1111111111111111111111111  
tytuł

2222222222222222222222222  
Agenda

3333333333333333333333333  
Skąd nazwa?  
Albo z gry stworzonej w BASICu Star Trek z 1971. Skrin. Albo z TIE fighter z serii filmów Star Wars.

44444444444444444444444444  
Co to three-way?  
Znak większe niż, mniejsze niż, lub znak równe. To jest w skrócie to co zwraca operator spaceship. Więcej szczegółów wkrótce.

5555555555555555555555555555  
Użycie (integer)  
Zwraca wynik i w zależności od tego wypisany zostaje odpowiedni komunikat. Następuje przyrównanie do zera. Ify tylko w celu pokazania działania.

6666666666666666666666666666  
Użycie(string)  
Podobny przykład  
Leksykograficzny,, to na przykład jak słowa są posegregowane w słowniku.  
Przy tworzeniu C++20 zauważono, że spaceship zawsze będzie wykonywał porównania leksykograficzne. Mogłoby to doprowadzić do niewystarczającego, wygenerowanego kodu. Dotyczy to szczególnie operatora równości.   
Przykładem niech będzie porównanie dwóch stringów. Stringu „funkcja” i stringu „fun”.  
W celu optymalizacji najpierw porównywana jest długość obu stringów, a jeżeli są równe, porównywane są element po eemencie, aż do znalezienia różnicy.  
Aby spełnić takie warunki nasz intwrapper mógłby być napisany w taki sposób.

77777777777777777777777777777  
Użycie(vector)  
wskazuje na pierwszy element i dlatego mamy taki wynik porównania

8888888888888888888888888888888888  
Jak było wcześniej  
Teraz pokażę porównanie typu przed i po. String porównywany ze stringiem, lub z elementami typu const char. W c++17 takie.  
Funkcja friend, żeby po zadeklarowaniu poza klasą miała dostęp do prywatnych elementów klasy.  
Różne typy chcę ze sobą porównywać, dlatego tyle przeciążeń.   
CIString – case insensitive(nie ważne czy duża czy mała litera)  
const char\* tworzy wskaźnik na string

99999999999999999999999999999999  
Jak jest teraz  
W c++20 wystarczą do tego 4 funkcje. Chciałem tu tylko zobrazować. Już tłumaczę dokładniej, co oznaczają zapisy w tym przykładzie.

10101010101010101010101010101010101010  
Mamy już jakieś doświadczenie z porównaniami typu three-way z języka C z memcmp/strcmp i z c++ z porównywania stringów operatorem compare(). Te operatory zwracały wartość typu int. Dodatnią gdy pierwszy argument był większy od drugiego, równą 0 gdy były równe i ujemną, gdy pierwszy argument był mniejszy od drugiego.  
Spaceship nie zwraca wartości typu int. Zamiast tego otrzymujemy obiekt, który jest jedną z kategorii porównań, której wartość opisuje wynik porównania. Są takie 3 podstawowe kategorie:  
-strong ordering – uporządkowanie w którym równość skutkuje również zastępowalnością. To znaczy jeśli (a ⬄ b) ==strong\_ordering::equal to również f(a)==f(b). W strong\_ordering nie wliczają się funkcje, które np zwracają adres argumentów, albo wielkość wektora. Liczą się tylko że tak powiem istotne właściwości danego typu. Taką Wartością wektora są elementy które zawiera, a nie ich adres. Strong ordering zwraca wartości strong\_ordering less, equal, i greater.  
-weak ordering – porządek w którym równość definiuje tylko równoważność klas. Dobry jest przykład z poprzedniego slajdu, porównanie stringów bez względu na wielkość liter. Weak\_ordering::equal nie oznacza że elementy są dokładnie równe, dlatego mówimy bardziej o równoważności(equal i equivalance jak tłumaczyć)  
-partial\_ordering – tutaj oprócz wartości greater, less i equal mamy również unordered. Taki rezultat otrzymujemy po przyrównywaniu do niezdefiniowanych wartości(NaN).

Strong ordering jest najpopularniejszy i w zasadzie domyślnym wyborem przy porównaniach. 1⬄2 da less, a 2⬄1 da greater.

Silniejsze porównania są konwertowalne do słabszych(głównie strong ordering do weak ordering). I zachowują rodzaj porównania po konwersji(strong::equal do weak::equivalent).

Wyniki przyrównywane są do zera. To nie jest int, to jest dosłownie zero.

Dolne, co lepiej wybrać. Przypuśćmy, że a jest równe 1, a b jest równe 2. 1 spaceship 2 zwróci strong ordering::less, co jest mniejsza od zera. Całość wyrażenia zwróci nam true, co jest właściwym wynikiem.  
Spaceship jest bardziej szczegółowy, bo może zwrócić 4 różne wyniki, podczas gdy wyrażenie boolowskie tylko dwa: true i false. W przypadku niezdefiniowanych wartości zwróci nam partial\_ordered::unordered zgodnie ze stanem faktycznym.

11 11 11 11 11 11 11 11 11 11 11 11 11 11 11  
Przed c++20  
Może się zdarzyć kod taki, jak poniżej. Całkiem standardowe przeciążanie operatorów.  
Niestety, ten kod się nie kompiluje. Error C3615, constexpr function 'is\_lt' cannot result in a constant expression  
constant expression to wyrażenia, które mogą wykonują się w czasie kompilacji

12 12 12 1212 12 12 12 12 12 12 122 12 12  
Przed c++20  
No tak, zapomnieliśmy o constexpr przy funkcji porównania. Dodajemy więc constexpr przy każdym operatorze porównania. Gdybyśmy chcieli dodać oprócz tego is\_it sprawdzającego równość, dodać jeszcze is\_gt(greater), trzeba by znowu do każdego przeciążenia noexcept. Po prostu żmudny proces.

13 13 1313 13 131 31 31 3 13 13 13 1 31 3 13 13 1 31 3 13 13 13  
W c++20  
I tutaj wkracza operator spaceship. Tak wyglądałby ten sam kod napisany w C++20.  
Pierwszą różnicą jest zaincludowany <compare>. Ten nagłówek zaopatruje kompilator we wszystkie potrzebne spaceshipowi kategorie porównań. W tym przykładzie w miejsce auto wydedukowany zostanie strong\_ordering.  
  
Oprócz tego usunięto 5 linijek. Nie musimy również niczego definiować. Is-it zadziała pomimo tego constexpr, nawet jeśli nie wskazaliśmy tego przy operatorze spaceship.

14 141 41 41 1 41 4 14 14 1 4 14 14 1 41 4 1 41 4 1   
Pomocny kompilator  
W c++20 zaprezentowano również koncepcję przepisanych wyrażeń. Dobrymi kandydatami do wytłumaczenia tego jest właśnie operator spaceship i operator==.  
Przy przeładowaniu operatorów kompilator wyszukuje najlepszych andydatów do użycia na podstawie typów zmiennych.  
Dla naszego wyrażenia a mniejsze niż b, może być wyszukiwane dla typu a albo operator⬄, albo zakres namespace operator⬄ który akceptuje taki typ. Kompilator znajduje, że faktycznie typ a znajduje się w IntWrapper::operator⬄ i przepisuje a< b jak a⬄b < 0 . To przepisanie jest później wykorzystywane, jako kandydat do normalnych przeładowań.  
Dlaczego to działa i jest poprawne? Na tym właśnie polega operator spaceship. Porównanie three-way zwraca nie wyniki w postaci binarnej, tylko bardziej porządek(ordering), którym można wyrazić każdą zależność. Szybki przykład 1 ⬄2 zwraca strong\_ordering::less. Jest to informacja nie tylko o tym, że 1 jest różne od 2, ale także , że jest mniejsze. Jest to najdokładniejszy opis relacji między jedynką i dwójką.

Gdybyśmy pisali w C++17 taki przy is\_gt\_42 wyskoczył by taki błąd.  
Trzeba by wtedy dopisać dodatkowe funkcje friend do intwrappera.  
Natomiast wykorzystując poprzednią, dłuższą definicję c++17 w kompilatorze c++20 kod zadziała. Bo w c++20 kopilator spróbuje użyć przepisanego wyrażenia 42⬄a <0 i zorientuje się, że ne ma konwersji z intwrapper do int. Poszukując kompilator spróbuje 0 < a ⬄ 42, zobaczy że jest konwersja z int do intwrapper i użyje włąśnie tego kandydata.

15 15 15 15 15 15 15 15 1 51 51 5 15 15 115 1 515 15  
Pomocny kompilator  
Wygenerowany przez kompilator operator spaceship radzi sobie nie tylko z pojedynczymi członkami klasy. Radzi sobie z całymi zestawami porównań dla różnych typów.Przekształca macierze składające się z różnych typów w listę pod obiektów i porównuje je rekurencyjnie.

16 16 16 16 16 16 16 16 1 61 6 16 16 1 61 6   
Kolejne zmiany  
Dobra, kompilator może przepisywać operatory. A co gdybyśmy chcieli przechytrzyć kompilator, tak jak tu operatorem <?  
Do walki o zostanie użytym przez kompilator stanie tych trzech kandydatów.  
W C++17 doszło by do niejednoznaczności. W C++20 kompilator ma możliwość wyboru najbardziej logicznego przeciążenia. Pojawiła się dodatkowa zasada, że kompilator ma wybrać przeciążenie, które nie zostało przepisane, lub zsyntezowane. W związku z tym kompilator będzie używał tego dopisanego przeciążenia.

17 17 171 717 17 17 177 17 1717   
Dopasowanie argumentów  
Rozważmy taki przykład.  
Zarówno struktura A i struktura B ma operator ==. W związku z tym z porównania a ==b i b ==a oczekujemy takiego wyniku(przekreślone).  
Tak byłoby w c++17, ale w c++20 wynik będzie taki(nieprzzekreślone).  
Powodem jest to, że operator porównania z A jest po pierwsze lepszym dopasowaniem, ponieważ B szuka obiektu typu const. Po drugie nie wymaga od kompilatora rzutowanie obiektu który nie jest typu const do const.  
dzięki komendzie –show-all-implicit-cast możemy to lepiej zobaczyć.  
O ile samo rzutowanie nie jest niczym złym, o tyle z tego powodu staje się gorszym dopasowaniem przy przeciążeniu operatorów w c++20. Spotkałem się też z określeniem spaceship jako consistent comparisons. I to chyba bardzo tu pasuje  
Dodam tylko, że ten kod jest zły. Nie powinno być tak, że tylko jeden operator porównania jest typu const. Albo obydwa, albo żaden. Wykrzaczyłby się przy użyciu z obiektem const.