zobaczymy w programie poniżej.

```
#include <iostream>
#include <memory>
using namespace std;
//*****************************************************************************************************************
int main()
{
    int h = 1;                                                                                          ❶
    int k = 12;
    double e = 2.71;

    // definicja wyrażenia lambda
    auto  impakt = [obj = 5,  h = h + 400,  zmienna = k * (k + 1),      &ref = e] ()                    ❷
```

```
{
    // k = 1;        błąd, bo k nie było wychwycone (służy jednie do inicjalizacji)   ❸

    // e = 1.0;      błąd (jak wyżej)                                                 ❹
    ref = ref + 10;                                                                   ❺
    // zmienna++;          modyfikacja dozwolona tylko jeśli lambda...
                           //                          ...ma przydomek mutable       ❻

    cout << "obj " << obj
         << ", h = " << h                                                             ❼
         << ", zmienna = " << zmienna
         << ", ref = " << ref
         << endl;
    return;
};

impakt();                  // wykonanie wyrażenia lambda                            ❽
cout << "Po wykonaniu lambdy impakt  h = " << h << endl;                            ❾
}
```

Tak wyglądał będzie ekran po wykonaniu tego programu

```
obj 5, h = 401, zmienna = 156, ref = 12.71
Po wykonaniu lambdy impakt  h = 1
```

Omówienie

❷ W naszym programie definiujemy wyrażenie lambda. Definiujemy je (dla uproszczenia) nie w wywołaniu jakiejś funkcji algorytmu, ale – że tak powiem – w postaci „wolno stojącej". Zostaje ono zapisane w obiekcie lambda o nazwie impakt. Fakt, że tak właśnie postąpiliśmy nie ma nic wspólnego z przedmiotem obecnego paragrafu. Po prostu dzięki temu, uniknąłem konieczności definiowania funkcji-algorytmu.

Jak widzisz, lista wychwytywania (czyli to, co jest zamknięte w kwadratowym nawiasie) jest dość rozbudowana, jest w niej kilka członów oddzielonych przecinkami.

```
[obj = 5,  h = h + 400,  zmienna = k * (k + 1),  &ref = e]
```

Jest ich aż tyle, bo chcę pokazać różne warianty nowego typu zapisu wychwytywania z inicjalizacją. Nie przerażaj się jednak, zwykle zastosujesz tylko jeden, no może dwa takie człony. Omówimy je po kolei.

człon: *obj = 5*

Taki zapis na liście wychwytywania odpowiada jakby zapisowi

```
auto obj = 5;          // pseudokod
```

co oznacza następujące polecenie dla kompilatora: Zdefiniuj w ramach tego wyrażenia lambda lokalną zmienną o nazwie obj i inicjalizuj ją wartością wyrażenia (5). Wyrażenie to jest (jak wiadomo) typu int, więc niech zmienna obj będzie właśnie takiego typu.

człon: *h = h +400*

Taki zapis mówi kompilatorowi, że chcielibyśmy, aby zdefiniował lokalny obiekt o nazwie h i inicjalizował go wyrażeniem (h + 400). Ale uwaga: to h stojące na prawo od znaku = (czyli w wyrażeniu inicjalizacyjnym) to zupełnie inne h niż to po lewej. Dotyczy ono jakiegoś h, które istnieje i jest dostępne w zakresie, w którym akurat naszą lambdę definiujemy. W tym przypadku będzie to obiekt ❶ o nazwie h, zdefiniowany w zakresie funkcji main.

Zapamiętaj to. Nawet jeśli mamy tę samą nazwę po jednej i drugiej stronie znaku = w wyrażeniu wychwytującym, np.:

```
h = h
```

owo h z lewej, oznacza lokalny obiekt, który definiujemy na użytek ciała wyrażenia lambda, zaś to h z prawej jest nazwą jakiegoś obiektu, który powinien być dostępny w zakresie, w którym naszą lambdę definiujemy.

Aby uniknąć pomyłek i nieporozumień, zwykle daję tym lokalnym obiektom inne nazwy, niż mają obiekty stojące po prawej. Przykład tego widzisz na następnej pozycji listy wychwytywania, czyli...

człon: *zmienna = k * (k + 1)*

Tutaj sprawa jest jasna i czytelna, więc nie trzeba tego objaśniać. Dodam więc tylko, że obiekt o nazwie k został tu tylko użyty w wyrażeniu inicjalizującym, ale to wcale nie znaczy, że został wychwycony. Zresztą zobacz, próba użycia go w ciele wyrażenia lambda (❸) wywoła błąd kompilatora. (Dlatego tę instrukcję musiałem opatrzyć znakami komentarza).

człon: *&ref = e*

A to jest definicja referencji (przezwiska) o nazwie ref, które od razu dowiaduje się (dzięki inicjalizacji), że jest przezwiskiem obiektu e. Sam obiekt e nie jest wychwycony, więc użycie go w ciele (❹) byłoby błędem. Ponieważ jednak wytworzyliśmy właśnie referencję do tego obiektu, to można z niego korzystać za jej pomocą (❺).

❻ Przypominam, że wprawdzie wyrażenia lambda pozwalają nam wychwycić jakieś obiekty przez wartość (czyli przez kopię), ale kompilator nie pozwoli nam tych kopii modyfikować. Aby to było możliwe, powinniśmy naszą lambdę opatrzyć przydomkiem mutable (zob. *Opus* §30.3.5). Ponieważ akurat tego nie zrobiliśmy, musiałem instrukcję ❻ ująć w komentarz.

❼ Oto jest instrukcja, w której wypisujemy na ekranie wartości różnych obiektów dostępnych nam w ramach ciała naszego wyrażenia lambda. Na ekranie możesz zobaczyć na przykład wartość lokalnego obiektu h (401).

❾ Potem, po zakończeniu definicji wyrażenia lambda i po wykonaniu go (❽), wypisujemy wartość tego h, które było zdefiniowane w funkcji main. Jak widać ta wartość nie została zmieniona i nadal wynosi 1.

Podsumujmy. C++14 wzbogacił możliwości listy wychwytywania wyrażeń lambda. Możemy tam wytwarzać (i inicjalizować) nowe pomocnicze obiekty, z których chcemy skorzystać w ciele wyrażenia lambda.
Inicjalizacja tych pomocniczych obiektów może być dowolnym wyrażeniem – nawet korzystającym z obiektów dostępnych w zakresie, w którym naszą lambdę definiujemy. O pewnym szczególnym zastosowaniu takiej inicjalizacji porozmawiamy w następnym paragrafie.

B.7.3 Przykład wychwycenia na zasadzie przeniesienia (move)

Jak wiemy, lista wychwytywania pozwala wyrażeniu lambda uzyskać dostęp do obiektów przez wartość, czyli przez kopię (albo przez sporządzenie referencji do

danego obiektu). A co będzie, jeśli interesującego nas obiektu nie da się kopiować, wolno go tylko przenosić?

> *To znaczy, że informację zawartą w tym obiekcie można jedynie wyjąć z jednego obiektu i włożyć do drugiego. Nie może ona być w obu obiektach równocześnie.*

Wyrażenie lambda nie może takiego obiektu wychwycić przez wartość (kopię).

Przykładem takich niekopiowalnych obiektów są „sprytne wskaźniki" realizowane przez biblioteczną klasę std::unique_ptr. (*Opus* §27.7.1).

Jeśli w programie posługujemy się nimi, to jak sprawić, żeby nasze wyrażenie lambda było zdolne wychwycić taki wskaźnik? W C++11 takiego obiektu wychwycić się nie dało. C++14 już nam to umożliwia. O tym teraz porozmawiamy.

Sprawę rozwiązuje bardzo prosto poznana w poprzednim paragrafie *inicjalizacja na liście wychwytywania*. Zobaczmy to w przykładzie. Będą w nim dwa wyrażenia lambda. Jedno wychwyci sprytny wskaźnik do pojedynczego obiektu, a drugie – do tablicy obiektów.

```
#include <iostream>
#include <memory>
using namespace std;
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
struct Tklasa                                                                                                                    ❶
{
   int skladnik = 0;
   ~Tklasa() {
      cout << "Destruktor ~Tklasa" << endl;                                                                                       ❷
   }
};
//*******************************************************************************************************************************
int main()
{
    cout << "Definicja sprytnego wskaznika do pojedynczego obiektu\n";
    unique_ptr<Tklasa>    spryt { new Tklasa } ;                                                                                 ❸

    {  // lokalny zakres w celach szkoleniowych ----------                                                                       ❹

        auto lambda1 = [sw = std::move(spryt) ] ()                                                                             ❺
        {
            sw->skladnik = 4;
            cout << "Wykonuje sie lambda1" << endl;
        };                                                                                                                     ❻

        if(!spryt)                                                                                                             ❼
            cout << "A spryt  NIE ma juz prawa własności " << endl;

        cout << "Przed wykonaniem lambdy" << endl;
        lambda1();                      // 1. wykonanie lambdy                                                                 ❽
        lambda1();                      // 2. wykonanie lambdy

        cout << "--- Teraz nastapi koniec zakresu istnienia lambdy1\n";
    } // koniec lokalnego zakresu ------------------------                                                                     ❾
```

```
    cout << "--- Skonczyl sie zakres istnienia lambdy1\n";

    cout << "PROBY Z TABLICA" << endl;

    // sprytny wskaźnik do tablicy 5-elementowej (z rezerwacja tej tablicy)
    constexpr int rozmiar = 5;
    unique_ptr<Tklasa[]>   wsktab { new Tklasa[rozmiar] } ;                    ❿

    {   // drugi lokalny  zakres (w celach szkoleniowych) ==============      ⓫

        auto lamTab = [stab = std::move(wsktab),   rozmiar] ()                 ⓬
        {
            for(int i = 0 ; i < rozmiar ; ++i){
                stab[i].skladnik = i;
            }
            cout << "Pracuje lamTab\n";
        };                                                                     ⓭

        cout << "Przed wywolaniem lamTab\n";
        lamTab();                                                              ⓮
        lamTab();
        cout << "=== Zakonczyl sie obszar istnienia lamTab" << endl;
    } // koniec lokalnego zakresu ============================               ⓯
    cout << "=== Koniec funkcji main" << endl;
}
```

Na ekranie pojawi się tekst:

```
Definicja sprytnego wskaznika do pojedynczego obiektu
A spryt  NIE ma juz prawa własności                                            ❼
Przed wykonaniem lambdy
Wykonuje sie lambda1
Wykonuje sie lambda1
--- Teraz nastapi koniec zakresu istnienia lambdy1
Destruktor ~Tklasa                                                             ❾
--- Skonczyl sie zakres istnienia lambdy1
PROBY Z TABLICA
Przed wywolaniem lamTab
Pracuje lamTab
Pracuje lamTab
=== Zakonczyl sie obszar istnienia lamTab
Destruktor ~Tklasa                                                             ⓯
Destruktor ~Tklasa
Destruktor ~Tklasa
Destruktor ~Tklasa
Destruktor ~Tklasa
=== Koniec funkcji main
```

Omówienie

O bibliotecznej klasie „sprytnych wskaźników” unique_ptr rozmawialiśmy w *Opusie* w §27.7.1. Służy ona do tworzenia obiektów zachowujących się podobnie jak wskaźniki. W sprytnym wskaźniku możesz przechować adres jakiegoś obiektu, który wytworzyłeś w zapasie pamięci operatorem new. Sprytny wskaźnik ma tę dodatkową zdolność, że gdy kończy się czas jego życia, to – że tak powiem – „pociąga on za sobą do grobu” obiekt, którego adres przechowywał. Jak sprytny wskaźnik to robi? Po

prostu w jego destruktorze jest instrukcja delete kasująca obiekt, którym się do tej pory opiekował.

Jak wiemy, może być wiele wskaźników wskazujących na ten sam obiekt. W przypadku sprytnych wskaźników mogłoby to być groźne. Wyobraź sobie, że na jeden obiekt stworzony w zapasie pamięci operatorem new wskazuje kilka sprytnych wskaźników. Gdy jednemu z tych sprytnych wskaźników skończy się czas życia, jego destruktor zlikwiduje cenny obiekt w zapasie pamięci. Jeśli pozostałe wskaźniki nadal będą chciały z tym „cennym" pracować – wywoła to katastrofę.

Aby tej katastrofy uniknąć, sprytne wskaźniki unique_ptr stosują zasadę, że tylko jeden z nich może w danej chwili znać adres naszego obiektu z zapasu pamięci. Jeśli przypiszemy taki wskaźnik do jakiegoś innego

```
spryciarz_nowy = spryciarz_stary;
```

to odbędzie się to nie na zasadzie skopiowania cennego adresu, ale na zasadzie przeniesienia go. Innymi słowy spryciarz_stary przekaże cenny adres spryciarzowi_nowemu, a sam o tym adresie zapomni.

Niby to genialne, ale w przypadku gdybyśmy chcieli pracować z takim obiektem w ramach wyrażenia lambda, to jak zorganizować jego wychwycenie? Przecież nie można tego zrobić na zasadzie wychwycenia przez wartość (czyli przez kopię). Zatem jak taki sprytny wskaźnik może zostać wychwycony przez wyrażenie lambda?

Jak wychwycić: *sprytny wskaźnik do **pojedynczego obiektu***

❸ Oto definicja sprytnego wskaźnika o nazwie spryt. Jest on typu unique_ptr<Tklasa>, czyli może się opiekować adresem obiektu typu Tklasa. (To taka pomocnicza klasa zdefiniowana na górze programu ❶). W nawiasie klamrowym widzimy, że nasz sprytny wskaźnik jest od razu inicjalizowany. Czym? Adresem obiektu typu Tklasa, który wytwarzamy operatorem new w zapasie pamięci.

❹ i ❾ W celach szkoleniowych definiujemy sobie w programie pewien lokalny zakres. Oczywiście w Twoim programie tak nie musisz robić. Mnie jest on potrzebny, by pokazać Ci kiedy dokładnie nastąpi „pociągnięcie do grobu".

❺ Oto definicja prostego wyrażenia lambda. Jego ciało jest rozpisane na kilka linijek.

Dla uproszczenia naszej rozmowy definicja ta nie jest zapakowana do wywołania jakiejś funkcji-algorytmu, ale jest przypisana do obiektu lambda o nazwie lambda1. (Podobnie robiliśmy w poprzednich przykładach, aby nie zaprzątać sobie dodatkowo głowy algorytmami).

Zwróć uwagę na listę wychwytywania. Tu jest kwintesencja tego programu. Definiujemy tutaj jakiś obiekt o nazwie sw i przypisujemy do niego wartość wyrażenia move(spryt). Innymi słowy sprawiamy tutaj, że adres przechowywany w spryt zostaje przeniesiony do obiektu sw. (O funkcji move rozmawialiśmy w *Opusie* §22.10).

Pytanko: jakiego typu jest obiekt sw? Odpowiedź: takiego jakby tam stało słowo zastępcze auto.

```
auto sw = std::move(spryt);                    // czyli typu: unique_ptr<Tklasa>
```

> Tak sprawiliśmy, że od tej pory obiektem wytworzonym operatorem new (w instrukcji ❸) opiekuje się wyłącznie sprytny wskaźnik sw.

W ciele wyrażenia lambda1 stawiamy jakieś przykładowe instrukcje korzystające z obiektu wskazywanego przez sw, następnie stawiamy klamrę } kończącą definicję

tego ciała. Potem ❻ jest średnik, bo to przecież koniec długiej instrukcji, która rozpoczęła się w ❺.

Ta długa instrukcja:
1) zdefiniowała wyrażenie lambda i ...
2) zapamiętała je w obiekcie lambda o nazwie lambda1.

❼ Jesteśmy znowu w zakresie funkcji main. Dla ciekawości sprawdzamy teraz sprytny wskaźnik o nazwie spryt. Jeśli jest w nim teraz już nullptr, to znaczy, że naprawdę oddał swój cenny skarb sprytnemu wskaźnikowi sw. Na ekranie widzimy 7, że tak rzeczywiście się stało.

❽ Oto wykonanie naszego wyrażenia lambda. Robię to dwukrotnie, żebyś zobaczył, że sprytny wskaźnik sw istnieje cały czas, dopóki istnieje obiekt lambda1. Nie znika więc po wykonaniu funkcji lambda. Przecież ktoś mógłby wywołać lambda1 po raz kolejny.

❾ Oto nadchodzi koniec naszego „szkoleniowego" lokalnego zakresu, w którym zdefiniowaliśmy nasz obiekt lambda1. Tutaj więc obiekt lambda1 przestaje istnieć. Skoro tak, to przestanie również istnieć jego (lokalny) sprytny wskaźnik sw. W chwili swojej śmierci tenże biblioteczny sprytny wskaźnik sw wywołuje swój destruktor, w którym (uwierz mi na słowo) jest instrukcja delete kasująca nasz obiekt Tklasa z zapasu pamięci. To właśnie moment „pociągnięcia do grobu". Czy to działa? Działa, potwierdza to gadający destruktor ❷, którego wypis pojawia się na ekranie 9.

Jak wychwycić: *sprytny wskaźnik do **tablicy obiektów***

Ponieważ w zapasie pamięci najczęściej rezerwuje się nie tyle pojedyncze obiekty, co tablice takich obiektów, zobaczmy i taką sytuację.

❿ Oto definicja sprytnego wskaźnika mogącego pokazywać na tablicę obiektów typu Tklasa. Jest on typu unique_ptr<Tklasa[ ]>. Ten nawias kwadratowy sprawia, że sprytny wskaźnik dowiaduje się, że to, czym będzie się opiekować, jest tablicą, zatem w razie likwidacji ma posłużyć się operatorem delete [ ] (a nie prostym delete).

Definiowany tutaj sprytny wskaźnik (o nazwie wsktab) jest od razu inicjalizowany. Co robi ta inicjalizacja? Wytwarza w zapasie pamięci (operatorem new[ ]) pięcioelementową tablicę obiektów typu Tklasa i jej adres przekazuje sprytnemu wskaźnikowi. Od tej pory wsktab wie, którą tablicą ma się opiekować.

⓫ Znowu w celach wyłącznie ilustracyjnych definiujemy lokalny zakres. Oczywiście w swoim programie nie musisz tego robić.

⓬ Oto definicja drugiego wyrażenia lambda w naszym przykładzie. Jej ciało jest rozpisane na kilka linijek. Po nim następuje średnik ⓭ kończący tę długą instrukcję definicji ⓬. Jak widzisz, to wyrażenie lambda przypisane zostaje do obiektu lambda o nazwie lamTab.

Dla nas najważniejsza jest tutaj lista wychwytywania. Oto jej pierwszy człon:

```
stab = std::move(wsktab)          // czyli jakby: auto stab = std::move(wsktab)
```

Jest to definicja lokalnego (dla lambdy) obiektu o nazwie stab. Jakiego jest on typu? Takiego jak wyrażenie inicjalizujące go. Wyrażeniem inicjalizującym jest wywołanie bibliotecznej funkcji move. Sprawia ona, że cenny adres przechowywany w sprytnym wskaźniku wsktab zostanie stamtąd wyjęty i włożony do sprytnego wskaźnika stab. (Od tej pory to wskaźnik stab odpowiada za ewentualne „pociągnięcie do grobu" tablicy, natomiast wsktab staje się bezrobotny).

Aby móc w wyrażeniu lambda pracować z tą tablicą, powinniśmy znać jej rozmiar. Załatwia to drugi człon listy wychwytywania (⓬). Jak widać, tę wartość wychwytujemy po prostu przez wartość.

⓮ Oto dwukrotne wywołanie funkcji lambda lamTab. Robię to dwukrotnie, żebyś nie pomyślał, że to zakończenie wykonywania lamTab spowoduje likwidację tablicy. Nie! Przecież tablica może być potrzebna w sytuacji, gdybyśmy lambdę chcieli wywołać po raz trzeci.

Likwidacja tablicy nastąpi dopiero wtedy, gdy będzie ginęła sama lambda. A kiedy się to stanie? Już za chwilę. To właśnie w tym celu w programie utworzyłem lokalny zakres ⓫↔⓯.

⓯ Ponieważ lamTab jest zdefiniowana w tym zakresie, więc teraz zostaje zlikwidowana. Na ekranie możemy zobaczyć teksty potwierdzające, że sprytny wskaźnik (będący częścią lambdy) na chwilę przed swą śmiercią wywołał operator delete[ ] likwidujący tablicę. Dowiadujemy się o tym znowu dzięki gadającemu destruktorowi elementów tablicy. To on wypisał na ekranie stosowne wypisy 15.

Podsumujmy

W tym paragrafie dowiedzieliśmy się, że wprowadzenie w C++14 nowości jaką jest możliwość definiowania na liście wychwytywania nowych lokalnych obiektów (wraz z ich inicjalizacją) – otworzyło dodatkowe drzwi. Można teraz wychwytywać obiekty na zasadzie ich przenoszenia (a nie tylko na zasadzie kopiowania).

Zwykle wystarcza nam kopiowanie. Bywają jednak obiekty, których informacji nie można kopiować; można ją tylko przenosić. Takie są na przykład sprytne wskaźniki unique_ptr. Tutaj zobaczyliśmy, jak praktycznie zrealizować takie wychwytywanie metodą przenoszenia.

B.8 C++14 a funkcje *constexpr*

B.8.1 Zniesienie wielu ograniczeń w ciele funkcji *constexpr*

W skrócie mówiąc, funkcje constexpr to funkcje, które mogą zostać uruchomione jeszcze przez kompilator, po to by obliczyły (i zwróciły jako swój rezultat) jakąś stałą typu constexpr (czyli, żartobliwie mówiąc, stałą „odwieczną").

> *Te stałe „odwieczne" są kompilatorowi potrzebne na przykład przy definiowaniu wielkości tablic, czy też w etykietach* case*, czy jako wartości parametrów szablonów.*

O funkcjach constexpr rozmawialiśmy w *Opusie* wielokrotnie:

§6.21 – Funkcje constexpr

§6.21.1 – Wymogi, które musi spełniać funkcja constexpr (w standardzie C++11)

§16.22 – Funkcje składowe z przydomkiem constexpr

§21.13 – Konstruktory constexpr mogą wytwarzać obiekty constexpr.

Jak pamiętasz, w standardzie C++11 są ostre ograniczenia dotyczące tego, jakie instrukcje mogą się znajdować w ciałach funkcji constexpr. Najkrócej mówiąc: w funkcji constexpr może być tylko jedna instrukcja return, a przy niej możemy ewentualnie

postawić wyrażenie składające się z prostych operacji arytmetycznych lub wywołania innej funkcji constexpr. Nie można więc w ciele funkcji constexpr użyć ani instrukcji if, ani switch, ani żadnej pętli np. for, while. Nie można też zdefiniować tam jakichś pomocniczych zmiennych.

Standard C++14 znosi bardzo wiele z tych ograniczeń. Teraz można już używać instrukcji pętli, instrukcji switch, if, można też definiować swoje zmienne (byle typu zdolnego do tworzenia obiektów typu constexpr).
Poniżej zobaczymy zestawienie tego, czego nadal nie wolno, ale sam przyznasz, że to już nie są żadne dotkliwe ograniczenia naszej fantazji twórczej:

Ciało funkcji constexpr może zawierać dowolne instrukcje z wyjątkiem:

- deklaracji asm

  to deklaracja, którą można przekazać jakieś informacje asemblerowi; nie rozmawialiśmy o tym, bo to, czy ta deklaracja jest w ogóle możliwa i jakie informacje można przekazać asemblerowi, zależne jest od implementacji konkretnego kompilatora,

- instrukcji goto,

  (i tak zawsze odradzałem używanie tej instrukcji)

- bloku try

  (no bo kto to słyszał, żeby w czasie kompilacji rzucać wyjątki!)

- definicji zmiennych typu nie-literalnego,

  (czyli niezdolnego do tworzenia stałych constexpr)

- ani mających przydomek static lub thread_local,

  o thread_local nie rozmawialiśmy, ale jest to przydomek ze standardowej biblioteki do programowania wielowątkowego,

- ani takich zmiennych, które nie mają inicjalizacji,
- ani instrukcji modyfikującej zmienną, której czas życia rozpoczął się w ramach funkcji constexpr. Należą do tego też sytuacje wywołania jakiejś funkcji składowej nie-constexpr i równocześnie niestatycznej (czyli zwykłej).

Warto przypomnieć, że także w C++14 (podobnie jak w C++11) funkcja constexpr nie może być virtual, jej argumenty muszą być constexpr, a jej rezultat ma być typu literalnego (czyli zdolnego do posiadania stałych typu constexpr).

Zrozumiałe jest chyba, że skoro funkcja constexpr ma działać w trakcie kompilacji, to nie powinniśmy w niej umieszczać akcji możliwych tylko w czasie prawdziwego wykonania programu. Na przykład nie powinniśmy w niej próbować uruchamiać operatora new (w celu tworzenia obiektów w zapasie pamięci).

Zobaczymy teraz przykład, w którym są dwie funkcje typu constexpr. Będzie to nowa realizacja znanych nam z §6.21.2 funkcji obliczających silnię i wartość sinusa.

```
#include <iostream>
#include <memory>
#include <cmath>
using namespace std;
//*************************************************************************************************************
constexpr  double pi = 3.141592 ;
//*************************************************************************************************************
```

```
constexpr unsigned long long silnia(int ile)                                                    ❶
{
    unsigned long long s = 1;
    switch(ile)                          // dozwolona w C++14 instrukcja switch
    {
    case  0: case 1:
        return s;

    default:
        for(int n = 1 ; n <= ile ; ++n)      // dozwolona w C++14 pętla for
        {
            s = s * n;
        }
        return s;
    } // koniec switch
    return s;
}
//*************************************************************************************************************
constexpr  unsigned long long silnia_5 = silnia(5);                                             ❷
//*************************************************************************************************************
constexpr double stopnie_na_radiany(double stopni)                                              ❸
{
    return pi * stopni / 180.0;
}
//*************************************************************************************************************
constexpr  double sinus_stopni(double x, int wyrazow = 10)                                      ❹
{
    x = stopnie_na_radiany(x) ;                                                                 ❺

    double wynik { };                                                                           ❻

    for(int n = 0 ; n <= wyrazow ; ++n)                                                         ❼
    {
        int znak_wyrazu = ( (n%2) ? (−1) : (1) ) ;                                              ❽
        int wykladnik = (2*n +1);                                                               ❾

        long double potega_x = x;
        for(int i = 1 ; i < wykladnik ; i++)        // obliczenie potęgi  x                     ❿
        {
            potega_x *= x;
        }
        double wyraz_ciagu  = znak_wyrazu * potega_x / silnia(wykladnik);                       ⓫
        wynik += wyraz_ciagu;                                                                   ⓬
    }
    return wynik;                                                                               ⓭
}
//*************************************************************************************************************
constexpr double  sinus_30 = sinus_stopni(30);                                                  ⓮
constexpr double  sinus_45 = sinus_stopni(45);
constexpr double  sinus_60 = sinus_stopni(60);
constexpr double  sinus_90 = sinus_stopni(90);
constexpr double  sinus_120 = sinus_stopni(120);
constexpr double  sinus_180 = sinus_stopni(180);
```

```
constexpr  double tablica_sinusow[ ]            // co 2.5 stopnia                    ⓯
{
    sinus_stopni(0),
    sinus_stopni(2.5),
    sinus_stopni(5),
    sinus_stopni(7.5),
    sinus_stopni(10)
    // i tak dalej
};
//*****************************************************************************************************************************
int main()
{
    constexpr  unsigned long long silnia_8 = silnia(8);                             ⓰
    // Te silnie i sinusy powinny już być obliczone w trakcie kompilacji
    cout << "silnia_5 = " << silnia_5 << ", silnia_8 = " << silnia_8 << endl;       ⓱

    cout << "sinus_30 = " << sinus_30
         << "\tsinus_45 = " << sinus_45
         << "\tsinus_60 = " << sinus_60
         << "\nsinus_90 = " << sinus_90
         << "\tsinus_120 = " << sinus_120
         << "\tsinus_180 = " << sinus_180
         << endl;

    for(int elem = 0 ; elem < 5 ; ++elem)
    {
        cout << "tablica_sinusow[" << elem << "] = " << tablica_sinusow[elem] << endl;  ⓲
    }
    cout << "Oczywiscie funkcji constexpr wolno uzyc rowniez jako zwyklej\n";
    for(int kat = 0 ;  kat <= 180 ; kat += 45)
    {
        double wynik = sinus_stopni(kat);                                           ⓳
        cout << "kat " << kat << " stopni, \tsinus = " << wynik  << endl;;
    }
}
```

Na ekranie wypisany zostanie tekst

```
silnia_5 = 120, silnia_8 = 40320
sinus_30 = 0.5      sinus_45 = 0.707107     sinus_60 = 0.866025
sinus_90 = 1 sinus_120 = 0.866026   sinus_180 = 6.55001e−07
tablica_sinusow[0] = 0                                                              ⓲
tablica_sinusow[1] = 0.0436194
tablica_sinusow[2] = 0.0871557
tablica_sinusow[3] = 0.130526
tablica_sinusow[4] = 0.173648
Oczywiscie funkcji constexpr wolno uzyc rowniez jako zwyklej
kat 0 stopni,  sinus = 0                                                            ⓳
kat 45 stopni,      sinus = 0.707107
kat 90 stopni,      sinus = 1
kat 135 stopni,     sinus = 0.707107
kat 180 stopni,     sinus = 6.55001e−07
```

Komentarz do tego programu

❶ Oto definicja funkcji constexpr obliczającej silnię. Jak widzisz, w jej ciele jest niedopuszczalna do tej pory definicja zmiennej pomocniczej (o nazwie s). Teraz, w myśl standardu C++14, wolno nam taką zmienną zdefiniować, bo:

- ❖ jest ona typu literalnego, czyli zdolnego do tworzenia stałych dosłownych swojego typu. Typ long long oczywiście ma tę cechę. (Oto przykład stałej dosłownej typu long long: 45LL).
- ❖ jest ona inicjalizowana (u nas: wartością 1),
- ❖ nie ma przy niej żadnych dodatkowych przydomków static, thread_local wspomnianych powyżej.

W ciele naszej funkcji widzisz zwykłe użycie instrukcji switch i pętli for (niedopuszczalnych przecież w standardzie C++11).

Nie będę omawiał działania tej funkcji, bo przecież o obliczaniu silni rozmawialiśmy w §6.21.2. Tam też możesz sprawdzić, jak to robiliśmy za pomocą rekurencji, skoro tych wszystkich obecnych udogodnień nie wolno było jeszcze stosować. Pamiętaj jednak, że pokazuję teraz nową wersję tej funkcji nie po to, by powiedzieć, że jedna z nich jest lepsza/gorsza. Celem tego przykładu jest pokazanie, że teraz, w C++14, masz większą swobodę, bo wolno Ci więcej.

❷ Oto definicja globalnego obiektu constexpr o nazwie silnia_5. Jego wyrażenie inicjalizacyjne to wywołanie funkcji silnia. To wywołanie odbędzie się jeszcze w czasie kompilacji. Kompilator wywoła tę funkcję z argumentem 5, a rezultat wstawi do definiowanego tutaj obiektu.

❹ Oto definicja funkcji constexpr o nazwie sinus_stopni. Posłuży nam ona do obliczania sinusa. Przypomnę, że wartość sinusa danego kąta x (podanego w radianach) można obliczyć na podstawie słynnego wzoru Taylora. Jego rozwinięcie to suma takich wyrażeń:

$$\sin(x) \quad x \quad \frac{x^3}{3!} \quad \frac{x^5}{5!} \quad \frac{x^7}{7!} \quad \frac{x^9}{9!} \quad \ldots$$

W §6.21.2 funkcję tę zrealizowaliśmy tak:

```
constexpr double nasz_sinus(double x)
{
    return      x
              − (x*x*x) /silnia(3)
              + (x*x*x*x*x)/silnia(5)
              − (x*x*x*x*x*x*x) / silnia(7)
              + (x*x*x*x*x*x*x*x*x) / silnia(9);
}
```

Jak widać, w tej funkcji zdecydowaliśmy się na obliczanie sumy tylko pięciu pierwszych wyrazów tego ciągu. Każdy z tych członów sumowania musieliśmy napisać w tym wyrażeniu dosłownie. Nie można było sobie pomóc za pomocą pętli for, bo przecież C++11 nie pozwalał na pętle w ciele funkcji constexpr.

Porównajmy teraz, jak to zrobiliśmy korzystając z możliwości C++14

W linii ❹ widzisz, że pierwszym argumentem naszej funkcji sinus_stopni jest zadany kąt – to oczywiste. Ale jest jeszcze drugi argument. Za jego pomocą możemy

zdecydować ile pierwszych wyrazów ciągu ma uczestniczyć w obliczeniach. W ten prosty sposób zdecydujemy z jaką dokładnością chcemy dokonać obliczeń. Argument ten ma określoną wartość domniemaną = 10, co oznacza naprawdę dużą dokładność obliczeń.

❺ *To zupełny drobiazg. Wzór Taylora na sinus wymaga podania mu kąta w radianach, a nie w stopniach. My jednak wolimy podawać wartości kąta w stopniach. Zatem funkcja* sinus_stopni *wysyła przysyłaną jej wartość stopni do pomocniczej funkcji* stopnie_na_radiany ❷*, aby jej te stopnie przeliczyła na radiany. Jak widać, wolno nam z ciała funkcji* constexpr *wywołać inną funkcję* constexpr*. (Oczywiście Cię to nie dziwi, bo to dało się zrobić nawet w C++11).*

❻ A to jest definicja pomocniczej zmiennej na wynik obliczeń. Jak widać, C++14 pozwala nam (w ciele funkcji constexpr) zdefiniować tę pomocniczą zmienną, bo zmienna ta spełnia warunki, że:

a) jest typu literalnego,

b) jej typ nie ma przydomka static, (ani thread_local),

c) w definicji tej jest od razu inicjalizacja {} wartością domniemaną (dla typu double oznacza to zero).

❼ Oto instrukcja for, możliwa w ciele funkcji constexpr od czasu C++14. W każdym obiegu tej pętli obliczać będziemy wartość kolejnego wyrazu ciągu (szeregu) Taylora.

❽ Ponieważ znak poszczególnych wyrazów się zmienia, zatem definiujemy pomocniczą zmienną o nazwie znak_wyrazu, która przyjmuje wartość +1 dla wyrazów (n) parzystych, a −1 dla n nieparzystych.

❾ We wzorze na rozwinięcie szeregu Taylora widzimy, że w poszczególnych wyrazach występuje potęgowanie. Wykładnik tej potęgi dla danego wyrazu ciągu obliczymy według wzoru (2*n + 1), gdzie n jest numerem wyrazu ciągu (numerujemy od zera).

❿ A teraz samo potęgowanie. Robimy je „ręcznie”, za pomocą pętli for,

„ręcznie” – bo nie ma gwarancji, że biblioteczna funkcja potęgująca std::pow *ma przydomek* constexpr *wymagany w tej sytuacji.*

⓫ Mając już przygotowane wszystkie czynniki składające się na wartość bieżącego wyrazu ciągu dokonujemy obliczenia. Jak widać, w tym wyrażeniu występuje również wywołanie funkcji silnia.

⓬ Następnie obliczoną wartość bieżącego wyrazu dodajemy do dotychczasowej wartości sumy wyrazów poprzednich.

⓭ Po zakończeniu pętli obliczającej sumę zadanej ilości wyrazów szeregu Taylora zwracamy rezultat, będący wartością sinusa dla interesującego nas kąta.

Zobaczmy jak korzysta się z tych funkcji *constexpr*

⓮ To jest definicja obiektu globalnego sinus_30 o przydomku constexpr. Ma ona też inicjalizację. W wyrażeniu inicjalizacyjnym jest wywołanie funkcji sinus_stopni. Wywołanie to odbędzie się jeszcze w czasie kompilacji, zatem wartość funkcji ❹ zostanie obliczona jeszcze przez kompilator. Podobnie będzie z następnymi definicjami znajdującymi się bezpośrednio pod instrukcją ⓮.

⓯ A to – ciekawostka. Definicja tablicy_sinusuów. Jest ona typu double i ma przydomek constexpr. Ma ona od razu inicjalizację „klamrową”. W niej widać wywołania funkcji

sinus_stopni dla poszczególnych wartości kąta. W ten sposób możemy przygotować sobie, znane Ci ze szkoły, tablice trygonometryczne funkcji sinus dla kątów od zera do dowolnej wartości (ze skokiem 2.5 stopnia).

⓰ Oto definicja innego obiektu constexpr. Tym razem jest to znajdująca się w funkcji main definicja *lokalnego* obiektu o nazwie silnia_8. (Przedtem, w ❷, mieliśmy definicję obiektu *globalnego*). Ponieważ argumentem wywołania funkcji silnia jest stała dosłowna (która jest constexpr), więc i to wywołanie zostanie wykonane jeszcze w czasie kompilacji.

⓱ W tej i następnych linijkach drukujemy na ekranie wartości obliczonych obiektów constexpr.

⓲ Tutaj drukujemy także treść naszej tablicy funkcji trygonometrycznej tablica_sinusow.

Jak wiemy, funkcja constexpr ma licencję na pracę jeszcze w czasie kompilacji, ale przecież możemy ją wywołać także w czasie normalnego wykonywania programu. Tak się dzieje tutaj ⓳. Skąd wiem, że to wywołanie NIE odbędzie się w czasie kompilacji? A stąd, że argument wywołania jest zwykłą zmienną nie mającą przydomka constexpr.

Zobaczyliśmy tutaj jak wyglądają funkcje constexpr napisane według zaleceń standardu C++14. Jak widać, pisanie tych funkcji w C++14 jest dużo wygodniejsze, bo zdjęto kilka zakazów utrudniających nam programowanie. Co prawda funkcje constexpr wydłużają czas kompilacji (bo np. te wszystkie sinusy trzeba wtedy poobliczać), ale za to skraca się czas wykonywania programu.

> *Jeśli chcesz się przekonać ile zyskasz, stosując w danej sytuacji funkcję* constexpr*, to po prostu zdefiniuj ją i zmierz czas wykonania programu w jej obecności, a potem usuń z jej definicji przydomek* constexpr*.*

B.8.2 Funkcje składowe *constexpr* w C++14 nie są już automatycznie *const*

W klasie możemy zdefiniować funkcję składową opatrzoną przydomkiem constexpr. W C++11 taka funkcja składowa miała przez domniemanie przydomek const, (czyli nie miała prawa modyfikować danych składowych swojej klasy).

> C++14 zmienia tę zasadę, to znaczy funkcja składowa constexpr nie ma obowiązkowo przydomka const. (Jeśli go chcemy, to powinniśmy go umieścić tam sami).

Jakie znaczenie ma ta zmiana?

Jak pamiętasz, w przypadku przeładowania funkcji składowej o danej nazwie, ma znaczenie fakt, czy dana funkcja ma przydomek const czy nie. Zatem mogą równocześnie istnieć takie dwie funkcje składowe

```
struct M
{
    int fs()              { return 5; }      // To są "różne" funkcje składowe
    int fs() const        { return 200; }    // więc mogą istnieć obok siebie
};
```

Wyobraź sobie teraz, że tworzysz program posługując się kompilatorem C++11 i zdefiniowałeś na przykład taką klasę, w której pierwsza z tych funkcji jest dodatkowo typu constexpr

```
struct K
{
    constexpr   int  fs()      { return 0; }      ❶
                int  fs()      { return 55; }
};
```

Program ten kompiluje się poprawnie. Nie ma konfliktu między tymi dwoma funkcjami składowymi, bo w myśl reguł C++11 do funkcji składowej ❶ został (automatycznie) dodany przydomek const.

Jeśli jednak w przyszłości skompilujesz ten program kompilatorem C++14, to nie będzie tego automatycznego dodawania przydomka const do funkcji składowych constexpr. W tej sytuacji kompilator zgłosi w tym miejscu błąd, mówiąc że takie dwie przeładowane funkcje nie mogą istnieć w tym samym zakresie ważności (czyli w tej klasie K).

Jeśli wiec, pracujesz w C++11, a przewidujesz że Twój program w przyszłości może być kompilowany w C++14, to w deklaracjach funkcji składowych constexpr dopisuj „ręcznie" przydomek const.

B.9 Atrybuty

Atrybuty zostały wprowadzone jeszcze do standardu C++11. To coś w rodzaju dodatkowych informacji, które wysyłamy do kompilatora po to, żeby pomóc mu w jego trudnej pracy. Dzięki nim, może on polepszyć generowany przez niego program. Nie rozmawialiśmy o nich w *Opusie*, bo uznałem, że to zbytnie szczegóły. Ponieważ jednak za chwilę poznamy bardzo ciekawy i pożyteczny atrybut [[deprecated]] wprowadzony do C++14, zobaczmy najpierw dwa atrybuty, które istnieją od czasów C++11.

atrybut *[[noreturn]]*

Tym atrybutem możemy oznaczyć funkcję, o której wiemy, że sterowanie nigdy nie wróci z niej normalnym trybem. Na przykład dlatego, że funkcja ta kończy zawsze swoją pracę rzucając jakiś wyjątek. Nigdy nie kończy swej pracy normalnie, czyli instrukcją return.

```
[[ noreturn ]] void funkcja()
{
        //...
        throw "error";
}
```

Jeśli poinformujemy kompilator o takiej naturze danej funkcji, to może on uprościć proces jej kompilacji.

atrybut *[[carries_dependency]]*

Atrybut ten dotyczy zagadnień związanych z programowaniem wielowątkowym. O standardowych klasach bibliotecznych obsługujących programowanie wielowąt-

kowe nie rozmawialiśmy w tej książce, bo z założenia „*Opus*” poświęcony jest samemu językowi C++. Co prawda poznaliśmy tu kilka najbardziej podstawowych bibliotecznych klas (takich jak std::string czy std::vector), ale gdyby chcieć opisać także pozostałe klasy biblioteki, to wtedy ta książka musiałaby być dwa razy grubsza. Jeśli będziesz studiować opis klas biblioteki standardowej i doczytasz się do klas realizujących wielowątkowość. Wówczas w naturalny sposób spotkasz się z zagadnieniem „przenoszenia zależności” między poszczególnymi wątkami programu i wówczas znaczenie artybutu [[carries_dependency]] stanie się dla Ciebie jasne.

Tymczasem powitajmy z radością prosty i bardzo pożyteczny...

B.9.1 Nowy atrybut *[[deprecated]]* wprowadzony w C++14

Wyobraź sobie, że jesteś już programistą, któremu można już zaufać. Tworzysz nie tylko swoje programy, ale tworzysz także pożyteczne klasy, jakby bibliotekę klas opatrzoną Twoim nazwiskiem. Oczywiście jesteś ambitny, więc ciągle dążysz do doskonałości. Klasy, które wymyśliłeś rok temu, zostały z uznaniem przyjęte przez Twoich przyjaciół i posługują się oni nimi w swoich programach. Tymczasem Ty, patrząc na tę swoją pracę sprzed roku, widzisz że można by to i owo zrobić lepiej. Wprowadzasz więc te zmiany. Nie ma problemu, gdy zmiana polega na jakimś sprytniejszym sposobie realizacji ciała jakiejś funkcji, bo przecież taka zmiana nie wymaga od użytkowników naszej klasy żadnych zmian w ich programach. Gorzej jednak, gdy wymyślisz funkcję, która wymaga dodatkowych pięciu argumentów, albo wymyślisz klasę pracującą na zupełnie innej zasadzie. Co wtedy? Kolegów korzystających z Twoich klas powinieneś poinformować o konieczności zmian w ich programach i o tym, jak teraz „po nowemu” korzystać z nowej wersji biblioteki.

> Dobra praktyka wymaga, żeby nie stawiać dotychczasowych użytkowników pod ścianą, mówiąc: *albo zmienicie te fragmenty waszego programu, które korzystają z moich klas, albo od tej pory wasze programy nie będą się dały skompilować*. W okresie przejściowym powinieneś zapewnić i stary, i nowy sposób korzystania z Twojej biblioteki.

Oczywiście powinieneś też nauczyć kolegów jak z tego wszystkiego korzystać. W zasadzie nie ma problemu. Rezerwujesz pokój seminaryjny na odpowiednią godzinę, po czym wygłaszasz seminarium oznajmiając nową generację Twojej biblioteki. Jeśli środowisko użytkowników jest ograniczone do jednego zespołu, da się nad tym zapanować. Wyobraź sobie jednak, że pracujesz w międzynarodowej firmie. Z Twojej biblioteki korzystają nie tylko koledzy w Twoim mieście. Macie też oddział w Singapurze, Genewie, Rio de Janeiro, Bostonie. Także i tamci użytkownicy powinni być świadomi, że z tej najnowszej wersji Twojej biblioteki radzisz korzystać inaczej. Jak ich o tym poinformować? Nie ma mowy o tym, byś pojechał na światowe *tourné*, oznajmiające, że oto zmieniłeś zdanie i teraz Twojej biblioteki używa się inaczej (lepiej). Przesadzam?

To, że oprogramowanie ciągle się unowocześnia, jest jakby wpisane w pracę programisty. Trzeba przyjąć jako fakt i nauczyć się postępować w takich sytuacjach.

C++14 daje nam proste rozwiązanie tego programu. Jest to atrybut [[deprecated]] [1], czyli taka jakby dodatkowa informacja dla kompilatora, którą będzie on informował korzystających z naszej pracy innych programistów, że dany element jest w pewnym sensie przestarzały, i że radzimy go używać inaczej.

Atrybut [[deprecated]] możemy postawić przy deklaracji klasy, synonimu nazwy (typedef) zmiennej, niestatycznej (czyli zwykłej) danej składowej, funkcji, funkcji składowej, argumentu funkcji, typu wyliczeniowego, specjalizacji szablonu itp.

Oto dwie możliwe formy zapisu:

```
[[deprecated]]                              //  forma bez wyjaśnienia
[[deprecated("Tekst wyjasniajacy") ]]       //  forma z wyjaśnieniem
```

Ta druga forma pokazuje, że słowu kluczowemu deprecated może towarzyszyć argument będący tekstem (C-stringiem). Zwykle umieszcza się tu wyjaśnienie dlaczego dany element uznajemy za przestarzały i co ewentualnie sugerujemy robić „zamiast".

Co nam da umieszczenie takiego atrybutu?

Gdy w kompilowanym właśnie programie kompilator napotka instrukcję korzystającą z tak oznaczonej funkcji lub zmiennej (lub nazwy typu), to wypisze na ekranie tekst ostrzegający, że dany element jest uznawany za „przestarzały". Do tego ostrzeżenia doda też nasz *tekst wyjaśniający* co w tej sytuacji zalecamy zrobić.

Zauważ, że „przestarzały" nie oznacza: „błędny".

> Atrybutem tym zniechęcamy do używania danej funkcji czy klasy, ale jeśli ktoś z niej mimo wszystko skorzysta, to program powinien nadal pracować poprawnie.

Innymi słowy ten atrybut to tylko informacja, że coś wyszło z mody, no ale ciągle jeszcze jest dopuszczalne. Może jednak za jakiś czas, w którejś z następnych wersji naszej biblioteki, zostanie to już zlikwidowane, zatem w ten sposób już dziś radzimy rozważyć sugerowane nowocześniejsze rozwiązanie.

Zobaczymy teraz sposoby umieszczenia atrybutu [[deprecated]] w różnych sytuacjach.

B.9.2 Oznaczenie wybranej funkcji jako przestarzałej

Załóżmy, że w naszym programie jest stara (ale ciągle poprawna) funkcjaA, którą od niedawna zrealizowaliśmy jakoś inaczej, lepiej pod nazwą funkcjaB. Od tej pory, chcemy zniechęcać użytkowników (kolegów programistów) do używania tej starej funkcjiA. W tym celu w deklaracji funkcjiA stawiamy atrybut [[deprecated]].

```
[[deprecated("zamiast funkcjiA lepiej uzywaj funkcjiB") ]]
void funkcjaA() ;

// Powtórna deklaracja, tym razem bez atrybutu...
void funkcjaA() ; // Raz użyty (powyżej) atrybut - obowiązuje. Nie powtarza się go.

// definicje funkcji
void funkcjaA() { return; }
void funkcjaB() { return; }
```

1) ang. *deprecated* – przestarzały [czytaj: *„deprekejted"*]

```
//*************************************************************************************************************
int main()
{
    funkcjaA();
}
```

Kompilator odpowie na to takim ostrzeżeniem (cytuję w uproszczeniu)

```
main.cpp: In function int main():
main.cpp: warning: void funkcjaA() is deprecated: zamiast funkcjiA lepiej uzywaj funkcjiB
```

Zapamiętaj, że wystarczy, aby atrybut [[deprecated]] pojawił się w jednej deklaracji funkcji i od tej pory kompilator przyjmuje go do wiadomości. Jeśli za chwilę pojawi się deklaracja tej samej funkcji ale bez tego atrybutu – nie spowoduje to odwołania cechy deprecated.

B.9.3 Argument funkcji uznany za przestarzały

Jest taka możliwość, żeby wybrany argument funkcji oznaczyć jako przestarzały. Kiedy to może się przydać? Wielu ludzi głowi się nad tym bezowocnie. Przecież jeśli oznaczymy jeden z argumentów naszej funkcji atrybutem [[deprecated]] to ile razy *w ciele tej funkcji* odniesiemy się do tego argumentu, otrzymamy dobrotliwe ostrzeżenie kompilatora o tym, że (sami) uznaliśmy ten argument za przestarzały. No ale po co nam ta wiedza? Nie dowie się o tym żaden użytkownik wywołujący naszą funkcję, a przecież z myślą o nim stawia się atrybuty [[deprecated]].

```
// Przestarzały argument funkcji – kiedy to się może przydać?
double funkcja(     int m,
                    [[deprecated("Te wartosc domniemana zmienic w przyszlych wersjach")]]
                    double x = 5.5)
{
    return m + x;
}
//*************************************************************************************************************
int main()
{
    funkcja(14);
}
```

Kompilator wypisze takie ostrzeżenie

```
In function double funkcja(int, double):
warning: x is deprecated: Te wartosc domniemana zmienic w przyszlych wersjach
  return m + x;
             ^
```

B.9.4 Przestarzałe niestatyczne składniki klasy: funkcja składowa i dana składowa

Atrybutem [[deprecated]] można opatrzyć wybrany niestatyczny (czyli zwykły) składnik klasy. Możemy tak oznaczyć jakąś funkcję składową lub jakąś daną składową. Zobaczmy jak to się robi.

```
class K {
public:
    int czerwony;
```

```
    [[deprecated("uzywaj raczej skladnika 'czerwony'" )]]
    int zielony;

    //    funkcja składowa
    [[deprecated("zamiast tej funkcji uzywaj raczej xxx")]]
    double funsk(int arg)
    {
        return 3.14 * arg;
    }
};
//*****************************************************************************************************************
int main()
{
    K obiekt;
    obiekt.zielony = 222;
    obiekt.funsk(4);
}
```

Oto ostrzeżenia kompilatora (w uproszczeniu):

```
warning: K::zielony is deprecated: uzywaj raczej skladnika 'czerwony'
  obiekt.zielony = 222;
           ^~~~~
warning: double K::funsk(int) is deprecated: zamiast tej funkcji uzywaj raczej xxx
  obiekt.funsk(4);
```

B.9.5 Obiekt oznaczony jako przestarzały

Atrybut [[deprecated]] możemy postawić przy nazwie zmiennej (globalnej lub lokalnej), do której używania chcemy zniechęcać. Oczywiście może to być zmienna typu wbudowanego, ale może też być typu jakiejś klasy. Oto takie sytuacje:

```
[[deprecated("zamiast z tego obiektu globalnego korzystaj raczej z abc")]]
int glob;       // zmienna globalna

struct S
{
    int x;
};

[[deprecated("Globlany obiekt klasy S o nazwie terytorium jest przestarzaly, korzystaj raczej z ttt")]]
S  terytorium;
//*****************************************************************************************************************
int main()
{
    glob = 4;
    terytorium.x = 100;

    [[deprecated("zniechecamy do uzywania tego obiektu") ]]
    char znak;   // lokalna zmienna typu wbudowanego

    znak = '%';

    [[deprecated("Lokalny obiekt o nazwie lokal jest przestarzaly, korzystaj raczej z xyz")]]
```

```
    S lokal;        // zmienna lokalna

    lokal.x = 5.78;
}
```

Oto uproszczona postać wydruku z procesu kompilacji

```
warning: glob is deprecated: zamiast z tego obiektu globalnego korzystaj raczej z ...
  glob = 4;
  ^~~~
warning: terytorium is deprecated: Globlany obiekt klasy S o nazwie terytorium jest przestarzaly,
korzystaj raczej z ttt
  terytorium.x = 100;
  ^~~~~~~~~

warning: znak is deprecated: zniechecamy do uzywania tego obiektu
  znak = '%';
  ^~~~
warning: lokal is deprecated: Lokalny obiekt o nazwie lokal jest przestarzaly, korzystaj raczej z xyz
  lokal.x = 5.78;
  ^~~~~
```

B.9.6 Zbiorcza definicja kilku zmiennych (z ewentualną inicjalizacją)

Jeśli mamy jedną instrukcję, będącą definicją kilku zmiennych, to jak i gdzie umieścić atrybut [[deprecated]] jeśli ma on dotyczyć wybranej zmiennej? Oto fragment programu ilustrujący jak to się robi:

```
int a =11,
b [[deprecated("lepiej jej nie uzywac") ]],
c [[deprecated("ta zmienna w nastepnych wersjach programu bedzie zlikwidowana") ]] = 2;

// posłużenie się tymi zmiennymi
b = 33;
c = 88;
```

Kompilator powie:

```
In function int main():
warning: b is deprecated: lepiej jej nie uzywac
  b = 33;
  ^
warning: c is deprecated: ta zmienna w nastepnych wersjach programu bedzie zlikwidowana
  c = 88;
  ^
```

B.9.7 Typy, które uznajemy za przestarzałe

Bywa tak, że napisaliśmy zupełnie nową wersję jakiejś naszej klasy bibliotecznej. Co prawda, że jest już całkiem nowa wersja klasy, ale w okresie przejściowym powinniśmy pozwalać użytkownikom używać jeszcze tej starej. Ustawiamy więc przy tej starej klasie atrybut [[deprecated]], który będzie użytkowników informował, że lepiej aby powoli przystosowywali swoje programy do pracy z nową wersją klasy.

Jak to zrobić, zobaczysz poniżej. Zwróć uwagę, że w przypadku gdy za przestarzały uznajemy typ (a nie obiekt), to umiejscawiamy atrybut po słowie class lub po słowie enum, a przed nazwą typu.

```
class [[deprecated ("Nie tworz obiektow tej klasy, uzyj innej np. Tnowa")]] Tstara
{
public:
    int skladnik;
    //...
};

Tstara   karczma;   // definicja obiektu

enum  [[deprecated ("Nie tworz obiektow tego typu enum, uzyj innego np. Toperacje_enum")]]
Takcje_enum { stop, go, pause, forward };

Takcje_enum   magnetofon;
//*************************************************************************************************************
int main()
{
}
```

Kompilator wypisze ostrzeżenia:

```
warning: Tstara is deprecated: Nie tworz obiektow tej klasy, uzyj innej np. Tnowa
 Tstara   karczma;   // definicja obiektu
          ^~~~~~

warning: Takcje_enum is deprecated: Nie tworz obiektow tego typu enum, uzyj innego np.
Toperacje_enum
Takcje_enum   magnetofon;
              ^~~~~~~~
```

Dlaczego, w przypadku przestarzałych typów, atrybut stawiamy **po** słowie class (lub **po** słowie enum)? Przecież przedtem, przy przestarzałych obiektach lub funkcjach, ów atrybut stawialiśmy **przed** całą deklaracją (definicją) zmiennej lub funkcji. Mógłbyś pomyśleć, że byłoby prościej, gdyby zawsze było tak samo... Oto moja odpowiedź. Jest w tym pewien cel. Chodzi o to, że czasem (choć rzadko) definicja klasy jest od razu połączona z definicją pierwszego obiektu tej klasy.

```
class T
{
    /* ciało klasy */
} obiekt;
```

Można powiedzieć, że jest to definicja obiektu typu T o nazwie obiekt; a przy okazji wyszczególniamy jak konkretnie zbudowany jest ten typ T;

- Jeśli przed tą całą instrukcją postawimy atrybut [[deprecated]] to dotyczył będzie on definiowanego tutaj *obiektu*.

  ```
  [[ deprecated ]] class T
  { /* ciało klasy */ } obiekt;          //    przestarzały jest definiowany tutaj obiekt
  ```

- Jeśli jednak atrybut [[deprecated]] postawimy tuż przed nazwą typu (zaraz po słowie class) to kompilator uzna, że atrybut ten dotyczy *typu* o nazwie T, który niniejszym uznajemy za przestarzały.

  ```
  class [[ deprecated ]] T
  { /* ciało klasy */ } obiekt;          //    przestarzała jest klasa T
  ```

Ponieważ typem definiowanym przez użytkownika jest też typ wyliczeniowy enum, więc i w takim przypadku zasada stawiania atrybutu [[deprecated]] jest podobna.

```
[[ deprecated ]] enum Tnazwa_enum
{ /* lista wyliczeniowa */ } obiekt;          //   przestarzały jest definiowany tu obiekt

enum [[ deprecated ]] Tnazwa_enum
{ /* lista wyliczeniowa */ } obiekt;          //   przestarzały jest ten typ enum
```

B.9.8 Przestarzałe synonimy typów (definiowane za pomocą *typedef* i *using*)

Oczywiście nie ma sensu określać typów wbudowanych (np. int, double, long) jako przestarzałe. Jednak jeśli któremuś z typów (wbudowanych lub użytkownika) została kiedyś nadana dodatkowa nazwa (synonim) to *taki synonim* można określić jako przestarzały. To sens ma. Oto jak to robimy:

```
[[deprecated("Lepiej nie uzywac tego typu ")]]
typedef  unsigned int  kanaly_t ;

using  Channels [[deprecated("ten synonim wychodzi wlasnie z mody")]]  = unsigned int ;

// posłużenie się tymi synonimami
kanaly_t   k;
Channels   ch;
```

Oto ostrzeżenia kompilatora:

```
main.cpp: warning: kanaly_t is deprecated: Lepiej nie uzywac tego typu
 kanaly_t       k;
                ^

warning: using Channels = unsigned int is deprecated: ten synonim wychodzi wlasnie z mody
 Channels       ch;
                ^~
```

Ciekawostka: W standardzie C++14 nie znalazłem wzmianki o tym, że atrybut [[deprecated]] wolno postawić w deklaracji using, ale wydaje się to logiczną konsekwencją. Bardzo wierny standardowi kompilator GNU C++ pozwala na to. Jak widzisz powyżej, kompilator chciał abym postawił ten atrybut po nazwie synonimu, a przed znakiem =.

B.9.9 Oznaczanie atrybutem *[[deprecated]]* specjalizacji szablonu klasy

Jako przestarzałą można oznaczyć wybraną specjalizację szablonu. Oto jak to zrealizować **w przypadku szablonu klas**:

```
template <typename K>
class Tretorta
{
public:
    K  skladnik;
};
// Załóżmy, że specjalizację tego szablonu dla parametru string – uznajemy za przestarzałą
template <>
class  [[deprecated("przestarzala specjalizacja dla parametru string")]]   Tretorta<string>
{
     // ...
```

```
};
// Przykład użycia
Tretorta<string>   przydatny_obiekt;
```

Kompilator wypisze ostrzeżenie:

```
warning: Tretorta is deprecated: przestarzala specjalizacja dla parametru string
 Tretorta<string>   przydatny_obiekt;
                    ^~~~~~~~~~~~~
```

B.9.10 Oznaczanie atrybutem *[[deprecated]]* specjalizacji szablonu funkcji

Jeśli chcemy określić jako przestarzałą wybraną specjalizację **szablonu funkcji**, to robimy to w taki sposób:

```
template <typename K>
bool czy_mniejsze (K a, K b) { return (a < b) ? true : false; }

template <>    // specjalizacja szablonu funkcji
[[deprecated("Zamiast tej specjalizacji lepiej uzyc innej")]]
bool czy_mniejsze(double w, double v)  {return w < v; }
//*****************************************************************************************************************
int main()
{
    bool odpowiedz = czy_mniejsze(2.71, 3.14);
}
```

Kompilator ostrzeże nas tymi słowy:

```
In function int main():
warning: bool czy_mniejsze(K, K) [with K = double] is deprecated: Zamiast tej specjalizacji lepiej uzyc
innej
  bool odpowiedz = czy_mniejsze(2.71, 3.14);
                                          ^
```

B.10 C++17 ante portas

W tym rozdziale poznałeś nowe aspekty języka C++ wprowadzone przez standard C++14. Czasem były to drobne usprawnienia (jak na przykład separatory cyfr, zapis binarny stałych dosłownych), a czasem coś, co na pierwszy rzut oka wygląda na drobnostkę (jak szablony zmiennych), a w istocie otwiera swoim działaniem nowe, ciekawe możliwości.

Wiedzę na temat nowego standardu C++14 przyswoiłeś sobie chyba na tyle, że chyba zgodzisz się ze mną, iż przewrotu nie było.

> Istota języka C++, którą przedstawia *„Opus magnum C++11”*, wystarczy Ci do napisania 99.5% potrzebnych Ci programów. Przecież C++14 nie unieważnia C++11. Leciutko go tylko rozwija. Leciutko.

Tymczasem świat cały czas idzie naprzód i standard C++17 stoi już *ante portas*. Nie daj się zwariować! Nie wpadnij w pułapkę i nie myśl, że nie będziesz się uczył C++11 lub C++14, skoro właśnie opublikowano standard C++17. To tak jakby ktoś powiedział:

–Nie będę się uczył angielskiego z dotychczasowego podręcznika, bo na londyńskim West Endzie pojawiły się w ciągu ostatniego roku nowe gwarowe zwroty, których mój bieżący podręcznik angielskiego nie uwzględnia.

No, prawda, nie uwzględnia, ale... Najpierw zadaj sobie pytanie czy w ogóle umiesz i rozumiesz angielszczyznę, i czy masz w niej cokolwiek do powiedzenia.

Tak jest w każdej dziedzinie, również w bliskiej mi fizyce. Ciągle publikuje się nowe artykuły w międzynarodowch periodykach fizycznych, ale to nie znaczy, że studenci mają przestać się uczyć z dotychczasowych podręczników R. Feynmana. To on rozniecił wyobraźnię tysięcy młodych studentów fizyki i to dzięki niemu corocznie wielu ludzi wkracza w niezywkły, fascynujący świat tajemnic przyrody. Gdyby jeden ze studentów powiedział, że "nie będzie czytał Feynmana, bo w *Journal of Physics* są nowsze arytkuły o fizyce", to słuchający go inni studenci mogliby się tylko uśmiechnąć: *„Znaj proporcjum, mocium panie!”*

Oczywiście wcześniej czy później trzeba się będzie przynajmniej pobieżnie zapoznać z tym, co zaleca, a co odradza nowy standard C++17. W tym sensie to Cię oczywiście nie minie. Jeśli jesteś niecierpliwy, po prostu zajrzyj do dokumentacji standardu C++17. Standard ten możesz sobie sprowadzić przez Internet (za skromną opłatą).

Gdybyś jednak wolał to zgłębiać w moim towarzystwie, to proszę, daj mi trochę czasu. Kolejny dodatkowy rozdział o C++17 napiszę wkrótce, ale dopiero wtedy, gdy sam nabiorę do tych zagadnień trochę dystansu. Gdy trochę popracuję z C++17 i będę miał jakieś swoje osobiste refleksje na temat tych nowelizacji. Zatem: cierpliwości!