

TOM

5

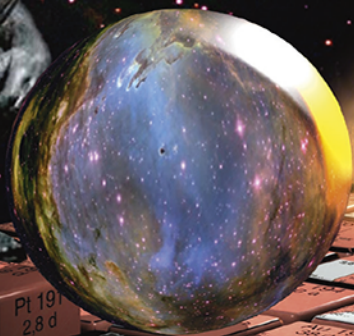
Jerzy Grębosz

Opus magnum C++

ŁATWY PODRĘCZNIK

Wyprawa w głąb

C++20



Helion

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki i opracowanie graficzne książki: Jerzy Grębosz

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

Wydanie pierwsze

ISBN: 978-83-289-3984-4

Copyright © Jerzy Grębosz 2026

Printed in Poland

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>

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

0	Proszę tego nie czytać!.....	1
0.1	Wyruszamy na kolejną wyprawę!	1
1	Nowe atrybuty w C++20	3
1.1	Atrybuty <code>[[likely]]</code> i <code>[[unlikely]]</code>	3
1.2	Atrybut <code>[[nodiscard]]</code> z tekstem	5
1.3	Atrybut <code>[[no_unique_address]]</code> do optymalizacji pamięci obiektu	6
1.4	Ćwiczenia	8
2	Trójstanowy operator porównania <code><=></code>	9
2.1	Operator porównania <code><=></code> dla typów całkowitych	9
2.2	Operator <code><=></code> wobec typów zmiennoprzecinkowych	10
2.3	Trzeci możliwy typ rezultatu <code><=></code> , czyli porównanie słabe (<i>weak_ordering</i>)	13
2.4	Operator <code><=></code> można zadeklarować jako <code>= default</code>	17
2.5	Składniki typu zmiennoprzecinkowego <code>a = default</code>	19
2.6	Konwersje między kategoriami porównań	22
2.7	Jak strumieniem <code>cout</code> wypisać rezultat porównania słownie	23
2.8	Ćwiczenia	24
3	<code>using enum</code> – co za ulga w C++20!	26
3.1	Zwykłe <code>enum</code> (tzw. plain <code>enum</code>)	26
3.2	<code>enum class</code> – czyli zakresowy <code>enum</code>	27
3.3	C++20 pomaga uprościć zapis (<code>using enum</code>)	28
3.4	Ćwiczenia	31
4	Nowoczesne formatowanie tekstu – poznaj <code>std::format</code>	33
4.1	Funkcja <code>std::format</code>	33
4.2	Znacznik formatujący	34
4.3	Pole: <code>id</code> – indeks	35
4.4	<i>Dwukropek</i> – rozpoczyna część z określeniami sposobu formatowania	35
4.5	Pole formatowania: <i>szerokość</i>	35
4.5.1	Szerokość sterowana argumentem (dynamicznie)	36
4.6	Pole: <i>wyrównanie</i> (<i>justowanie</i>)	37
4.7	Pole: <i>wypełnienie</i> (<i>char</i>)	37
4.8	Pole: <i>znak</i> (liczby dodatniej lub ujemnej)	38

4.9	Pole: <i>prefix</i> (podstawa systemu liczenia)	38
4.10	Pole: wiodące <i>zera</i>	38
4.11	Pole: <i>specyfikacja typu wypisywania</i>	39
4.12	Pole: <i>precyzja</i> , czyli dokładność	42
4.12.1	Na zakończenie – dynamiczne podawanie dokładności	44
4.13	Przykłady złożonych znaczników	44
4.14	Ćwiczenia	45
5	<i>Span</i>	47
5.1	Geneza, czyli skąd ten pomysł?	47
5.2	<i>std::span</i>	51
5.3	Dalsze korzyści z klasy <i>span<T></i>	54
5.4	Ciekawe funkcje składowe klasy <i>std::span</i>	57
5.5	Ćwiczenia	58
6	<i>constexpr</i>, <i>constexpr</i> – kiedy warto liczyć w compile-time	60
6.1	<i>constexpr</i> – funkcje wykonywane tylko w trakcie kompilacji	60
6.1.1	Tablica liczb obliczana w czasie kompilacji programu	64
6.2	<i>constexpr</i> – inicjalizacja obiektów statycznych w czasie kompilacji	67
6.2.1	<i>constexpr</i> – bezpieczna inicjalizacja zmiennych globalnych	69
6.2.2	Dobre praktyki: jak definiować i inicjalizować zmienne globalne	71
6.3	Kilka przypomnień i ostrzeżeń	72
6.4	Ćwiczenia – <i>constexpr</i> i <i>constexpr</i>	72
7	<i>template<auto></i> w nowej odsłonie	74
7.1	Co to jest NTTP?	74
7.2	Przykłady różnych NTTP	75
7.2.1	Stała zmiennoprzecinkowa jako NTTP w szablonie funkcji	75
7.2.2	Stała zmiennoprzecinkowa jako NTTP w szablonie klasy	76
7.2.3	Stała będąca obiektem klasy literalnej	77
7.2.4	Stała będąca referencją do C-stringu	78
7.2.5	Stała będąca pojemnikiem typu <i>std::array</i>	80
7.2.6	Stały obiekt struktury (konfiguracja algorytmu)	80
7.2.7	Stała NTTP będąca funkcją lambda z przydomkiem <i>constexpr</i>	82
7.3	Podsumowanie	83
7.4	Ćwiczenia	84
8	Nowości w C++20 dotyczące lambda	88
8.1	Przypomnienie	88
8.2	C++20 → lambda z listą parametrów szablonu	89
8.3	Ćwiczenia	91
9	Koncepty w C++20 – jak szepnąć kompilatorowi, czego oczekujesz	93
9.1	Wymogi i koncepty	93
9.2	Gdzie można narzucać wymagania (koncepty)?	96
9.3	Porozmawiajmy bliżej o słowie kluczowym <i>requires</i>	97
9.4	Z czego może składać się zbiór wymogów?	98
9.5	Koncepty z biblioteki standardowej (koncepty standardowe)	99
9.5.1	Podstawowe koncepty (warunki) dotyczące typów i ich relacji	100
9.5.2	Koncepty związane z porównywaniem wartości	102

9.5.3	Koncepty, które sprawdzają zachowania obiektów.....	102
9.5.4	Koncepty dotyczące elementów, które można wywołać jak funkcje	103
9.5.5	Koncept dotyczący wyniku wywołania.....	104
9.5.6	Typ relacyjny	105
9.5.7	Koncepty relacyjne (porównania i porządek).....	107
9.6	Kwestia stylu	111
9.7	Łączenie konceptów: koniunkcja i alternatywa.....	112
9.8	Koncepty użyte w wyrażeniach lambda	114
9.8.1	Parametry o typie wnioskowanym (<i>auto</i>) – czyli lambda bez jawnego <...>	114
9.8.2	Lambda z parametrami szablonu – czyli lambda mająca <...>.....	116
9.9	Co tak naprawdę zmieniają koncepty w C++20?	117
9.10	Ćwiczenia	117

10 *range* – zakres sekwencji elementów119

10.1	Przypomnienie o algorytmie <i>std::sort</i>	119
10.2	Wkraczamy w krainę <i>ranges</i>	121
10.3	Widoki na panoramę górską, czyli <i>views</i>	124
10.4	Zestawienie ciekawych funkcji-adapterów <i>views::</i>	131
10.5	Adapter <i>views::iota</i> generujący ciąg liczb i widok	133
10.5.1	Jak to wszystko właściwie działa „pod maską”?	135
10.6	Adapter <i>views::elements</i>	135
10.7	Adaptory <i>views::values</i> , <i>views::keys</i>	136
10.8	Adaptory strukturalne <i>views::join</i> , <i>views::split</i> i <i>views::common</i>	138
10.8.1	Adapter <i>views::common</i>	141
10.9	Od strumienia do sekwencji: wczytywanie danych z użyciem <i>ranges</i>	142
10.9.1	Idea: wczytywanie danych za pomocą biblioteki <i>ranges</i>	142
10.9.2	Widok interpretujący strumień znaków ze stringu	143
10.9.3	Widok interpretujący strumień znaków z pliku.....	145
10.9.4	Widok interpretujący strumień z klawiatury jako dane typu <i>int</i>	147
10.10	Pojedynek strumień versus widok	148
10.11	Ćwiczenia	150

11 Nowe narzędzie: moduły153

11.1	Pliki nagłówkowe – rozwiązanie z czasów przed C++20.....	153
11.2	Moduły – lepsza wersja nagłówków.....	154
11.3	Jak zbudowany jest plik modułu?.....	157
11.4	Jak to działa – co robi kompilator, gdy spotyka moduł?	163
11.5	Definicja klasy w module	163
11.5.1	Mała, krótka klasa w module	163
11.5.2	Moduł z bardziej rozbudowaną klasą.....	166
11.5.3	Kiedy użyć którego sposobu?	172
11.6	Organizacja większych modułów	173
11.6.1	Moduł z partycjami	174
11.6.2	Blok <i>export</i> i blok <i>export namespace</i>	179
11.6.3	Sekcja prywatna w deklaracji modułu.....	182
11.7	Importowanie biblioteki standardowej	187
11.8	Jak udało mi się uruchomić moduły w praktyce?.....	187
11.8.1	Konfiguracja krok po kroku w środowisku Qt Creator	188
11.9	Ćwiczenia	191

12 Korutyny.....193

12.1	Skąd ten pomysł?.....	193
12.2	Korutyna – pierwsze starcie.....	197

12.3	Korutyna – jak to wszystko może działać?.....	200
12.4	Co naprawdę robi kompilator przy wywołaniu korutyny?	201
12.4.1	Ad krok 2. – Jak naprawdę zbudowany jest <i>Tsterownik_pracy</i> ?.....	204
12.4.2	Ad krok 3. – Co w programie robi <i>initial_suspend()</i> ?.....	207
12.4.3	Ad krok 5. – Instrukcja <i>co_yield</i> to tak naprawdę jest <i>co_await</i>	208
12.4.4	Ad krok 7. – Co robi funkcja <i>final_suspend()</i> ?	210
12.5	Korutyna, która sukcesywnie dostarcza liczb (generator liczb).....	212
12.6	Dwukierunkowa korutyna – rozmowa zamiast monologu	215
12.7	Korutyny, które przekazują sobie sterowanie	221
12.7.1	Przykład programu z dwoma korutinami, które przekazują sobie sterowanie	223
12.8	Gdy korutyna mówi: „Obudź mnie później”	231
12.9	Najprostsza pętla zdarzeń	235
12.10	Refleksje o mechanice kończenia działania korutyny	240
12.11	Praktyczne użycie – od mechaniki do narzędzia	241
12.11.1	ETAP 1. Sterownik, który zwraca wartość	242
12.11.2	ETAP 2. Wersja z pilnowaniem własności ramki	244
12.11.3	ETAP 3. Gdy jedna korutyna czeka na drugą	247
12.11.4	ETAP 4. Wyjątki i propagacja błędów w korutinach	254
12.12	Pojedynek korutyny versus wątki.....	258
12.13	Symulacja komunikacji klient – serwer	261
12.13.1	To jest paragraf dla zaawansowanych, znających operacje sieciowe	269
12.14	Na zakończenie.....	272
12.15	Ćwiczenia	273

13 Posłowie.....274

Skorowidz.....276



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. Pojawia 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 wymyślałem 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`, `dźwięk`. 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 dziwnie: `dźwięk`. 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 `źmudna_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 `źmudną_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 zadania

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/univ\_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;                  ❸
    // przypisanie nowej wartości do zmiennej globalnej
    nglobal = fun_kompil(4);                                    ❹
    cout << " nglobal = " << nglobal << endl;                    ❺
    // wydruk: nglobal = 16

    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.

- 6 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 $100 + 25$).

- 7 Oczywiście można wywołania funkcji z przydomkiem `constexpr` użyć w dowolnym wyrażeniu, tutaj na przykład korzystamy z niego przy przypisywaniu do obiektu typu `double`.

Podsumujmy:

Słowo kluczowe `constexpr` 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 `constexpr` była bardzo prosta. Nieprzypadkowo.

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

Nie każda funkcja nadaje się do tego, by mieć przydomek `constexpr`

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

W funkcjach `constexpr` 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 `constexpr`,
- ❖ 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/constexpr`),
- ❖ mieć ciała, które nie zwracają wartości (`void` może wystąpić tylko w nielicznych metaprogramistycznych zastosowaniach).

Destruktor nie może być oznaczony specyfikatorem `constexpr`.

Jak funkcje `constexpr` 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:

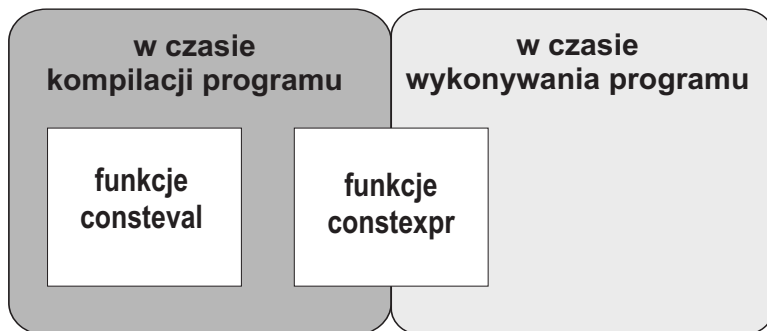
- ❖ Funkcje `constexpr` kompilator mógł wywołać w trakcie kompilacji, jeśli było to możliwe, czyli jeśli argumenty wywołania były stałymi `constexpr`. Jeśli jednak argumenty wywołania nie były stałymi `constexpr` (tylko jakimiś zmiennymi), to kompilator uznawał, że mają być wywołane dopiero w czasie 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.

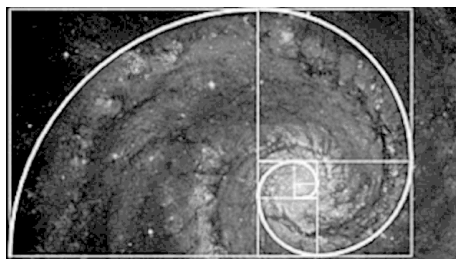
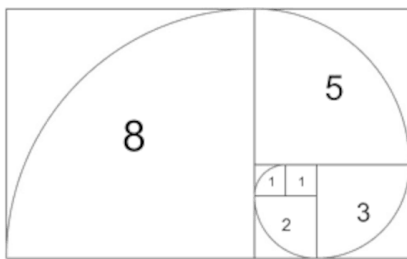
6.1.1 Tablica liczb obliczana w czasie kompilacji programu

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 `constexpr` 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
{
    unsigned long long element[70];           // pierwsze 70 wyrazów ciągu
```

1

```

};
//*****
constexpr Tstruktura_ciaq_Fibonacciego  zbuduj_tablice_fib()           ❷
{
    Tstruktura_ciaq_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_ciaq_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_ciaq_Fibonacci`. Jej jedynym składnikiem jest tablica 70 elementów typu `unsigned 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_ciaq_Fibonacci`.

- ❸ W ciele funkcji definiujemy lokalny obiekt `t`, który jest klasy `Tstruktura_ciaq_Fibonacci`. 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_ciaq_Fibonacci` 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_ciaq_Fibonacci`.

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.

Podkreślimy: 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*, *constexpr* 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 funkcję 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()`? Nieokreślona 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
```

1

plik *a.cpp*:



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

2

3

plik *main.cpp*:



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



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 nagłówki
#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 `constexpr` tam, gdzie wynik musi być znany w czasie kompilacji.
- ❖ Używaj `constinit` do globalnych zmiennych (unikniesz sytuacji związanych z nieokreślonymi wartościami wynikłymi 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	<code>constexpr</code>	<code>constexpr</code>	<code>constinit</code>
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 – *constexpr* i *constinit*

- I Co oznacza specyfikator `constexpr` przy funkcji?
- II Podaj przykład funkcji `constexpr` i jej poprawnego użycia.
- III Co się stanie, jeśli spróbujesz wywołać funkcję `constexpr` 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 <i>constinit</i> .
VII	Jakie są korzyści stosowania <i>consteval</i> i <i>constinit</i> w porównaniu z <i>constexpr</i> ?
VIII	Co stanie się, jeśli <i>constinit</i> zostanie użyte, ale inicjalizacja zmiennej nie będzie mogła odbyć się w <i>compile-time</i> ?
IX	Porównaj <i>consteval</i> i <i>constexpr</i> . Podaj jedną zasadniczą różnicę.
X	Napisz funkcję <i>consteval</i> , która oblicza sześćcian liczby, i użyj jej do inicjalizacji zmiennej z <i>constinit</i> .



Skorowidz

!

#include compare 12
#pragma once 154

A

adapter
 > fun. pracująca na widokach 127
adaptery
 > diagnostyczne i pomocnicze 132
 > filtrujące 131
 > generujące 132
 > ograniczające 132
 > porządkujące 132
 > przekształcające 131
 > strukturalne 132
algorytm std::sort 119-120
algorytmy a span 56
alias enum class 31
alternatywa konceptów 112-113
animacja - dławienie się 197
architektura asynchroniczna 239
atrybut
 > likely 3-4
 > no_unique_address 6-7
 > nodiscard 5
 > unlikely 3-4
automat stanów 197
await_ready() 219
await_resume() 219
await_suspend() 219
await_transform 219
awaiter 209
 > przekierowania 251
 > przełączający 222
 > wznowienia 252

B

BMI - Binary Module Interface 163
biblioteka
 > programowania asynchronicznego 231
 > ranges 119

 > standardowa - jej importowanie 187
 > standardowa pojemników STL 54
blok
 > export (w module) 179
 > export namespace (w module) 179
bębenkowy licznik 186

C

ciąg Fibonacciego 64
ciągłe sekwencje 47
co_await - z argumentem 238
co_await read(socket) 269
co_await write(socket) 269
co_return 242
co_yield 199, 203
 > porównanie z co_await 208
common_reference_with 100
common_with 100
compare (#include) 12
consteval 60-73
 > a constexpr 62
 > a zmienne globalne 67
 > na czym może operować 67
 > obostrzenia 62
constexpr
 > a constinit 68
constinit 67-71
 > a constexpr 68
 > do czego używamy 69
convertible_to 100
copyable 102-103
coroutine_handle promise_type > 206
coroutine_handle< promise_type > 228
część implementacyjna modułu 171
część interfejsowa modułu 156
część przedmodułowa 171

D

deklaracja
 > modułu 158, 160
 > przynależności modułu 171
derived_from 100

dokładność podawana dynamicznie 44
 done() 240
 dwukropek w polu formatowania 35
 dynamiczne podawanie dokładności 44
 dyrektywy preprocesora
 > w module 157
 dyspozytor 221, 227, 240, 269
 dławienie się animacji 197

E

eksportowane nazwy modułu 156
 enum class 27
 > a użycie aliasu 31
 enum plain 26
 equality_comparable 102
 equality_comparable_with 102
 equivalence_relation 107
 equivalent 11, 13
 export module - instrukcja 157-158
 extern constinit 71

F

Fibonacciego ciąg 64
 fabryka widoków 145
 final_suspend() 201, 210
 floating_point (koncept) 100
 format 33
 from_promise(*this) 206
 funkcja format a manipulatory 40
 funkcja natychmiastowa 60

G

generator liczb 212-214
 get_return_object 202, 206
 globalny fragment modułu 157, 160
 greater 11

H

harmonogram 239

I

immediate function 60
 implementacja 166
 implementacyjna
 > część modułu 171
 import partycji modułu 177
 importowanie biblioteki standardowej 187
 indeks
 > pole formatowania 35
 initial_suspend() 201-202, 207
 instrukcja module 157

integral (koncept) 100
 interfejs modułu 182
 interfejs znaczeniowy 117
 invocable 103
 iteracja 121
 ixr 166

J

jednoprzebiegowy zakres 142
 jednoznaczna własność ramki 244
 justowanie 37

K

kategorie znaczeniowe 95
 klasa
 > pusta 6
 > w module 163-172
 > zagnieżdżenie 227
 klient – serwer komunikacja 261-271
 komunikacja klient – serwer 261-271
 koncept 93
 > a requires 96
 > jako jawny wrunek logiczny szabl. 97
 > przy typie param. szablonu 97
 > przy wnioskowanym arg. fun. 96
 koncepty
 > alternatywa 112-113
 > do porównywania wartości 102
 > dot. ob. wywoływalnych 103
 > dot. typów i ich relacji 100
 > dot. wyniku wywołania 104
 > koniunkcja 112-113
 > relacyjne (prównania i porządku) 107
 > sprawdzające zach. obiektów 102
 > standardowe 99-110
 > w wyr. lambda 114-116
 koniunkcja konceptów 112-113
 konstruktor przenoszący 246
 konwersje między kategoriami porównań 22
 korutyna
 > sterownik pracy 204
 > a stos 202
 > a wątki 258-260
 > czeka na drugą 247
 > czeka na sygnał 231
 > dwukierunkowa komunikacja 215-220
 > generator liczb 212-214
 > jej sterownik 202
 > kończenie pracy 240
 > kto może być jej właścicielem 241
 > kto zarządza jej życiem 240

- › odbiera przesłaną wartość 220
 - › przekazuje sterowanie innej korutynie 221-230
 - › punkty zawieszenia 220
 - › ramka 200
 - › uchwyt 228
 - › warunek wznowienia 239
 - › wyjątki 254
- koszt przełączania 259

L

lambda

- › a koncepty 114-116
- › koncepty w param. szablonu 116
- › ograniczenia w C++14 89
- › uogólniona 88
- › z jawną listą param. szablonu 89
- › z listą parametrów szablonu 89-90
- › z ustalonymi typami 88

lenistwo 128-130

less 11

liczba jednoczesnych zadań 260

licznik bębnowy 186

likely - atrybut 3

lokalny zakres

- › a using enum 30

M

makrodefinicje

- › w module 157
 - › z wnętrza modułu 161
- manipulatory a funkcja format 40
- maszyna stanów 197

module

- › export 157
- › instrukcja 157

moduł

- › a makrodefinicje 157, 161
- › a using namespace std 158
- › część interfejsowa 156
- › część przedmodułowa 171
- › deklaracja przynależności 171
- › fragment prywatny 182
- › globalny fragment 157
- › jego deklaracja 158, 160
- › jego interfejs 182
- › jego nazwa 158
- › plik implementacyjny 166
- › sekcja globalna 157, 160
- › sekcja prywatna 182
- › w nim definicja klasy 163-172
- › wewnątrz 156, 158

- › z partycjami 174
 - › zewnętrzny wizerunek 182
- moduły 153-192
- movable 102

N

natychmiastowa funkcja 60

nazwa kwalifikowana

- › elementów listy wyliczeniowej 31

nazwany warunek kompilacyjny 94

nodiscard - atrybut 5

noop_coroutine() 228

ntp - NTTP 74

- › f. lambda z constexpr 82
- › jako referencja do C-stringu 78
- › nowości w C++20 75
- › stała zmiennoprzecinkowa
 - w szablonie funkcji 75
 - w szablonie klasy 76
- › stały obiekt kl. literalnej 77
- › stały pojemnik std::array 80
- › stałym obiektem struktury 80

O

One Definition Rule – ODR 154

obiekt

- › automatyczny 67
- › statyczny 67

odraczanie, rodzaj oszczędności 130

operacje asynchroniczne 269

operacje sieciowe 268

operator

- › co_await 215

operator statku kosmicznego 9

operatory porównania w pakiecie 19

P

Perl 9

partial_ordering 11, 23

- › equivalent 11
- › greater 11-12
- › less 11-12
- › unordered 11-12

partycje modułu 174

pieczenie herbatników 124

plain enum 26

plik

- › *.ixx 166
- › deklarujący moduł 166
- › implementacyjny modułu *.ixx 166

pole formatowania 34

- › indeks 35
- › precyzja 42-43
- › precyzja a sposób wyświetlania liczb 43
- › prefix (podst. systemu liczenia) 38
- › specyfikacja typu wypisywania 39-41
- › szerokość 35-36
- › wiodące zera 38
- › wypełnienie 37
- › wyrównanie (justowanie) 37
- › znak liczby 38

policy_type 200

port USB - jako span 54

poziome przekazanie sterowania korutyn 223

pragma once 154

precyzja czyli dokładność 42-43

predicate 105

prefix (podstawa systemu liczenia) 38

preprocesora dyrektywy

- › w module 157

private module fragment 183

promise_type 200

protokół oczekiwania 219

prywatny fragment modułu 182

przekazywanie sterowania 230

przenoszący operator przypisania 246

przeplatanie operacji READ i WRITE 268

przełączanie trójstanowego op. 15

przełączanie wykonywania 259

przynależność modułu 171

punkty zawieszenia 220

pusta klasa 6

pętla zdarzeń 235-239

R

ramka korutyny 200-201

- › a jednoznaczna własność 244

ranges - biblioteka 119

ranges::filter 127

ranges::find 124

ranges::istream_view 144

ranges::sort 123

ranges::transform 124

regular 103

regular_invocable 103

relacyjny typ 105

relation 107

requires 95

- › a koncept 96
- › użyte do opisu zestawu wymogów 98
- › użyte w dekl. szablonu 97

resume() 198, 203

rethrow_exception() 254

return_void() 201

rezultat porównania słownie 23

równoległość 260

S

śruby - przykład 14

same_as 100

scheduler 221, 239-240

sekcja

- › globalny fragment modułu 160
- › prywatna w module 182
- › wewnątrz modułu 160

sekcja globalna modułu 157

sekwencja elementów 121

semiregular 102

serwer sieciowy 261

signed_integral (koncept) 100

spaceship operator 9

span 51-53

- › ciekawe funkcje składowe 57
- › dynamiczny 52
- › korzyści 54-56
- › najważniejsza korzyść 54
- › pojemniki, z którymi NIE działa 55
- › pojemniki, z którymi działa 55
- › składniki tej klasy 52
- › statyczny 52
- › współpraca z STL 54

span::back() 58

span::data() 58

span::empty() 58

span::first 58

span::front() 58

span::last 58

span::size() 58

span::size_bytes() 58

span::subspan 58

spanning pointer 51

specyfikacja typu wypisywania 39-41

statku kosmicznego - operator 9

stałe odwieczne 63

std::common_reference_with 100

std::common_with 100

std::convertible_to 100

std::copyable 102-103

std::derived_from 100

std::equality_comparable 102

std::equality_comparable_with 102

std::equivalence_relation 107

std::floating_point 100

std::integral 100

std::invocable 103

std::movable 102

std::predicate 105

std::regular 103
 std::regular_invocable 103
 std::relation 107
 std::same_as 100
 std::semiregular 102
 std::signed_integral 100
 std::sort 119-120
 std::strict_weak_order 107
 std::totally_ordered 102
 std::totally_ordered_with 102
 std::unsigned_integral 100
 sterownik korutyny 202

- › jako szablon klasy 242

 sterownik pracy korutyny 204
 strażnik nagłówka 154
 strict_weak_order 107
 string formatujący 33
 strong_ordering 10, 23

- › equal 10
- › greater 10
- › jako typ rezultatu 10
- › less 10

 suspend_always 207
 suspend_never 207
 system asynchroniczny 239
 szablonowa lambda 89
 szerokość - pole formatowania 35-36
 szerokość sterowana argumentem (dynamicznie) 36

T

tablica

- › jako range 124
- › obliczana w trakcie kompilacji 64

 template i auto 74
 totally_ordered 102
 totally_ordered_with 102
 trójstanowy operator porównania 9-25

- › a typy zmiennoprzecinkowe 10-12
- › dla typów całkowitych 9
- › jako =default 17-18
- › przeładowanie 15

 typ o jawnej strukturze 84
 typ relacyjny 105

U

USB port jako span 54
 uchwyt korutyny 228
 unhandled_exception() 240, 254
 union 7
 unlikely - atrybut 4
 unordered 11

unsigned_integral (koncept) 100
 using enum 28-30

- › a instrukcja switch 29
- › kiedy nie używać 31

 using namespace std

- › w module 158

V

view 124-130
 views::all 132
 views::common 132, 138-141
 views::counted 132
 views::drop 132
 views::drop_while 131
 views::elements 131, 135
 views::empty 132
 views::filter 131
 views::iota 132-134
 views::istream 144
 views::join 132, 138-141
 views::key 132
 views::keys 136-137
 views::reverse 129, 132
 views::split 132, 138-141
 views::take 132
 views::take_while 129, 131
 views::transform 128, 131
 views::values 132, 136-137

W

warunek wznowienia korutyny 239
 weak_ordering 13-16, 23

- › equivalent 16
- › greater 13
- › less 13

 widok (view) 128
 wiodące zera 38
 wnętrze modułu 156, 160
 wyjątki w korutinach 254
 wymogi

- › dot. budowy typu 99
- › dot. wyrażeń 98
- › dot. wyrażeń z okr. wyniku 98
- › proste 98
- › zagnieżdżone 99
- › zbiór 98
- › złożone 98

 wypełnienie 37
 wypisywanie liczb zmiennoprzecinkowych 40
 wyrównanie 37
 właściciel ramki korutyny 241
 wątki a korutyny 258-260

Y

yield 199

yield_value 201, 208

Z

zagnieżdżenie klasy 227

zakres

› sekwencji elementów 121

zakresowy enum 27

zakresowy for a span 56

zasada jednej definicji (ODR) 154, 163

zbiór wymogów 98

zdarzeń petla 235-239

zewnątrzny wizerunek (interfejs) modułu 182

znacznik formatujący 34

› indeks 34

znaczniki stringu formatującego 33

znak (liczby dodatniej lub ujemnej) 38

zniszczenie korutyn 268

zużycie pamięci 259

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 

Wyprawa w głąb C++20

Poznaj 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 cennego 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