Templates und Compilezeitpolymorphie

Agenda

- Warum #define vermeiden
- Funktionstemplates
- Klassentemplates
- Spezialisierungen
- Type traits
- Template Metaprogramming
- Variadic Templates
- Constexpr

Warum #define vermeiden

- Defines sind immer global
- Haben keinen Typ
- Sind beim Debuggen oft nicht einsehbar
- Können Namenskonflikte erzeugen / nicht mit einem namespace belegbar

Grundlagen Templates

- Templates ermöglichen einen Codeabschnitt mit verschiedenen Datentypen aufzurufen
- Datentypen müssen zur Compilezeit bekannt sein
- Alle Operationen, die für die Typen verwendet werden, müssen verfügbar sein
- Syntax ist *template<class/typename> Name*
  - Class: Template ist eine Klasse
  - Typename: Template ist ein beliebiger Typ
- Aufruf kann explizit oder implizit sein
  - Explizit: *<typ> Funktion(params)*
  - Implizit: *Funktion(params)*
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-templates

Grundlagen Templates

- Funktionen müssen leider im Header implementiert werden
  - https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl
- Bei Fehlern in Templatecode bekommt man nur selten hilfreiche Meldungen

Funktionstemplates

- Templates können für spezielle Funktionen angegeben werden
- Funktion kann somit mit verschiedenen Datentypen aufgerufen werden
- Beispiel:
  - Std::find() kann mit allen Iteratortypen benutzt werden
- https://godbolt.org/g/HcVpEB

Klassen templates

- Templates können auch für ganze Klassen angeben werden um bestimmte Membervariablen zu setzten
- Template kann hier nur explizit angegeben werden
- z.b. Containerklassen
- https://godbolt.org/g/BiYdz2

Spezialisierungen

- Für verschiedene Datentypen können mehrere Funktionen erzeugt werden
  - Template Typ muss hierfür bei der Funktion angegeben werden
- Ähnlich zu Funktionsüberladungen
- https://godbolt.org/g/8cNDmH

Type traits

- Seit C++11
- Mittels Type traits lassen sich zur Kompilierzeit Eigenschaften von Datentypen herausfinden.
  - http://en.cppreference.com/w/cpp/header/type_traits
- Mit *std::enable_if* können Template Spezialisierungen anhand von Datentyp Eigenschaften zugewiesen werden
  - Leider komplexe, unlesbare Syntax
  - *Std::enable_if_t<type_trait<T>::value> = 0*
  - Extrem hilfreich um Implizite Casts zu vermeiden.
- *https://godbolt.org/g/ApVNJ3*

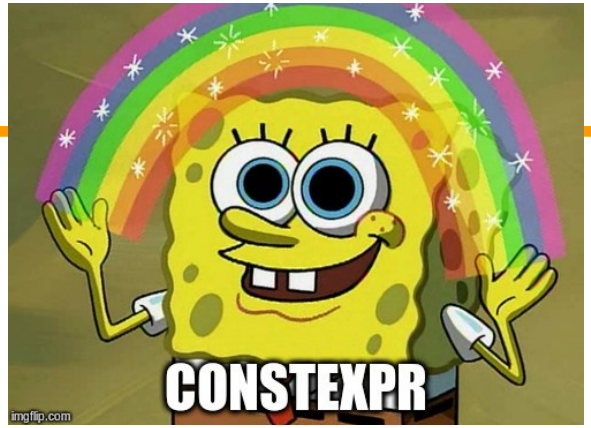
Template Metaprogramming

- Möglichkeit Programme zur Compilezeit auszuführen
  - Vorteil: Laufzeitberechnungen nicht mehr notwendig (z.B. für Hash-Werte)
  - Nachteil: Deutlich Längere Compilierzeit
- Werte müssen static const sein, da sie sonst zur Compilezeit nicht ausgewertet werden können
- Übergabewerte müssen als Templatewert angegeben werden
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#tmeta-t…ogramming-tmp
- https://godbolt.org/g/yWcGaz

Variadic Templates

- Seit C++11
- Wie bei Variadic können beliebig viele Übergabeparameter angegeben werden
- Bei Templatefunktionen braucht man 2 Spezialisierungen
  - Der Standartfall mit *template<typename T, typename... Targs>*
  - Der Spezialfall mit einem Parameter *template<typename T>*
- Funktion muss rekursiv aufgelöst werden, für die verschiedenen Übergabeparameter
- Da es zu Compilezeit gelöst wird, gibt es keinen Overhead zur Laufzeit
- https://godbolt.org/g/n4drWm

Constexpr



- Seit C++11
- Weist den Compiler an, die Variable, Funktion etc. zur Compilezeit zu lösen
- Kann mit Templates kombiniert werden, muss aber nicht
  - C++11 erlaubte nur einzeilige constexpr Funktionen
  - C++14 hebt das auf
- Extrem mächtig, da ganze Klassenaufrufe, zur Compilezeit verfügbar sind
  - Algorithmen voraussichtlich ab C++20
- https://godbolt.org/g/VCnZtm

Anwendungsbeispiel für Embedded Devices

- Aneinanderhängen von Byte kompatiblen Datentypen in einem neuen Wert
  - https://godbolt.org/g/ks4gED
- Register lesen
  - https://godbolt.org/g/VsFgji
- Quelle: https://github.com/offa/stm32-eth

Nächstes Mal:
Nebenläufigkeit und Parallelität