
TOM 5
ŁATWY PODRĘCZNIK
Jerzy Gręboszz
Opus magnum C++
Wyprawa w głąb
C++20
Tl 196
Pt 191
2.8 d
Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki i opracowanie graficzne książki: Jerzy Grębosz

Zdjęcie Mgławicy Orzeł w grafice na okładce oraz zdjęcia łazika Curiosity i powierzchni Marsa – wykorzystane w tytułach rozdziałów – dzięki uprzejmości NASA.

Wydanie pierwsze

ISBN: 978-83-289-3984-4

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/omcpp5
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
https://ftp.helion.pl/przyklady/omcpp5.zip

Spis treści

0 Proszę tego nie czytać!

0.1 Wyruszamy na kolejną wyprawę!

Ta książka jest następnym tomem cyklu ***Opus magnum C++***

Jeśli dotarłeś aż tutaj, to znaczy, że znamy się już od dawna. Być może spotkaliśmy się przy pierwszych tomach *Opus magnum C++*, może nawet w kolejnym, czwartym tomie *Misja w nadprzestrzeń C++14/17*. Tak czy inaczej – witaj ponownie, stary przyjacielu. Cieszę się, że znów wyruszamy razem.

> *Poprzednia książka zabrała nas wysoko – w „nadprzestrzeń" nowych standardów C++14 i C++17. Tym razem kierunek naszej podróży będzie nieco inny. Nie polecimy wyżej, lecz zejdziemy głębiej. Standard C++20 wprowadził bowiem zmiany, które nie tylko dodają nowe narzędzia, ale w wielu miejscach sięgają do samej struktury języka.*

W trakcie tej wyprawy spotkamy mechanizmy, które znacząco zmieniają sposób myślenia o programach

Zobaczymy nowe atrybuty pomagające kompilatorowi podejmować lepsze decyzje. Poznamy trójstanowy operator porównania, który porządkuje świat relacji między obiektami. Przyjrzymy się nowoczesnemu formatowaniu tekstu oraz drobnym udogodnieniom składni, dzięki którym kod staje się bardziej przejrzysty.

Potem zejdziemy jeszcze głębiej. Pojawią się narzędzia, takie jak `span` czy biblioteka `ranges`, które wprowadzają nowy sposób pracy z sekwencjami danych. Spotkamy koncepty – subtelny, ale bardzo potężny mechanizm pozwalający kompilatorowi lepiej rozumieć intencje programisty. Zajrzymy do świata modułów, które porządkują strukturę dużych projektów. A na końcu naszej drogi czekają korutyny – jeden z najbardziej fascynujących mechanizmów współczesnego C++.

Wszystkie te elementy razem pokazują coś ważnego: C++ nie stoi w miejscu

Język, który przez dziesięciolecia był narzędziem wymagającym ogromnej dyscypliny, staje się coraz bardziej wyrazisty i coraz lepiej dopasowany do sposobu myślenia programisty. C++20 jest jednym z największych kroków w tej ewolucji.

Mam nadzieję, że ta książka będzie dla Ciebie nie tylko przewodnikiem po nowych elementach standardu, ale także inspiracją do dalszych poszukiwań. Najlepszym sposobem poznawania języka programowania jest przecież zawsze własne eksperymentowanie z kodem.

Kilka uwag typograficznych

✧ Przyzwyczaiłeś się już pewnie, że w *Opusie*, na dole strony, w przypisach, pojawiają się uwagi o wymowie niektórych angielskich słów występujących w C++. Wiesz więc, że nie są przeznaczone dla takiego wytrawnego anglisty jak Ty, lecz dla najmłodszych uczących się czytelników.

✧ W przykładowych programach w tej książce występują nazwy zmiennych. Celowo wymyślam dla nich polskie nazwy, bo wtedy wyraźnie widać, co w programie jest naszą twórczością, a co stanowią angielskie słowa kluczowe języka C++. Zatem w naszych programach będziesz mógł na przykład spotkać się ze zmiennymi o nazwach moc_dopuszczalna, dzwiek. Oczywiście ten ostatni wyraz to dźwięk, ale tych polskich liter nie wolno mi użyć w programie. Nazwy zmiennych i funkcji powinny składać się wyłącznie z liter alfabetu angielskiego. Dlatego trzeba jednak pisać trochę dziwacznie: dzwiek. Wkrótce się z tym oswoisz.

Z drugiej strony, gdy później opisujemy program, pozwalam sobie używać polskich liter, bo w polskim zdaniu wygląda to czytelniej. Zatem piszę, że „do funkcji żmudna_funkcja wysyłamy argument dźwięk". W tych opisach pozwalam sobie nawet na coś więcej: pozwalam sobie na fleksję, czyli końcówki odmiany przez przypadki. Na przykład piszę, że „wywołujemy żmudną_funkcję, inkrementujemy moc_dopuszczalną" lub: „do dźwięku przypisujemy wartość 330".

Pamiętaj, że te zabiegi spolszczające wolno mi robić poza programem – w jego opisie. Natomiast w samym programie nazwa zmiennej musi być zawsze bez tych polskich liter – i zawsze w mianowniku.

Jak zawsze będę bardzo wdzięczny za wszelkie uwagi czytelników

Jeśli zauważysz błąd, nieścisłość albo fragment wymagający doprecyzowania, napisz do mnie. Nawet najdrobniejsza uwaga może pomóc ulepszyć kolejne wydania tej książki. Mój adres mailowy to:

jerzy.grebosz@ifj.edu.pl

Kod źródłowy przykładowych programów oraz ewentualna errata są dostępne na mojej stronie internetowej związanej z tą książką. Aktualny adres mojej strony to:

https://www.ifj.edu.pl/private/grebosz/new_look/ksiazki.html

W razie gdyby adres mojej strony się zmienił, łatwo go będzie znaleźć w internecie, wpisując hasła: *jerzy grebosz private ksiazki*.

Po każdym rozdziale tej książki znajdują się ćwiczenia i zadnia

Odpowiedzi do nich także szukaj na tej części mojej strony, która poświęcona jest *Wyprawie w głąb C++20*.

https://www.ifj.edu.pl/private/grebosz/uniw_php/wybor.php

A zatem wyruszamy w kolejną podróż – tym razem w głąb języka C++. Kto wie, jakie jeszcze odkrycia czekają na nas po drodze. Jest takie powiedzenie:

> Idąc tą drogą, pomyśl o tych, którzy szli tędy, gdy tej drogi jeszcze nie było.

– powiedzenie australijskie

6 *consteval*, *constinit* – kiedy warto liczyć w compile-time

W C++20 pojawiają się dwa nowe słowa kluczowe. To consteval oraz constinit. W obu tych nazwach obecne jest słowo „const", więc można by pomyśleć, że chodzi o obiekty z przydomkiem const. A jednak tak nie jest.

Człon „const" w specyfikatorach (przydomkach) consteval, constexpr oznacza, że chodzi nam o akcję jeszcze na etapie kompilacji programu (*compile-time*). W standardzie nie ma bowiem specjalnego określenia: „oblicz w czasie kompilacji". Zamiast tego są właśnie określenia consteval i constexpr.

6.1 *consteval* – funkcje wykonywane tylko w trakcie kompilacji

Słowo kluczowe consteval stawia się tylko jako przydomek do funkcji. W tym słowie widzisz człon „eval", który jest skrótem od angielskiego *evaluation* – czyli „obliczanie".

Stawiając to słowo przy definicji funkcji, wyrażamy życzenie, by funkcja obliczała stałą wartość swojego rezultatu jeszcze w trakcie kompilacji naszego programu. Jeśli nie da się tego zrobić, kompilator ma obowiązek zgłosić błąd kompilacji.

> *Taką funkcję nazywa się czasem funkcją natychmiastową (ang.* immediate function*), dlatego że wykonywana jest natychmiast, jeszcze w czasie kompilacji.*

Co to znaczy, że funkcję da się obliczyć w trakcie kompilacji programu?

Kiedyś już o tym mówiliśmy, ale przypomnę. Obecne kompilatory są bardziej inteligentne niż dawniej. Potrafią nie tylko kompilować, ale nawet mogą wykonywać proste operacje arytmetycznie. Kompilator, widząc zapis:

```
int m = 100 + 30 / 10;
```

potrafi obliczyć to wyrażenie i do zmiennej m wstawi obliczoną wartość 100 + (30/10), czyli wartość 103.

> Teraz kompilatory są jeszcze sprytniejsze. Potrafią wykonywać nawet proste funkcje pod warunkiem, że ich argumentami są **stałe** znane im już w czasie kompilacji.

Zatem jeśli w trakcie pracy nad naszym programem kompilator napotka gdzieś w tekście programu wywołanie funkcji z przydomkiem consteval, to sam od razu tę funkcję wykona. Następnie w to miejsce programu wstawi wartość będącą obliczonym przez siebie rezultatem tej funkcji.

(Trochę tak, jak to dzieje się z funkcjami inline). Zatem taka funkcja nawet nie zaistnieje w Twoim wynikowym programie. We wszystkich miejscach, gdzie jest jej wywołanie, zostanie zastąpiona obliczoną przez siebie wartością.

> Zauważ, że tę obliczoną przez kompilator stałą wartość możemy przypisać do obiektu. Obiekt ten wcale nie musi mieć przydomka const.

Oto przykład takiej sytuacji:

```
#include <iostream>
using namespace std;
//*****************************************************************************************************************
consteval int fun_kompil(int x)                                                                          ❶
{
    return x * x;
}
//*****************************************************************************************************************
int nglobal = fun_kompil(10);                                                                            ❷
//*****************************************************************************************************************
int main()
{
    cout <<" nglobal = " << nglobal << endl;          // wydruk: nglobal = 100                          ❸
    // przypisanie nowej wartości do zmiennej globalnej
    nglobal = fun_kompil(4);                                                                             ❹
    cout <<" nglobal = " << nglobal << endl;          // wydruk: nglobal = 16                           ❺


    int nlokalna = 100 + fun_kompil(5);                                                                  ❻
    cout <<" nlokalna = " << nlokalna << endl;        // wydruk: nlokalna = 125

    // funkcja consteval może wystąpić w dowolnym wyrażeniu
    double x = 10.7 + fun_kompil(2);                                                                     ❼
    cout <<" x = " << x << endl;                      // wydruk: x = 14.7
}
```

Krótkie omówienie

❶ Oto definicja bardzo prostej funkcji z przydomkiem consteval. Słowo to wolno nam postawić tylko w definicji/deklaracji funkcji. Funkcja ma nazwę fun_kompil, a z jej ciała widać, że przysłany do niej argument typu int podnosi do kwadratu i to zwraca jako rezultat.

❷ Jak widać, funkcja fun_kompil może zostać użyta na przykład do inicjalizacji obiektu globalnego. Rezultat tej inicjalizacji możemy potem sprawdzić dzięki instrukcji ❸.

❹ Obiekt nglobal wcale nie jest stały, więc można do niego jeszcze raz coś przypisać. W wyrażeniu przypisującym możemy na przykład powtórnie użyć wywołania funkcji consteval. Skoro teraz argument jest inną stałą wartością, to tym razem kompilator obliczy 4 * 4 = 16. Taką wartością zastąpi w tym miejscu to wywołanie fun_kompil(4).

❺ Dla sprawdzenia wypisujemy na ekranie wartość nglobal. Jest tam teraz 16.

❻ Można też użyć funkcji fun_kompil w wyrażeniu inicjalizującym zmienną lokalną o nazwie nlokalna.

Teraz argumentem funkcji jest stała wartość 5. Kompilator oblicza 5 * 5 = 25 i tą wartością zastępuje wywołanie fun_kompil(5). Na ekranie zostanie więc wypisana wartość 125 (czyli 100 + 25).

❼ Oczywiście można wywołania funkcji z przydomkiem consteval użyć w dowolnym wyrażeniu, tutaj na przykład korzystamy z niego przy przypisywaniu do obiektu typu double.

Podsumujmy:

> Słowo kluczowe consteval stawiamy w definicji funkcji, którą kompilator powinien wywoływać i wykonywać jeszcze w czasie kompilacji programu. Wywołania takiej funkcji możemy umieszczać w różnych miejscach programu z odpowiednimi stałymi argumentami. Kompilator obliczy te wywołania i ich wynik wstawi w miejsce wywołania.

W naszym przykładzie funkcja consteval była bardzo prosta. Nieprzypadkowo.

Funkcja natychmiastowa (consteval) jest przez domniemanie inline i musi spełniać wymagania funkcji constexpr.

Nie każda funkcja nadaje się do tego, by mieć przydomek *consteval*

Obowiązują tu takie same obostrzenia, jakie były przy funkcjach constexpr.

W funkcjach consteval mogą się znaleźć:

- proste obliczenia z użyciem takich operatorów jak: +, −, *, /, %, &&, ||, !,
- instrukcje sterujące: if, for, while, switch,
- rekurencja (jeśli da się rozwinąć w *compile-time*),
- lokalne zmienne i tablice,
- wywołania innych funkcji constexpr i consteval,
- zwracanie wartości rezultatu (zawsze obowiązkowe),
- typy wbudowane i proste struktury (struct, enum),
- tworzenie obiektów, które mają konstruktory constexpr.

Nie można w nich:

- używać zmiennych *run-time* (np. argumentów zwykłych funkcji, zmiennych globalnych bez constexpr),
- wykonywać operacji I/O (std::cout, pliki, operacje sieciowe),
- robić dynamicznej alokacji (new, delete, malloc, free),
- robić rzutowań/operacji niedozwolonych w constexpr,
- modyfikować globalnych zmiennych (chyba że są constexpr/constinit),
- mieć ciała, które nie zwracają wartości (void może wystąpić tylko w nielicznych metaprogramistycznych zastosowaniach).

Destruktor nie może być oznaczony specyfikatorem consteval.

Jak funkcje *consteval* mają się do funkcji *constexpr*?

Znamy już podobną sytuację – funkcje z przydomkiem constexpr. (Możliwe już były w C++11, a w C++14 rozszerzono ich możliwości). Funkcje z przydomkiem constexpr także **mogły** pracować jeszcze w czasie kompilacji programu.

Mógłbyś więc zapytać: jaka jest między nimi różnica?

Różnica między funkcjami consteval a funkcjami constexpr jest taka, że:

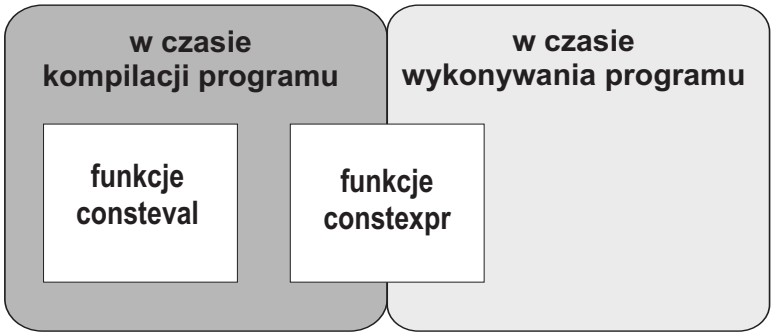
- Funkcje constexpr kompilator mógł wywołać w trakcie kompilacji, jeśli było to możliwe, czyli jeśli argumenty wywołania były stałymi constexpr.
  Jeśli jednak argumenty wywołania nie były stałymi constexpr (tylko jakimiś zmiennymi), to kompilator uznawał, że mają być wywołane dopiero w czasie pracy programu (czyli *run-time*).
- Natomiast funkcje consteval mogą pracować **tylko** w czasie pracy kompilatora (*compile-time*). Wywołanie ich z argumentami będącymi zmiennymi znanymi dopiero w trakcie pracy programu (*run-time*) ma wywołać protest kompilatora.

Funkcje *consteval* gwarantują, że zostaną wykonane jeszcze w czasie kompilacji

Dzięki temu jeszcze w czasie kompilacji można użytkownikowi przygotować nawet wielkie tablice stałych. Co więcej,

> tak przygotowane stałe mogą być nawet stałymi „odwiecznymi" (constexpr), czyli takimi, które mogą wystąpić w programie wszędzie. Nawet tam, gdzie zwykłe obiekty const by nie wystarczyły (np. jako rozmiar definiowanej tablicy czy jako stałe przy etykietach case).

Różnicę w czasie wykonywania funkcji constexpr i consteval można sobie wyobrazić za pomocą takiego grafu.



Funkcja *constexpr* może być wykonana w czasie kompilacji, ale nie musi.
Funkcja *consteval* musi być wykonana w czasie kompilacji.

Kiedy zatem przydają się funkcje z przydomkiem *consteval*?

- Gdy chcemy mieć poczucie bezpieczeństwa, że funkcja nie zostanie mimo wszystko cofnięta do wykonywania w czasie pracy programu (tak mogłoby się zdarzyć, gdybyśmy ją zrobili jako constexpr). Tymczasem consteval daje nam pewność, że będzie wykonana w trakcie kompilacji (a jeśli to niemożliwe, to kompilator zasygnalizuje błąd kompilacji).
- Gdy chcemy już w trakcie kompilacji poznać rozmiary typów, struktur, klas.
- Wtedy gdy jeszcze w czasie pracy kompilatora chcemy wytworzyć jakieś stałe czy maski bitowe. Takie stałe czy maski będą mogły mieć cechę constexpr („stałe odwieczne"), czyli będą się nadawały do użycia np. w etykietach case.

- Wtedy gdy jeszcze w czasie pracy kompilatora chcemy wytworzyć tablice – o określonym, stałym rozmiarze. (Pamiętamy, że rozmiar takich tablic musi być stałą znaną już w czasie kompilacji – właśnie „stałą odwieczną").
- Wtedy gdy chcemy w czasie pracy kompilatora nie tylko wytworzyć takie tablice, ale nawet zapełnić je jakimiś stałymi wartościami. Obliczenia takich stałych wartości mogą czasem zająć dużo czasu. Lepiej obliczyć je jeszcze w czasie kompilacji.

Jak się to robi konkretnie, zobaczymy w następnym paragrafie.

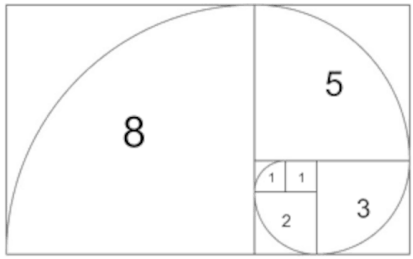
6.1.1 Tablica liczb obliczana w czasie kompilacji programu

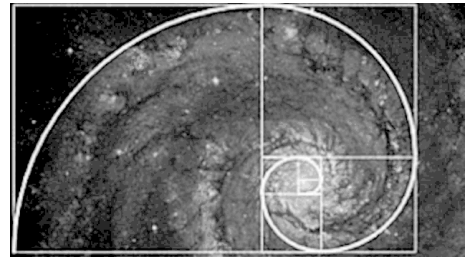
Zobaczymy teraz prosty przykład pokazujący, jak w trakcie kompilacji wytworzyć tablicę i od razu wypełnić ją określonymi wartościami.

> *Gdybyś chciał w ten sposób wytworzyć tablice funkcji trygonometrycznych, to mam (chwilowo) smutną wiadomość. Funkcje trygonometryczne takie jak std::sin w <cmath> w standardzie C++20 nie są jeszcze typu constexpr. Zatem z tablicami trygonometrycznymi zaczekamy do C++23.*

Pokażę więc, jak w trakcie kompilacji przygotować tablicę elementów prostego, ale bardzo interesującego ciągu. Będzie to bardzo powszechny w przyrodzie ciąg Fibonacciego. Co to za ciąg? Jego wyrazy mają tę własność, że kolejny wyraz (z wyjątkiem dwóch pierwszych) jest sumą dwóch poprzednich wyrazów.

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	...
0	1	1	2	3	5	8	13	21	34	...





Przed przykładem małe zastrzeżenie.

> Nie daj się zwieść. To nie ten ciąg jest przedmiotem tego paragrafu. On służy tylko do ilustracji, jak za pomocą funkcji z przydomkiem consteval wytworzyć tablicę, jak ją wypełnić i jak ją w całości zwrócić jako rezultat pracy tej funkcji.

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
struct Tstruktura_ciag_Fibonacciego                                                                                          ❶
{
    unsigned long long element[70];          // pierwsze 70 wyrazów ciągu
```

```
};
//*****************************************************************************************************************
consteval Tstruktura_ciag_Fibonacciego    zbuduj_tablice_fib()                                    ❷
{
    Tstruktura_ciag_Fibonacciego t { };                                                         ❸
    t.element[0] = 0;
    t.element[1] = 1;
    for (int i = 2; i < 70; i++)
    {
        t.element[i] = t.element[i−1] + t.element[i−2];                                         ❹
    }
    return t;                                                                                   ❺
}
//*****************************************************************************************************************
constexpr     Tstruktura_ciag_Fibonacciego   fib = zbuduj_tablice_fib();                      ❻
//*****************************************************************************************************************

int main()
{
    for(int n = 0 ; n < 70 ; ++n)
    {
        cout << "F(" << n << ") = " << fib.element[n] << "\n";                                  ❼
    }
}
//*****************************************************************************************************************
```

Na ekranie zobaczymy wiele elementów ciągu. Część z nich wycinam dla skrócenia wydruku.

```
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
F(10) = 55
------------------------ ciach --------------------
F(65) = 17167680177565
F(66) = 27777890035288
F(67) = 44945570212853
F(68) = 72723460248141
F(69) = 117669030460994
```

Krótkie omówienie

Jak dobrze pamiętasz, jeśli do funkcji wysyłamy tablicę, to w rzeczywistości otrzymuje ona wskaźnik do niej. Podobnie jest, gdy funkcja zwraca tablicę. Rezultatem funkcji staje się tylko jedna wartość – adres tej tablicy.

Innymi słowy, nawet jeśli funkcja zdefiniuje sobie jakąś lokalną tablicę i wypełni ją jakimiś wartościami, to nie może zwrócić tej tablicy jako rezultatu swej pracy.

Jedyne, co może zwrócić, to adres tej tablicy. Niestety będzie to „adres domu, który właśnie zostaje zburzony". Koniec przypomnienia.

> Jest jednak prosty sposób przesłania całej tablicy. Tablicę taką zapakujemy do struktury. Obiekt takiej struktury funkcja może zwrócić przez wartość. W zwróconym obiekcie będzie dokładna kopia całej naszej tablicy.

❶ Oto definicja prostej struktury o nazwie Tstruktura_ciag_Fibonacciego. Jej jedynym składnikiem jest tablica 70 elementów typu unsigned long long. (Potrzebny jest aż tak duży typ danej, bo (dalsze) wyrazy ciągu osiągają niemal astronomiczne wartości).

❷ Oto istota naszego przykładu. Jest to funkcja o nazwie zbuduj_tablice_fib. Ma ona przydomek (specyfikator) consteval, który gwarantuje, że funkcja będzie działać tylko w czasie kompilacji programu.
Zwróć uwagę, że:

a) funkcja (akurat tutaj) nie przyjmuje żadnych argumentów,

b) jako rezultat funkcja zwraca przez wartość (jeden) obiekt klasy Tstruktura_ciag_Fibonacciego.

❸ W ciele funkcji definiujemy lokalny obiekt t, który jest klasy Tstruktura_ciag_Fibonacciego. W definicji obiektu t widzimy nawiasy {}. Oznacza to, że obiekt t i będąca jego składnikiem tablica element zostają wstępnie wypełnione zerami.

W następnych instrukcjach widzimy przypisania. Do pierwszych dwóch elementów tablicy przypisujemy wartość 0 i wartość 1 – bo taka jest zasada w ciągu Fibonacciego. Niech Cię nie dziwi składnia. Przecież Tstruktura_ciag_Fibonacciego jest *strukturą*, więc mamy publiczny dostęp do jej składowej tablicy o nazwie element.

❹ Tu następuje pętla for, która oblicza dalsze elementy tablicy według zasady, że każdy następny element jest sumą dwóch poprzednich.

❺ A teraz uważaj: po zakończeniu wypełniania lokalnej struktury (a konkretnie wypełnieniu będącej jej składnikiem tablicy) obiekt tej klasy stawiamy przy instrukcji return. Zatem teraz obiekt tej struktury (ma on nazwę t) zostaje w całości zwrócony przez wartość (czyli przez kopię).

❻ Oto miejsce w programie, gdzie korzystamy z tego rezultatu. Tu wywołana zostaje funkcja zbuduj_tablice_fib. Rezultat pracy tej funkcji zostaje przypisany do obiektu fib, którego typ to Tstruktura_ciag_Fibonacciego.

> W ten sposób więc, jeszcze w czasie kompilacji, w obiekcie fib znajdzie się treść wytworzonej tablicy z elementami ciągu Fibonacciego.

Mógłbyś zapytać: czy w definicji obiektu fib potrzebny jest specyfikator (przydomek) constexpr?

Nie, nie jest konieczny, ale często stawia się go tutaj choćby po to, by przypomnieć, że ta tablica powstanie w czasie kompilacji.

Postawienie tu przydomka constexpr daje jeszcze jedną korzyść: elementy tej tablicy same mogą służyć nam jako źródło stałych przydatnych w różnych miejscach programu, gdzie są potrzebne „stałe odwieczne" .

❼ W funkcji main widzimy, jak (już w czasie zwykłego wykonania programu) możemy korzystać z przygotowanego wcześniej obiektu fib (a konkretnie z jego wewnętrznej tablicy element). Obiekt ten zawiera przygotowane w czasie kompilacji poszczególne wyrazy ciągu Fibonacciego.

Czy funkcja *consteval* może w swoim ciele modyfikować zmienne globalne?

Nie! Funkcja consteval zawsze działa w czasie kompilacji, a wtedy nie ma „normalnych" zmiennych globalnych, które można zmieniać.

W czasie kompilacji funkcja consteval może operować tylko na:

- argumentach tej funkcji,
- zmiennych lokalnych w tej funkcji,
- wartościach zwracanych przez tę funkcję.

Czyli: funkcja consteval nie może w swoim ciele przypisywać do globalnej zmiennej.

Jak więc poprawnie zrealizować inicjalizację zmiennej globalnej?

Jeśli chcesz, żeby funkcja consteval „ustawiła" globalną wartość, to musi to wyglądać na przykład tak:

```
consteval int    init_global()
{
    return 42;
}

constexpr int global = init_global();          // OK
```

Globalna zmienna global zostaje tu zainicjalizowana **wynikiem funkcji consteval przypisanym do niej**. Sama funkcja nie może zmienić jej „ze swojego ciała".

6.2 *constinit* – inicjalizacja obiektów statycznych w czasie kompilacji

Teraz zajmiemy się drugim ze słów kluczowych będących przedmiotem tego rozdziału. W tytule paragrafu zobaczyłeś określenie „obiekty statyczne". Zatem krótkie przypomnienie:

> *określenie „obiekty statyczne" jest jakby przeciwieństwem określenia „obiekty automatyczne", (czyli takie, które zwykle definiujemy lokalnie w ciałach funkcji, a którym nie towarzyszy przydomek* static*). Obiekty automatyczne są tworzone na stosie.*

Do obiektów statycznych należą więc:

> obiekty globalne,
>
> definiowane w funkcjach obiekty static,
>
> obiekty definiowane jako thread_local (to takie, które żyją od startu wątku do końca wątku).

Koniec przypomnienia.

Słowo kluczowe *constinit*

W nowym słowie kluczowym constinit jest człon *init*, który jest skrótem od angielskiego *initialisation* (inicjalizacja).

Zatem *constinit* to przydomek, który możemy nadać zmiennej lub stałej **statycznej** (czyli np. globalnej). Widząc go, kompilator inicjalizuje dany obiekt statyczny jeszcze w trakcie kompilacji.

*Podkreślmy: inicjalizuje go za pomocą stałej, ale **sam inicjalizowany obiekt nie musi być stałą**. Później, już w trakcie pracy programu, może się zmieniać.*

Tu chodzi tylko o to, że ten obiekt już na etapie kompilacji musi zostać zainicjalizowany, czyli musi otrzymać jakąś wstępną wartość.

Oto prosty przykład. Pokazuje inicjalizację zmiennej globalnej stałą wartością. W trakcie pracy programu zmienną tę można modyfikować.

```
#include <iostream>
using namespace std;
constinit int  zmienna_globalna = 5;  // musi być inicjalizowana w compile-time      ❶
//*************************************************************************************************************

int main()
{    cout << zmienna_globalna << endl;              // wstępna wartość 5

     zmienna_globalna = 42;                         // OK, można zmieniać            ❷
     cout << zmienna_globalna << endl;              // nowa wartość 42

     // constinit int   lokalna_automatyczna = 6;   // błąd                          ❸

     static constinit int  lok_stat = 6;                                             ❹
     cout << "zmienna lokalna statyczna = " << lok_stat << endl;   // wartość = 6
}
```

Sprawa jest prosta

❶ Inicjalizacja obiektu zmienna_globalna stałą 5 odbywa się jeszcze w czasie kompilacji. W pierwszej instrukcji funkcji main można zobaczyć tego potwierdzenie. Z definicji widać, że obiekt zmienna_globalna nie jest stały.

❷ Oto potwierdzenie, że w trakcie pracy programu można zmienić wartość tego obiektu.

❸ A to próba użycia słowa kluczowego constinit wobec zmiennej, która nie jest statyczna. Na to kompilator nie pozwala.

❹ Za to ten obiekt, o nazwie lok_stat, ma obok constinit także przydomek static, więc ta inicjalizacja w czasie kompilacji jest możliwa.

Czy zmienna może być równocześnie *const* i *constinit*?

Tak, może. To znaczy, że jest stała i jej wartość nadano w czasie inicjalizacji. Kolejność postawienia tych słów nie ma znaczenia.

Czy zmienna może być równocześnie *constexpr* i *constinit*?

Nie!

Z zestawu *constexpr*, *consteval* i *constinit* tylko jedno z tych słów kluczowych może się pojawić w deklaracji obiektu.

Złożenie słów *const constinit* jest w porządku – nie jest to równoważne *constexpr*, bo:

- const constinit to „stała globalna, bezpiecznie zainicjalizowana” (ale nie „stała odwieczna”),
- natomiast constexpr to „pełnoprawna stała możliwa do postawienia w każdym wyrażeniu” (czyli „stała odwieczna”).

Do jakich zmiennych może ten przydomek *constinit* być zastosowany?

Może być użyty tylko do zmiennych o czasie życia statycznym lub thread storage duration.

Czy *constinit* przyspiesza działanie programu?

Minimalnie. Jednak właściwa korzyść z inicjalizacji zmiennej w czasie kompilacji polega na tym, że nie wymaga ona wykonywania żadnych instrukcji inicjalizacji podczas wykonywania programu. Pomaga więc programistom upewnić się, że inicjalizacja odbyła się poprawnie bez konieczności zgadywania lub sprawdzania wygenerowanego zestawu.

Zatem przydomka *constinit* używamy:

- do inicjalizacji zmiennych **statycznych**, które mają znaną wartość startową w czasie kompilacji, ale niekonieczne są stałe w czasie wykonywania;
- do ochrony przed błędami inicjalizacji między różnymi plikami *.cpp.

Zastanowiło Cię pewnie to ostatnie zdanie. O tym porozmawiamy teraz.

6.2.1 *constinit* – bezpieczna inicjalizacja zmiennych globalnych

> Jeśli Twój program składa się z wielu plików i w tych plikach masz zdefiniowane różne zmienne globalne, to pamiętaj, że kolejność ich inicjalizacji jest niezdefiniowana.

Zwykle nie stanowi to problemu. Wyobraź sobie jednak, że postanowiłeś być sprytniejszy. Postanowiłeś zmienną globalną **x** z jednego pliku inicjalizować przy użyciu wartości zmiennej globalnej **y** z innego pliku. Tu mogą pojawić się problemy. Jest ryzyko, że zmienna **y** ma w danej chwili jeszcze wartość niezdefiniowaną.

Oto schemat takiej sytuacji. Mamy tu trzy pliki: main.cpp, a.cpp i b.cpp.

plik *a.cpp*:

```
#include <iostream>

int get_y();                // deklaracja funkcji z innego pliku

int x = get_y() + 1;        // definicja obiektu x inicjalizowanego wartością
                            // zmiennej globalnej y, którą funkcją sprowadzamy z innego pliku
```

plik *b.cpp*:

```
#include <iostream>

int y = 42;
int get_y() { return y; }
```

plik *main.cpp*:

```
#include <iostream>
//*************************************************************************************************************
extern int x;
extern int y;

int main()
{
    std::cout << "x = " << x << ", y = " << y << "\n";
}
```

Robimy to po staremu, więc żadna praca nie jest wykonywana w czasie kompilacji

Skoro tak, to wszystkie inicjalizacje zdarzą się na samym początku wykonywania programu, jeszcze przed wejściem programu do funkcji main.

Program startuje i najpierw zaczyna inicjalizację wszystkich zmiennych globalnych w poszczególnych plikach. Musi jednak zacząć od któregoś konkretnego. Jeśli zacznie inicjalizowanie globalnych zmiennych od pliku a.cpp, to „zobaczy" w zakresie globalnym taką definicję:

```
int x = get_y() +1;
```

Wywołuje więc funkcję get_y(), a ona sięga do obiektu globalnego **y**, który nie został jeszcze zainicjalizowany, bo na niego kolej przyjdzie dopiero za chwilę. Co zatem zwróci funkcja get_y()? Nieokreśloną jeszcze wartość.

> To właśnie problem: globalna zmienna **x** używa innej globalnej (**y**) w chwili, gdy ta nie została jeszcze zainicjalizowana.

Jak się przed tym uchronić? Użyć słowa kluczowego *constinit*

Oto jak to zorganizować.

plik *b.cpp*:

```
#include <iostream>
constinit int y = 42;          // y na pewno gotowe                    ❶
```

plik *a.cpp*:

```
#include <iostream>
extern constinit int y;        // używamy bez ryzyka                   ❷
constinit int x = y + 1;       // x = 43, bez niespodzianek            ❸
```

plik *main.cpp*:

```
#include <iostream>
//*************************************************************************************************************
extern constinit int a;
extern constinit int b;
//*************************************************************************************************************
int main()
{
```

```
    std::cout << "a = " << a << ", b = " << b << "\n";
}
```

Wyjaśnienie

Przy definicjach obu tych zmiennych globalnych **x** i **y**, czyli w...

❶ i ❸ postawiliśmy przydomek constinit. Słowo kluczowe constinit gwarantuje inicjalizację jeszcze w czasie kompilacji. Zatem zarówno **x**, jak i **y** są gotowe, zanim program zacznie się wykonywać.

Jeśli nawet kompilator zaczął pracę od pliku a.cpp, to tam zobaczył...

❷ deklarację extern mówiącą, że chodzi o obiekt **x** z pliku b.cpp.

❸ To jest definicja obiektu **x**, którego wartość startowa ma wynikać z wartości zewnętrznej (extern) zmiennej globalnej **y**, a ona ma dodatkowo przydomek constinit.

❶ Zatem najpierw kompilator „spojrzy" czy już ją inicjalizował. Jeśli jeszcze nie, to zrobi to teraz. Następnie wróci do zajmowania się inicjalizacją obiektu **a**. Obliczy znaną mu teraz wartość wyrażenia **y + 1** i to wstawi do obiektu **x**.

Bez constinit globalne zmienne mogłyby być inicjalizowane w nieprzewidywalnej kolejności, a to mogłoby grozić błędami. Dzięki constinit mamy zawsze pewną inicjalizację jeszcze w czasie kompilacji, czyli zero niespodzianek.

Teraz kolejność inicjalizacji nie ma znaczenia, bo wszystko policzone zostało w trakcie kompilacji. Gdy program startuje, wartości zmiennych globalnych są już gotowe w pamięci.

6.2.2 Dobre praktyki: jak definiować i inicjalizować zmienne globalne

W dużych projektach może się tak zdarzyć, że jedna globalna zmienna ma być używana w wielu plikach. Jak to zorganizować, żeby było bezpiecznie i żeby unikać powielania?

Oto zasada. Tworzymy plik, gdzie gromadzimy wszystkie zmienne globalne. Będzie się on naprawdę składał z dwóch plików: pliku *.cpp i pliku nagłówkowego *.h.

Plik nagłówkowy *config.h*:

```
// deklaracje zmiennych globalnych
extern constinit     int  max_users;
// ...itd.
```

Plik *config.cpp*:

```
#include "config.h"

constinit int     max_users = 100;       // definicja  zmiennej globalnej
// ...
```

Jak widać:

- tam, gdzie zmienną globalną **definiujemy**, stosujemy słowo kluczowe constinit,
- tam, gdzie tylko ją **deklarujemy**, stosujemy słowa kluczowe extern constinit.

W pozostałych plikach programu, w których z tych zmiennych zamierzamy korzystać, wstawiamy plik nagłówkowy.

Plik *main.cpp*:

```
#include "config.h"                    // <---Tu wstawiamy nagłówek
#include <iostream>
//*****************************************************************************************************************
int main()
{
    std::cout << max_users << "\n";   // użycie
}
```

Podsumowanie

Jeśli masz problem z niepewną kolejnością inicjalizacji globalnych zmiennych, to rozwiązaniem jest constinit umieszczone przy ich definicji. Przydomek constinit wymusza ich inicjalizację jeszcze w czasie kompilacji. W deklaracji takiej zmiennej globalnej (w innych plikach) stosuj kombinację constinit + extern.

Tworząc jedną definicję zmiennej globalnej, a potem jej wiele deklaracji, otrzymasz w rezultacie kod bezpieczny, przewidywalny i łatwy do utrzymania.

6.3 Kilka przypomnień i przestróg

- Używaj consteval tam, gdzie wynik musi być znany w czasie kompilacji.
- Używaj constinit do globalnych zmiennych (unikniesz sytuacji związanych z nieokreślonymi wartościami wynikłych z niepewnej kolejności inicjalizacji).
- Zawsze łącz constinit z extern w wielu plikach, zamiast inicjalizować kilka razy.
- constexpr nadal pozostaje domyślnym wyborem dla funkcji i zmiennych stałych.

Na sam koniec tabelka, gdzie zestawione są różne sytuacje, do których nadaje się lub nie określone słowo kluczowe.

Porównanie zastosowania trzech słów kluczowych w różnych sytuacjach

Cecha	constexpr	consteval	constinit
Obliczanie w compile-time	✔ (jeśli możliwe)	zawsze	tylko inicjalizacja
Może działać w run-time	✔	—	✔
Wartość stała w run-time	✔	✔	—
Działa dla funkcji	✔	✔	—
Działa dla zmiennych	✔	—	✔

6.4 Ćwiczenia – *consteval* i *constinit*

I Co oznacza specyfikator consteval przy funkcji?

II Podaj przykład funkcji consteval i jej poprawnego użycia.

III Co się stanie, jeśli spróbujesz wywołać funkcję consteval w *run-time*?

IV Do czego służy specyfikator constinit?

V Czy constinit oznacza, że zmienna jest stała (niezmienna)? Wyjaśnij.

VI Napisz przykład zmiennej globalnej, która korzysta z constinit.

VII Jakie są korzyści stosowania consteval i constinit w porównaniu z constexpr?

VIII Co stanie się, jeśli constinit zostanie użyte, ale inicjalizacja zmiennej nie będzie mogła odbyć się w *compile-time*?

IX Porównaj consteval i constexpr. Podaj jedną zasadniczą różnicę.

X Napisz funkcję consteval, która oblicza sześcian liczby, i użyj jej do inicjalizacji zmiennej z constinit.

Skorowidz

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Opus magnum C++

TOM 5

Wyprawa w głąb C++20

Poznawaj C++20 tak, jakby tłumaczył Ci go doświadczony kolega, z którym siedzisz przy jednym biurku

C++20 to największa od czasu wersji z numerem 11 aktualizacja języka C++. Wprowadza dziesiątki nowych mechanizmów, pozwalających pisać kod, który jest nowocześniejszy, bezpieczniejszy, czytelniejszy i wygodniejszy w utrzymaniu.

Wyprawa w głąb C++20 to kolejny tom cenionego cyklu *Opus magnum C++*, poświęconego najważniejszym zagadnieniom związanym z językiem C++. Autor, korzystając z wieloletniego doświadczenia zdobytego w międzynarodowych projektach, prowadzi czytelników krok po kroku przez nowe możliwości standardu C++20. Tak jak w poprzednich tomach, trudniejsze kwestie wyjaśnia prostym, przyjaznym językiem, a dodatkowo ilustruje je licznymi przykładami i praktycznymi programami.

Czego się nauczysz?

- Poznasz nowe atrybuty, dzięki którym skuteczniej porozumiesz się z kompilatorem
- Nauczysz się korzystać z trójstanowego operatora porównania (<=>)
- Odkryjesz nowoczesne metody formatowania tekstu z użyciem std::format
- Poznasz mechanizmy std::ranges, które pozwalają wygodniej przetwarzać sekwencje danych
- Dowiesz się, czym są koncepty (concepts) i jak dzięki nim tworzyć bezpieczniejsze, bardziej przejrzyste szablony
- Nauczysz się budować moduły — jako nowoczesny sposób organizowania dużych projektów
- Poznasz korutyny, jedną z najciekawszych i najbardziej nowatorskich możliwości C++20

Jeżeli znasz wcześniejsze standardy C++, ta książka pokaże Ci, jak ogromny krok naprzód wykonał współczesny C++, i pomoże w pełni wykorzystać jego nowe możliwości.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-3984-4

9 788328 939844

Cena: 79,00 zł