# C++ unit library stage opdracht

## Inhoudsopgave

[C++ unit library stage opdracht 1](#_Toc193790870)

[Inhoudsopgave 1](#_Toc193790871)

[Probleem 1](#_Toc193790872)

[Verwachte oplossing 2](#_Toc193790873)

[Eisen 2](#_Toc193790874)

[Onderdelen 5](#_Toc193790875)

[Binnen scope van de opdracht 5](#_Toc193790876)

[Onderzoek naar geschikte library 5](#_Toc193790877)

[Aanpassen examplecode 5](#_Toc193790878)

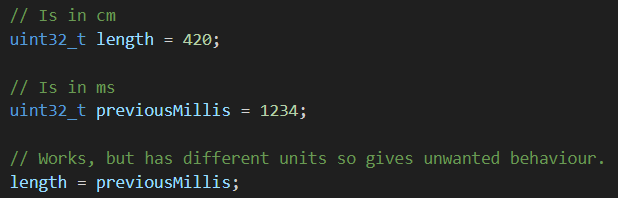
[Analyseren highest common denominator C++ versie 5](#_Toc193790879)

[Extra opdracht die buiten de scope van de C++ units library valt 5](#_Toc193790880)

[Originele tekst/ opdrachtbeschrijving: 5](#_Toc193790881)

## Probleem

Binnen Inspiro worden bij veel opdrachten verschillende eenheden gebruikt in de code. Er zijn een aantal manieren om onderscheid te maken tussen deze verschillende eenheden.   
Helaas zijn veel van deze methodes niet veilig en kunnen ze voor problemen zorgen. Een bekend voorbeeld hiervan is de NASA [Mars Climate Orbiter](https://en.wikipedia.org/wiki/Mars_Climate_Orbiter" \l "Cause_of_failure). Hieronder volgt nog een voorbeeld:



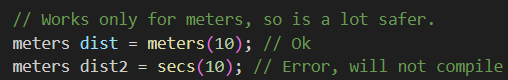
Zoals in dit voorbeeld te zien is, kan een tijd in ms gebonden worden aan een lengte in cm, wat niet zou mogen en voor problemen kan zorgen. Helaas is hier geen controle op en moet er naar het commentaar gekeken worden om te zien wat de variabele inhoudt.

In een groot project is dit onhandig, dus worden er soms zelf types gedefinieerd. Dit is een optie die leesbaarder is, maar het kost wel extra projecttijd om te implementeren en vaak wordt er alsnog niet gecontroleerd of er verkeerde conversies plaatsvinden.

Kortom, het probleem is dat er nog geen generieke oplossing is die de type-safety van eenheden verhoogt.

## Verwachte oplossing

De verwachte oplossing hiervoor is een library (een software bibliotheek) die deze eenheden bevat en controleert op verkeerde declaraties of andere functies. Hieronder is een voorbeeld:



Zoals in het voorbeeld te zien is kunnen er geen secondes gebonden worden aan een meter variabele.  
Met een library die deze eenheden bevat en functionaliteit biedt om te converteren tussen bijvoorbeeld meter en cm zou de type-safety nog extra vergroot worden. Mits de library hier op controleert, zoals in het voorbeeld hierboven.

# Eisen

Hieronder volgt een lijst van eisen die gesteld zijn aan de library. Knockout criteria houdt in dat wanneer hier niet aan voldaan wordt, de library sowieso niet geschikt is.

* **Knockout criteria**
  + *Software licentie*De software moet bruikbaar zijn voor Inspiro, dus bijvoorbeeld een MIT of Apache 2.0 licentie mag gebruikt worden, maar GPL-3.0 niet. Zie [deze link](https://inspirobv.sharepoint.com/:w:/s/Inspiro-KnowledgeCenter/ERpLmVwQ8AZBvhTe5uD9UNEBwlRSvSCoOuQVlAAWNNCZgw?email=lars.van.duijnhoven%40inspiro.nl&e=ubA9Qc&isSPOFile=1) bij H3.2 voor een overzicht van welke licenties gebruikt mogen worden.  
    Licenties die bijvoorbeeld forceren dat het volledige programma open source gemaakt moet worden zijn verboden om te gebruiken.
  + *C++ versie*  
    De hoogste versie die de library mag hebben hangt af van de uitkomst van het onderzoek naar de highest common denominator.  
    Een library die niet ondersteunt wordt door de huidige gebruikte microcontrollers is namelijk nutteloos.
  + *Wanneer worden de types gecontroleerd? (Dimensional analysis)*  
    Dimensional analysis is het controleren van types, dus of een meter (eenheid) aan een temperatuur (dimensie) gebonden kan worden. In dit voorbeld wordt  
    De foutmeldingen van verkeerd typegebruik moeten compiletime gecontroleerd en weergeven worden, dus runtime is niet geschikt. Dit komt doordat compiletime sneller te debuggen is en dus meer tijd scheelt dan runtime. Bij runtime moet het programma continu gebuild en gerund worden, wat allemaal extra tijd kost.
  + *Heap geheugen gebruik Statische allocatie?? Andere beschrijving*  
    Vanwege het gebruik op embedded apparaten mag er geen gebruik gemaakt worden van de heap. Dynamische allocatie via de heap is namelijk veel foutgevoeliger, wat niet wenselijk is. Alles moet op de stack, dus met statische allocatie. Hierbij kan er ook gemeten worden hoeveel van de stack gebruikt wordt door de library.
* **Andere criteria**
  + *Uitbreidbaarheid*  
    Is de library makkelijk uit te breiden door Inspiro? Met uitbreiden wordt bedoeld extra eenheden toevoegen. Als de eenheden zelf toegevoegd kunnen worden is dat beter voor het grootschalig gebruik van deze library. Als er namelijk een specifieke eenheid in een toekomstig project komt die nog niet in deze library zit, kan deze toegevoegd worden.
  + *Instelbaarheid*  
    Zijn de variabelen makkelijk in te stellen, zoals wisselen tussen float en integer types, bijvoorbeeld zodat afstand een float is en temperatuur een integer.  
    Hoe meer je dit zelf in handen hebt, hoe beter.
  + *Kwaliteit van documentatie*Deze is redelijk vanzelfsprekend. De kwaliteit van de documentatie moet wel goed zijn, want je hebt er alsnog niks aan als het niks uitlegt.
  + *Hoeveelheid documentatie*Verder is de hoeveelheid documentatie ook belangrijk. Als 1 functionaliteit extreem goed uitgelegd is, maar alle andere functionaliteiten niet, heb je vrij weinig aan de documentatie.
  + *Conversies*Zijn conversies mogelijk? Zo ja, zijn ze automatisch of moeten ze handmatig aangeroepen worden? Dit scheelt berekeningen en mogelijke rekenfouten, waardoor de code robuuster zou worden. Zie een voorbeeld van automatische conversies hieronder:

A black background with white text

AI-generated content may be incorrect.

* + *Externe dependencies*Het liefst zo min mogelijk externe dependencies, omdat deze naderhand aangepast kunnen worden. Deze aanpassingen kunnen ervoor zorgen dat de werking aangepast moet worden, waardoor bestaande code niet meer zal werken.
  + *CPPcheck resultaat*Als het door CPPcheck komt is dit ook een teken dat het goede code is.  
    Mocht dit niet zo zijn, dan is het al duidelijk dat de code waarschijnlijk minder goed geschreven is. Dit is een soort meetstaaf om snel de code te garanderen van kwaliteit.
  + *Unittest coverage*  
    Hoeveel unittests zijn er geschreven? Dit kan in 5 gradaties, namelijk: niks, weinig, normaal, veel, erg veel.  
    Dit is belangrijk, omdat dit ook als documentatie kan functioneren door voorbeelden te geven van hoe de library gebruikt kan worden.
  + *Ingebouwde eenheden*  
    Hoe meer eenheden de library standaard ingebouwd heeft die door Inspiro worden gebruikt, des te beter. Dit scheelt namelijk tijd om de eenheden er zelf in te zetten en verhoogt de bruikbaarheid van de library.
  + *C++ code guidelines*  
    Hoe goed voldoet de huidige librarycode aan de Inspiro code guidelines? Als er bijvoorbeeld veel verboden functionaliteiten gebruikt worden zoals macro’s, is dit niet bevorderlijk.
  + *Leesbaarheid van code*De code moet leesbaar en begrijpbaar zijn voor de programmeurs. Slecht leesbare code wordt een knockout, omdat dit niet gewenst is en zelfs extra tijd kan kosten i.p.v. tijd besparen.
  + *Moeilijkheid Cmake integratie*Hoe makkelijker/ sneller een library geïntegreerd kan worden met Cmake, hoe beter. Dit scheelt namelijk kostbare tijd.
  + *Afwijkende eenheden*Met afwijkende eenheden worden bijvoorbeeld decibel en temperatuur bedoeld.   
    Deze wijken af van standaard ratio’s of hebben een ander onderdeel waardoor ze problemen veroorzaken. Als dit ondersteunt wordt, betekent dit dat veel fouten voorkomen kunnen worden en dus is dit gewenst.
  + *Leesbaarheid van code*De code moet (snel) leesbaar zijn. Als de code moeizaam tot niet leesbaar is, is dit een knock-out. Dan kan het meer tijd kosten dan dat de library bespaard en willen de programmeurs waarschijnlijk niet overgaan.

# Onderdelen

## Binnen scope van de opdracht

### Onderzoek naar geschikte library

Om te beginnen moet er onderzoek gedaan worden naar welke libraries er zijn en welke geschikt zijn voor Inspiro. Hierbij is het onderzoeksdoel om erachter te komen welke library we kunnen gebruiken, waarvan een zelfgemaakte library ook een mogelijkheid is.

### Aanpassen examplecode

Als er een library gekozen is moet de library geïntegreerd worden in de CSDD examples van de STM32, ESP32 en Nordic nRF.  
De STM32 heeft hier prioriteit en moet dus als eerst af.  
Hierna komt de ESP32 en daarna de Nordic nRF.

### Analyseren highest common denominator C++ versie

Dit is een sub-opdracht die binnen de scope valt. Hier kan aan gewerkt worden als er tijd over is of wanneer er vanuit andere onderdelen behoefte is om dit uit te zoeken.Nadat de examples aangepast zijn moet de highest denominator voor de C++ versie gevonden worden de hierboven genoemde 3 microcontrollers. In andere woorden, wat is de hoogste C++ versie die zowel door de STM32, ESP32 en Nordic nRF ondersteund wordt.

## Extra opdracht die buiten de scope van de C++ units library valt

**Uitbreiden ISL-Cmake voor sanitizers**uitbreiden isl-cmake voor sanitizers (memory, stack, thread, undefined behavior, etc)

# Originele tekst/ opdrachtbeschrijving:

* selecteren en inrichting van een C++ units library om units als voltage, ohm, temperatuur etc type-safe uit te kunnen drukken
  + eisen: geen heap, compile time checking, ondersteuning C++11 en C++14, toe kunnen voegen eigen units
  + hint: <https://github.com/martinmoene/PhysUnits-CT-Cpp11> ziet er veel belovend uit
* aanpassen csdd examples voor esp32 & nordic nrf gebaseerd op stm32
* uitbreiden isl-cmake voor sanitizers (memory, stack, thread, undefined behavior, etc)
* analyseren highest common denominator voor C++ versie voor stm32, esp32, nrf