JEGYZŐKÖNYV

Mértékegységek és prefixumok követése

Korszerű számítástechnikai módszerek a fizikában 1

2021. május 7.

Mérést végezte: Farkas Bendegúz

A project célja

Egy olyan osztály kreálása, amely a hét SI alapmennyiséget, valamint ezek kombinációját tárolja és ezeket figyelembe véve tudjon alapműveleteket végezni(tehát a $3g + 4cm$-nek nincs értelme). Ezenkívűl a program tárolja az ismertebb SI prefixumokat és kövesse a mennyiségekre. Az *Xcode* nevű programban dolgoztam és a programot $c++$ nyelven irtam meg.

Bevezetés

Az egyik első dolog, amit minden fizikus hallgató megtanul, az az, hogy miként lehet megfelelően összehangolni a fizikai dimenziót az egyenlet mindkét oldalán. Ez egy nagyon egyszerű módszer a számítási hibák ellenőrzésére. Ha egy hosszúság megegyezik egy területtel, vagy egy tömegmennyiség hirtelen néhány másodpercet vesz igénybe, akkor elég nagy az esély arra, hogy hiba történt valahol a levezetések közben.

A C++ sablonok alkalmazása lehetővé teszi a fizikai mennyiségek dimenziójának nyomon követését. A karakterláncok , valamint az *std* :: *ratio* sablon segítségével a különböző mértékegységek könnyen használhatók és kezelhetőek.

Ez az ötlet vagy probléma nem új, és a megvalósítások is már jó ideje léteznek. A legtöbb egyszerű megvalósítás azonban csak a fizikai egység dimenzióinak egész számát követi nyomon. Eleinte úgy tűnik, hogy ez több mint elég, mivel olyasmi, mint a „négyzetgyök méter”, nincs fizikai ábrázolásban, és inkább a számítás hibájára utal. Bár ez igaz, mindazonáltal fontos a törtes dimenziók nyomon követése is. Például az euklideszi távolság kiszámításába bármely dimenzió két pontja között, amely magában foglalja a négyzetek összegének négyzetgyökének kiszámítását. A négyzetgyök művelet lényegében felezi a mennyiség fizikai dimenzióját, oly módon, hogy általában racionális kitevőt ad. Például, ha egy négyzet gyökét vesszük, akkor a kapott mennyiség formálisan a hosszúság egyik felének dimenziója. Bár ez nyilvánvalóan hiba, a sima egész számú kitevők használata könnyen elrejtheti ezt a nyilvánvaló hibát a felfelé kerekítés (1 kitevő, ismét hosszúság megadása) vagy lefelé kerekítés (0 kitevő, skalárt jelentő) miatt.

A fizikai mennyiségeket használó valós számításokhoz és alkalmazásokhoz ezért a racionális exponensek beillesztése alapvető követelmény. A racionális számok sablonok felhasználásával történő ábrázolása viszont nem egyszerű feladat. Míg egy racionális számot könnyen két egész számra bonthatunk - amelyekre léteznek beépített sablon-paraméterek -, a racionális számok ábrázolásakor a redundancia kérdésével is foglalkozni kell. A jól ismert Boost könyvtár valójában arra törekszik, hogy ilyen fordítási idejű racionális számokat biztosítson. Véleményem

szerint a teljes Boost egységrendszer meglehetősen bonyolult, és nagymértékben az optimalizálóra támaszkodik. Ezt kikerülve találtam egy könnyebb megoldást, amely a C++ nyelv két nagyszerű szolgáltatásán alapul: a string literálok, amelyek olyan operátorok, amelyek bizonyos beépített adattípusokat „kiterjesztenek" és az std::ratio sablon, amely fordítási idejű racionális számokat ad a megfelelő módon.

A C++11 std::ratio STL sablon

Az egész számokkal ellátott alapvető számtan mind szép és jó, amíg az ember ragaszkodik az elemi matematikához. A valamivel kifinomultabb szintre lépő valós alkalmazásokban gyorsan el kell jutni az egész kitevők tartományán kívülre. Amint a bevezetőben említettem, nem lesz elég egész kitevőkre korlátozni a dolgokat, ha a négyzetgyökök és hasonló műveletek jönnek a képbe. A négyzetgyök például egyszerűen felezi a dimenziós kitevőket, azaz

$$\sqrt{unit} = m^{\frac{a}{2}} \cdot kg^{\frac{b}{2}} \cdot s^{\frac{c}{2}} \cdot A^{\frac{d}{2}} \cdot K^{\frac{e}{2}} \cdot mol^{\frac{f}{2}} \cdot cd^{\frac{g}{2}},$$

oly módon, hogy a megfelelő dimenzióelemzéshez képesnek kell lenni az ilyen törtek nyomon követésére. Elképzelhető egy olyan rendszer, ahol minden egész számot két egész szám helyettesít, például X és Y a dimenzió számlálója és nevezője, amelyek a $(\frac{X}{Y})$ hányadost képviselik. Ez az egyszerű megközelítés azonban nem veszi figyelembe a reprezentáció redundanciáját

$$\frac{X \cdot k}{Y \cdot k} = \frac{X}{Y}, \qquad ahol\ k \in Z$$

,amely természetesen racionális számokban fordul elő, azaz a számláló és a nevező közös többszöröse.

A legnagyobb közös osztó kiszámításának problémája a racionális szám egyedi csökkentett reprezentációjának meghatározása érdekében az euklideszi algoritmus alkalmazásához vezet, amely rekurzív módon kiszámítja a két szám kölcsönös osztódásának fennmaradó részét. Ennek a számításnak a fordítás idején történő megkönnyítése érdekében ezt a rekurziót egy végtelenül beágyazott C ++ sablon konstrukcióval kell ábrázolni.

Szerencsére a C ++ 11 frissítés egy ilyen fordítási idejű racionális számsablont adott a standard könyvtárhoz, ami az std::ratio néven található meg. Ez két szempontból előnyös: Először is sok fordítóspecifikus kódtól szabadít meg(nem kell a boost könyvtár), másrészt az STL-sablon fordító általi hatékony optimalizálásának esélyei jobbak a felhasználó által definiált struktúrához képest mint ez - elvégre az STL megvalósítást ugyanazok adják, akik fejlesztik a fordítót. Az ésszerű számot ábrázoló std::ratio <számláló, nevező> sablonnal együtt $(q = \frac{a}{b} \in Q)$ számos sablon „függvény", például $ratio_{add}, ratio_{subtract}, ratio_{multiply}$ és a $ratio_{divide}$,

valamint az összehasonlító sablon „operátorokat” határoztak meg az alapvető racionális számtani számok biztosításához.

C++11 String literálok

Ezen a ponton fizikai mennyiségtípusokat használhatunk, és fordítási idő dimenzióval ellenőrzött kódot írhatunk, amely lehetővé teszi a racionális kitevőket. Azonban továbbra is fennáll a különbözõ mértékegységek azonos típusú fizikai mennyiségre való átkonvertálásának kérdése. Például különféle hosszúságegységek léteznek, amelyek mind konceptuálisan egyenértékűek, és állandó tényezők kölcsönösen kapcsolódnak egymáshoz. Bár minden bizonnyal jó megközelítés az SI-alapmennyiségek alapegységként történő használata, mégis hasznos lehet egy egyszerű módszer a helyes mennyiségtípus példázására és az egység-átalakítás automatikus végrehajtására.

Emellé jönnek a string literálok, amelyeket a C++11 frissítéssel vezettek be. Ez lehetővé teszi új utótagok definiálását bizonyos elemi adattípusok számára az új karakterlánc operátor, azaz az *opertor*”” használatával. Például, míg a 3.4 double változót jelent, a 3.4f utótag egy float adattípust hoz létre. A string literal operátor bevezetése a megfelelő paraméter adattípushoz lehetővé teszi az ilyen utótagok egyedi meghatározását. Az itt érdekelt két adattípus long double és unsigned long long int , mindkettő az adott típus „legnagyobb” változata

Tekintettel az RQuantity sablon osztályra a dimenzióellenőrzéshez és a származtatott mennyiségtípusokhoz, például a hossz vagy a tömeg, ez lehetővé teszi az egyszerű karakterlánc operátorok definiálását a közös mértékegységekhez.

Ez lehetővé teszi olyan közvetlen hozzárendelések végrehajtását, mint például Hosszúság Hossz = $12,43km$; vagy Tömeg valamennyiTömeg = $7kg + 0,23kg$;, ahol a teljes sablontípus példányosítást és egységkonvertálást a karakterlánc operátor végzi.A karakterlánc-operátornál azért van szükség unsigned long long int és a long double adattípusra is, mert ha valaki olyat akar írni, hogy: $7kg + 0,23kg$, akkor automatikusan átalakít egy egész számot is. Ez különösen egyszerűvé teszi a korábban tárgyalt dimenzióanalízis alkalmazását.

Az "constexpr" kulcsszó

A másik elég hasznos és gyakran használt kifejezés a constexpr kulcsszó, amelyet a C++11 nyelvű frissítésben vezettek be. A fő gondolat a programok teljesítményének javítása azáltal, hogy a futási idő helyett a fordítás idején végezi a számításokat. (hasonlóan a sablon metaprogramozásához) A constexpr megadja, hogy egy objektum vagy egy függvény értéke összeállítási időpontban értékelhető-

e, és a kifejezés felhasználható más állandó kifejezésekben. Ez azt jelenti, hogy az így deklarált változók és függvények olyan helyeken használhatók, ahol csak fordítási idejű állandó kifejezések megengedettek - legalábbis addig, amíg megfelelő állandó függvény argumentumokat használnak. A constexpr kulcsszó használata mind az RQuantity sablon számtani operátorainál, mind az egység string literáloknál arra ösztönzi a fordítót, hogy ahol szükséges, az állandókat tartalmazó számításokat fordítási időben hajtsa végre. Ezenkívül lehetővé teszi a dimenzióval rendelkező mennyiségek állandó inicializálóként való megadását a karakterláncok használatával, pl.: const Hosszúság myLen $= 7,2km - 987m$.

Diszkusszió

A program koránt sem tökélestes és lehetne rajta mit még csiszolni. Például jelenleg csak két szám közötti művelet lehetséges, de ezt ki lehetne terjesztni több változósra. Az átkonvertálást is meg lehetet volna elegánsabban oldani, hogy a program válassza ki a legoptimálisabb prefixumot és ne nekünk kelljen megadni. A program végén lévő void függvényt is jobban kellett volna automatizálni. Sajnos önhibámból kicsúsztam az időből és nem tudtam tovább fejleszteni a programot de ezektől eltekintve a projectet sikeresnek mondanám, sikerült pár új és szerintem hasznos dologgal találkoznom. Habár a dimenzió analizís nem túl bonyolult módszer, szerintem leprogramozni érdekes és tanulságos dolog volt.

Hivatkozások

- Scott Meyers, Ph.D. Software Development Consultant - Dimensional Analysis in C++
- David Abrahams and Aleksey Gurtovoy - C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth).