**50**

HBO ICT

TI

*C++ Reader*

C++ Programming & Software Engineering 1

**TICT-V2CPSE1-15**



Studiejaar 2017-2018

|  |  |
| --- | --- |
|  |  |
| Cursuseigenaar | Wouter van Ooijen |
| **Auteur(s)** | Wouter van Ooijen |
| **Datum** | 2017-01-30 |
| Versie | 2.0 |
|  | |

© Institute for ICT, Hogeschool Uterecht, 2017

Contents

[2. Inleiding 3](#_Toc491691431)

[3. Het bouwen van een applicatie 4](#_Toc491691432)

[3.1. Stappen 4](#_Toc491691433)

[3.2. Make 7](#_Toc491691434)

[4. Templates 11](#_Toc491691435)

[4.1. Functie template 11](#_Toc491691436)

[4.2. Klasse template 12](#_Toc491691437)

[4.3. std::array<> 14](#_Toc491691438)

[4.4. Specializatie 14](#_Toc491691439)

[4.5. Proxy pattern 15](#_Toc491691440)

[5. constexpr 17](#_Toc491691441)

[6. RAII 23](#_Toc491691442)

[7. Lempel-Ziv compressie 25](#_Toc491691443)

[8. Template metaprogramming 28](#_Toc491691444)

[8.1. Recursive specialization 28](#_Toc491691445)

[8.2. loop 29](#_Toc491691446)

[8.3. SFINAE 31](#_Toc491691447)

# Inleiding

Deze reader hoort bij de cursus V2CPSE1, die wordt gegeven in het eerste kwartaal van het tweede studiejaar Technische Informatica. Het overkoepelende thema van dit kwartaal is het realiseren van kleine embedded real-time systemen. De andere vakken van dit kwartaal richting zich op het modelleren van dergelijke systemen en het in projectvorm realiseren van een dergelijk systeem.

In de CPSE1 cursus worden een aantal Programming en/of Software Engineering onderwerpen behandeld die horen bij het werken met een klein embedded systeem. Er wordt van uitgegaan dat de student in vorige cursussen een ervaring heeft opgedaan met C, C++, en het gebruik van eenvoudige hardware.

Net als in vorige cursus wordt er gewerkt op een Windows PC. De applicaties worden gebouwd voor en uitgevoerd op twee platforms: de Windows PC zelf en de Arduino Due. In beide gevallen wordt gebruik gemaakt van de CodeLite IDE, de GCC compiler, en de bmptk/hwlib library. (Er is beperkte ondersteuning voor werken op Linux systemen, dit zal meer zelfredzaamheid van de student vergen dan gebruik van Windows.) De gebruikte software tools en de Arduino Due sluiten aan bij de themaopdracht. De installatie en configuratie van deze tools is beschreven in de reader van de cursus V1OOPC.

Er is in deze reader gekozen voor Nederlands voor de lopende tekst (om het voor de student zo leesbaar mogelijk te houden), en voor Engels voor de code voorbeelden (om de student toch te laten wennen aan Engels, wat de taal is waarin hij zal moeten leren te werken). Dit leidt er soms toe dat twee woorden worden gerbuikt voor hetzelfde ding (bv. bal versus ball, overerving versus inheritance).

Voor zover relevant wordt gebruik gemaakt van de mogelijkheden die de nieuwe C++14 standaard biedt. Let op dat veel voorbeelden die je op het internet en in boeken vindt maken nog geen gebruik van (alle) C++14 features, waardoor ze er op details wat anders uit kunnen zien.

Naast deze reader wordt er voor het assembler deel gebruik gemaakt van een aparte (engelstalige) reader voor dat onderwerp.

# Het bouwen van een applicatie



|  |
| --- |
| behandelde onderwerpen |
| * Preprocessor, compiler, linker * Declaratie, definitie * Gebruik van header files * make tool, makefile * Melodie, RTTTL file formaat |

## Stappen

Bij het bouwen van een applicatie vanuit C of C++ code worden een aantal stappen doorlopen:

1. Preprocessing
2. Compilatie
3. Linken

De preprocessor is een toolje dat run voordat de eigenlijke compiler start. De preprocessor neemt een (enkele!) source file en verwerkt de zgn. preprocessor directives (die beginnen met #) en produceert een nieuwe text file[[1]](#footnote-1) waarin die directives zijn ‘uitgewerkt’, bv.:

* #include wordt vervangen door de inhoud van de ge-include file
* #define verdwijnt, maar op alle plekken waar het gedefinieerde woord voorkomt wordt dat vevangen door zijn definitie
* \_\_FILE\_\_ en \_\_LINE\_\_ worden vervangen door de ‘huidige’ file naam en regenummer
* Alle commentaar verdwijnt
* #ifdef … #endif verdwijnen, en als de conditie niet waar was verdwijnt het stuk ertussen ook

Het is belangrijk te realiseren dat de preprocessor niets weet van C of C++ (de zelfde preprocessor wordt soms ook gebruikt voor andere talen). Het is een tool die text als input krijgt en text als output produceert.

|  |  |  |
| --- | --- | --- |
| **#include “y.h”**  **int x;**  **// one year later**  **x = y + year** | **// file y.h**  **extern int y;**  **#define year 12** | **extern int y;**  **int x;**  **x = y + 12;** |
| C source files en de output van de preprocessor | | |

Nadat de preprocessor een file onderhanden heeft genomen wordt zijn uitvoer aan de compiler gegeven. Die produceert (als er geen fatale fouten zijn opgetreden) een object file. Een object file bevat de code en data voor de dingen die in de source file zijn gedefinieerd. Een object file kan die dingen voor zichzelf houden, of beschikbaar stellen naar buiten. Omgekeerd kan een object file ook dingen van buiten zichzelf nodig hebben.

|  |  |
| --- | --- |
| **extern int y; // definitie van y**  **int x; // declarative van x**  **x = y + 12; // gebruik van x en y** | **NEED**  **y**  **DSEG**  **x 4**  **CSEG**  **load y**  **add 12**  **store x** |
| C source met bijbehorende object file | |

Ter herinnering: een declaratie is een beschrijving van de interface van een ding. Een definitie is dat, en ook nog het ding zelf. In C en C++ horen declaraties in header files (.h of .hpp) te staan, en definities in source files (.c of .cpp).

|  |  |  |
| --- | --- | --- |
|  | Declaratie | Definitie |
| type / class | Geen onderscheid: bestaan niet in de executable | |
| Functie | int next( int x ); | int next( int x ){  return x + 1;  } |
| Object (variabele) | extern int x; | int x; |

De preprocessor en de compiler verwerken iedere source file apart tot een object file. Als alle source files van een applicatie aldus gecompileerd zijn komt de linker aan de beurt die de object files combineert tot een executable. Hierbij worden vrijwel altijd ook object files meegenomen die in libraries zitten. Dit zijn bundels van (voor-gecompileerde) object files.[[2]](#footnote-2)

De taal van de linker is redelijk eenvoudig: hij pakt alle object files, en probeert voor ieder ding dat door een object file wordt vereist een bijbehorend ding (met de zelfde naam) te vinden dat door een andere object file beschikbaar wordt gesteld. Als hij telkens precies 1 zo’n beschikbaar ding vindt dan is alles goed. Als hij voor een vereist ding geen kandidaat vind, of meer dan 1, dan geeft de linker een error (undefined symbol, of multiple defined symbol).

|  |
| --- |
|  |
| De linker combineert object files tot een executable, die geen externe referenties meer heeft. |

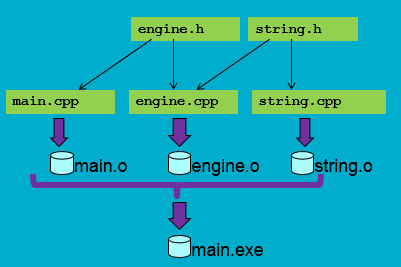
Het kan de linker niets kan schelen wat voor soort dingen hij aan elkaar knoopt: als de namen maar met elkaar kloppen is de linker tevreden. Dit kan tot problemen leiden als een dingen in de ene source file gedeclareerd is maar in een andere source file anders gedefinieerd.

|  |  |
| --- | --- |
| **char \*m**  **int main(){**  **f();**  **std::cout << m;**  **}** | **int m;**  **void f(){**  **m = 15;**  **std::cout << m;**  **}** |
| Conflicterende declaratie en definite van m | |

De compiler zal dit probleem niet ontdekken, want die werkt op 1 source file en ziet dus de andere (op dat moment niet). De linker zal dit probleem ook niet ontdekken, want die kijkt alleen naar de namen van dingen. De main file heeft een m nodig, en de andere file levert een m, dus wat de linker betreft is alles OK. Pas bij het runnen gaat het mis: in de code van de main wordt datagene wat in het geheugen op plek m staat een pointer is naar een reeks characters. Maar op de plek m staat een integer die de waarde 15 heeft gekregen, dus de main zal geheugenadres 15 (en de daaropvolgende plekken) lezen en de bytes die daar staan als characters afdrukken. Althans, dat is wat de code zal proberen te doen: in veel gevallen zal adres 15 niet beschikbaar zijn voor de applicatie. Op en desktop systeem zal het Operating System dit afvangen en er dus een (fatal) error optreden. Wat er op een micro-controller zal gebeuren is lastig te voorspellen. In veel gevallen zal je applicatie onvoorspelbaar gedrag vertonen, of lijken te crashen.

Om dit soort problemen te voorkomen is het de goede gewoonte om voor dingen die in meerdere files gebruikt worden altijd een declaratie in een header file te zetten, en de definitie in een .c of .cpp file. De declaratie in de header zorgt ervoor dat iedere code file (die de header gebruikt dmv. een #include) dezelfde declaratie gebruikt. Dat de definitie in een code file (dus niet in een header file) zit zorgt ervoor dat de linker maar 1 object file krijgt die het ding bevat. Als de definitie in de header file zou staan dan zou het ding in iedere object file terecht komen die de header file gebruikt, dus dat zou een linker fout opleveren.

Met deze conventie gaat alles dus goed, mits alle code files gecompileerd zijn met dezelfde versie van de header file(s). Als je de header file wijzigt en dan een deel van de code files compileert kan het nog steeds mis gaan omdat de niet overniew gecompileerde code files nog de oude versie van de header file gebruikten. Een regel die dit voorkomt is dat een object file altijd nieuwer moet zijn dan alle files (code file en header files) waar hij van afhankelijk is. In het volgende plaatje moet engine.cpp dus overniew gecompileerd worden als engine.h, string.h of engine.c veranderd is.



## Make

Make is een tool dat je kan gebruiken om te zorgen dat de juiste files die opnieuw gecompileerd worden. Daarvoor moet je een makescript file maken waarin je voor iedere te produceren file opgeeft van welke files hij afhankleijk is, en wat het commando is om die file te produceren. De onderstaande makefile geeft dit aan voor de files van het vorige plaatje. Iedere ‘rule’ (productieregel) bestaat uit twee delen: eerst de te produceren file, een dubbele punt, en de files waarvan hij afhankelijk is. Daarna een of meer regels die beginnen met een tab (dit moet echt een tab zijn, geen spaties) met commando’s die uitgevoerd worden om de te produceren file te maken.

|  |
| --- |
| **main.exe : main.o engine.o string.o**  **gcc main.o engine.o string.o –o main.exe**  **main.o : main.cpp engine.h**  **gcc –c main.cpp –o main.o**  **engine.o : engine.cpp engine.h string.h**  **gcc –c engine.cpp –o engine.o**  **string.o : string.cpp string.h**  **gcc –c string.cpp –o string.o** |

De volgorde van de productieregels is niet van belang. Alle productieregels worden door de make tool gelezen en opgelsagen. Vervolgens kijkt make wat er gemaakt moet worden (de target). Default is dat de eerste productieregel, maar je kan het ook op de command line opgeven. Make checkt of alle files ware de target direct of indirect van afhankelijk is up to date zijn. Zo niet, dan worden die files ‘bottom up’ aangemaakt door de bijbehorende opdrachten uit te voeren.

Naast ‘echte’ targets kent make ook ‘phony’ targets. Dat zijn geen files, dus als je make aangeeft dat een phony target moet worden aangemaakt dan worden de bijbehorende acties altijd uitgevoerd (want de file is er niet, dus make doet alsof die altijd moet worden aangemaakt). Dit wordt bv. vaak gebruikt om het commando ‘make clean’ te kunnen gebruiken. ‘clean’ is een phony target, en de bijbehorende actie is het verwijderen van alle geprodueerde files. In onderstaan voorbeeld is ook ‘build’ een phony target, dat veroorzaakt dat de file main.exe wordt gebouwd. Zo kan je het commando ‘make build’ geven, zonder dat je hoeft te onthouden hoe de executable file heet.

|  |
| --- |
| **rm := del**  **.PHONY: build clean**  **build: main.exe**  **clean:**  **$(rm) main.exe main.o engine.o string.o** |

De regel ‘rm := del’ is een assignment: de variabele rm krijgt de waarde del. Dit wordt vaak gedaan om alle systeem-afhankelijke delen te verzamelen in 1 stuk van het makescript. Op een Linux systeem wordt die regel ‘rm = rm –f’. De syntax $(rm) geeft aan dat daar de expansie (waarde) van de rm variabele moet worden gebruikt. Bij assignments is de volgorde wel van belang: de assignments worden in volgorde uitgevoerd.

Het kan handig zijn een makescript te verdelen over meerdere files, bv. omdat grote delen hetzelfde zijn voor meerdere projecten. Hiertoe kan je een include regel gebruiken zoals in onderstaand voorbeeld.

|  |
| --- |
| **# project-specific settings**  **TARGET ?= arduino\_due**  **# incluce the bmptk/Makefile.inc**  **include ../bmptk/Makefile.inc** |

De assignment met de ‘?=’ operator betekent dat TARGET de waarde arduino\_due krijgt \*tenzij\* TARGET al een waarde had. Veel variabelen in de bmptk Makefile.inc worden op deze wijze gezet, zodat ze in een project-specifieke makefile overruled kunnen worden.

Een makefile rule hoeft niet perse een compiler aan te roepen. Stel dat we een (embedded) applicatie maken die een mededeling moet laten zien. Die mededeling is een literal string die wordt teruggegeven door een functie in een .cpp file. We maken netjes een bijbehorende .hpp file aan om de functie vanuit main te kunnen aanroepen.

|  |  |  |
| --- | --- | --- |
| #include “message.hpp”  const char \*splash(){  return “Hello”;  } | const char \* splash(); | #include "message.hpp"  . . .  int main( void ){  . . .  hwlib::cout << splash();  } |
| message.cpp | mesage.hpp | main.cpp |

In een bmptk makefile is de main.cpp automatisch onderdeel van de applicatie, dus die hoeven we niet te noemen in de Makefile, maar de andere files moeten we wel noemen.

|  |
| --- |
| SOURCES := message.cpp  HEADERS := message.hpp |

De boodschap is nu hard gecodeert in de .cpp file. Een alternatief is dat we de .cpp file door de makefile laten aanmaken door een ander programma. We moeten dan in de Makefile noemen dat message.cpp afhankelijk is van (de executable van) dat programma. De bijbehorende command line roept die executable aan en leidt de uitvoer ervan naar de .cpp file.

|  |
| --- |
| message.cpp: make\_message.exe  make\_message > message.cpp |

De make\_message.exe executable wordt aangemaakt door make\_message.cpp te compileren. Dit kunnen we ook aangeven in de Makefile, zodat dit automatuisch gebeurt als we make\_message.cpp wijzigen. Dit gebeurt met het g++ commando, en we moeten aangeven dat de uitvoer naar de file make\_message.exe gaat. (Default gaat het resultat naar a.exe.)

|  |
| --- |
| make\_message.exe: make\_message.cpp  g++ make\_message.cpp -o make\_message.exe |

We hebben nu nog niet veel bereikt: de boodschap staat nu in een windows .cpp file in plaat van in een embedded .cpp file. Maar met dit mechanisme kunnen we de boodschap ook van een andere bron halen. In onderstaand voorbeeld staat de boodschap in de Makefile self, en wordt als command line parameter aan make\_message meegegeven. Als we de boodschap (in de Makefile) wijzigen moet message.cpp aangepast worden, dus we moeten ook aangeven dat message.cpp afhankelijk is van de Makefile.

|  |
| --- |
| SOURCES := message.cpp  HEADERS := message.hpp  message.cpp: Makefile make\_message.exe  make\_message splash "Hello brave new world!" > message.cpp    make\_message.exe: make\_message.cpp  g++ make\_message.cpp -o make\_message.exe |

Het make\_message programma is simpel, maar ziet er wat geknutseld uit omdat er in de uitvoer ” tekens moeten komen, die we in een string literal moeten opnemen als \”. De command line parameters (de functie naam en de struing zelf) zijn beschikbaar voor de main als argv[1] en argv[2][[3]](#footnote-3). Een net programa zou eerst checken of er wel 2 parameters zijn, dit is te zien aan argc.

|  |
| --- |
| #include <iostream>  int main( int argc, char \*\*argv ){  std::cout  << "#include \"message.hpp\"\n"  << "const char \* " << argv[ 1 ] << "(){\n"  << " return \"" << argv[ 2 ] << "\";\n"  << "}\n";  } |

De uitvoer is bv:

|  |
| --- |
| #include "message.hpp"  const char \* splash(){  return "Hello brave new world!";  } |

# Templates



|  |
| --- |
| behandelde begrippen:   * functie template, class template * impliciete instantiering van functie templates * template parameters * template staat in header file * operator[ ], operator[ ] const * proxy pattern * conversie operator |

## Functie template

Een template in C++ is een functie of klasse die extra parameters heeft, die tijdens het compileren worden meegegeven. Deze extra parameters kunnen typen (klassen) zijn of contante (integer) waarden. Het standaard voorbeeld van een functie template is het bepalen van het maximum van twee getallen. Het is niet moeilijk het maximum van twee integers te bepalen, hieronder staan twee versies. De eerste komt wellicht wat leesbaarder over, maar de tweede versie is wat een ervaren C++ programmeur zou schrijven. Het maakt gebruik van de ? operator, die werkt als een if statement maar dan voor twee expressies: als de waarde voor de ? waar is dan is het resultaat de eerste expressie, zo niet dan is het resultaat de tweede expressie.

|  |
| --- |
| int max( int a, int b ){  if( a > b ){  return a;  } else {  return b;  }  }  int max( int a, int b ){  return ( a > b ) ? a : b;  } |

Maar we willen wellicht ook het maximum van twee floats kunnen bepalen, en van twee doubles. Een waar om ook niet van twee kleuren, even aangenomen dat er voor kleuren een > operator bestaat? De source code van die andere max() functies is het zelfde als de integer max(), alleen het type van de parameters en de return waarde is anders. Dit kan, door het type dat we vergelijken uit de functie te lichten en te vervangen door T[[4]](#footnote-4), en dan aan te geven dat T een template parameter van de functie is. Omdat het type T wellicht een ingewikkeld object is waarvan het kopiëren tijd kost (of misschien niet eens mogelijk is) is de overdracht van de parameters veranderen van ‘by value’ naar ‘by const reference’.

|  |
| --- |
| template< typename T >  T max( const T & a, const T & b ){  return ( a > b ) ? a : b;  } |

Hiermee hebben we de compiler geïnstrueerd hoe hij, gegeven een type T, van twee waarden van dat type het maximum kan bepalen en teruggeven. Het enige wat daarbij wordt vereist is dat het betreffende type een operator> en een copy constructor heeft.[[5]](#footnote-5) Nu kunnen we in onze code naar hartenlust het maximum van dingen bepalen. Het leuke van een functie template is daarbij dat de compiler automatisch de functies aanmaakt zodra ze nodig zijn.

|  |  |
| --- | --- |
| std::cout << max( 11, 12 ) << "\n";  std::cout << max( 2.718, 3.14 ) << "\n";  std::cout << max( "ab", "cd" ) << "\n"; | |
| Gebruik van de max functie template | 02-01 |

De derde regel geeft een probleem: voor het type char[3] is geen operator> beschikbaar, dus als de compiler de max() functie voor dat type probeert aan te maken geeft hij een foutmelding:

|  |
| --- |
| main.cpp:14:31: error: no matching function for call to 'max(const char [3], const char [3])'  std::cout << max( "ab", "cd" ) << "\n"; |

De locatie van deze foutmelding is in dit geval de regel waar we de max() functie gebruiken, maar het had ook de regel in de functie template kunnen zijn waar de > gebruikt wordt. Het gebruik van templates kan soms zeer mysterieuze foutmeldingen geven.

Overigens is het niet nodig de max() template zelf te schrijven: de header <algorithm> bevat al een std::max(), die overigens standaard gebruik maakt van operator<, maar je ook de mogelijkheid biedt zelf een operator mee te geven die als ‘smaller than’ gebruikt moet worden. Daarnaast kan je met std::max ook het maximum van een reeks van waarden opvragen. En uiteraard is er ook een std::min().

In sommige gevallen is het nodig om expliciet op te geven voor welke parameters je een functie template wilt instantiëren. Dit doe je door die parameters op te geven tussen <> haakjes. In het volgende voorbeeld wordt dus de max<float> gebruikt in plaats van de max<int> die geselecteerd zou worden op grond van de parameters.

|  |
| --- |
| std::cout << max< float >( 5, 6 ) << "\n"; |

## Klasse template

Het zou handig zijn om een array te hebben dat iedere keer dat een element wordt gebruikt de index checkt. In C++ kan je dit zelf maken door het gebruik van templates en de speciale operator []. De volgende klasse template heeft twee template parameters: een type en een aantal. De klasse heeft als attribuut een array van dat aantal elementen van dat type. De methoden zijn volledig in de klasse body opgenomen. Dit is helaas verplicht bij templates, omdat de compiler om de template te kunnen instantiëren de volledige uitwerking van alle methoden nodig heeft. Templates staan dus altijd helemaal in de .hpp file.

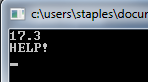
|  |
| --- |
| template< typename T, unsigned int N >  class array {  public:  T operator[]( int i ) const {  check( i );  return a[ i ];  }  T & operator[]( int i ){  check( i );  return a[ i ];  }  private:  T a[N];  void check( int i ){  if (( i < 0 ) || ( i >= N )){  std::cout << "HELP!\n";  while( true ){}  }  }  }; |

De check() methode controleert of de index binnen de grenzen valt. Daarbij kan gebruik gemaakt worden van de template parameter N, want voor een gegeven instantiering van dit template ligt N vast. Als de index niet binnen de grenzen valt schrijft check() een melding naar std::cout en hangt het programma op door een eeuwige lus binnen te gaan.[[6]](#footnote-6)

Naast de door de compiler gegenereerde methoden (default constructor, copy constructor, assignment, destructor) heeft de klasse twee operator[] methoden die sterk op elkaar lijken. De operator[] is de methode die wordt aangeroepen als je het object als array gebruikt. Een verschil tussen de twee methoden is het const keyword van de eerste net voor de {. Het effect hiervan is dat deze methode kan werken op een constant object, en aangeroepen zal worden als de [] operator wordt gebruikt op een const object. Deze methode geeft een T waarde ‘by value’ terug.

De tweede methode wordt gebruikt als de array notatie wordt gebruikt op een niet-const object. Deze methode geeft een T object terug ‘by reference’. Een reference kan ook aan de linkerkant van een assignment gebruikt worden, dus via deze methode kan een element van een (niet-const) array niet alleen worden gelezen maar ook geschreven, want als er naar de reference die de operator[] teruggaf wordt geschreven dan wordt er in feite naar het gerefereerde object, dus naar het array element, geschreven.

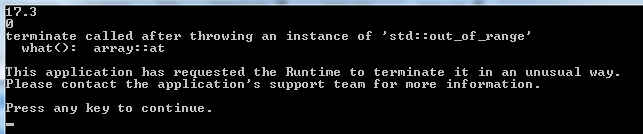
|  |  |
| --- | --- |
| int main(){  array< double , 9 > a;  a[ 1 ] = 17.3;  std:: cout << a[ 1 ] << "\n";  a[ -1 ] = 0.0;  } | |
| Do-it-yourself checked array | 02-02 |



## std::array<>

De standaard library bevat een template type array, dat de voorkeur heeft boven het gebruik van een C-stijl array. Dit type heeft behalve de beide [] operatoren ook twee at() functies, die de index checken en een exception genereren als de index ongeldig is. In onderstaand voorbeeld gebeurt dit pas op de laatste regel van main(), want de [] operator checkt de index niet!

|  |  |
| --- | --- |
| #include <iostream>  #include <array>  int main(){  std::array< double, 9 > a;  a[ 1 ] = 17.3;  std:: cout << a[ 1 ] << "\n";  a[ -1 ] = 0.0;  std:: cout << a[ -1 ] << "\n";  std:: cout << a.at( -1 ) << "\n";  } | |
| std::array | 02-03 |



## Specializatie

Ons array template gebruikt als implementatie een (C-stijl) array van de elementen.

|  |
| --- |
| template< typename T, unsigned int N >  class array {  public:  T operator[]( int i ) const {  return a[ i ];  }  private:  T a[N];  }; |

Als het template argument bool is, resulteert dat in een array van bool. Iedere bool neemt 1 byte in beslag (want een variabele kan niet kleiner zijn dan 1 byte). Een array van bool is dus nogal inefficient: in een byte kunnen 8 bitjes opgeslagen worden.

|  |
| --- |
| array< int, 10 > int\_10;  std::cout << "size of int\_10 = " << sizeof( int\_10 ) << "\n";    array< bool, 10 > bool\_10;  std::cout << "size of bool\_10 = " << sizeof( bool\_10 ) << "\n"; |



Om dit aan te pakken kunnen we een specializatie van ons algemene array template maken, speciaal voor bool. Deze versie is zelf een template met 1 template parameter

|  |
| --- |
| // hiervoor staat de generieke array template  // array template speciaal voor bool arrays  template< unsigned int N >  class array< bool, N > {  public:  bool operator[]( int i ) const {  return ( a[ i / 8 ] >> ( i % 8 ) ) & 0x01;  }  private:  unsigned char a[ ( N + 7 ) / 8 ];  };  . . .  array< int, 10 > int\_10;  std::cout << "size of int\_10 = " << sizeof( int\_10 ) << "\n";    array< bool, 10 > bool\_10;  std::cout << "size of bool\_10 = " << sizeof( bool\_10 ) << "\n"; |



Deze versie wordt alleen gebruikt als er als type bool wordt opgegeven.

## Proxy pattern

Overigens zit bij deze implementatie van een bool array nog een adder onder het gras: hoe implementeer je de operator[] die je links van de assignment kan gebruiken? De losse bitjes in het unsigned char array hebben geen individueel adres, dus je kan er geen pointer naar creeeren en (dus) ook geen reference. De std::array heeft om deze reden geen ruimte-efficiente specializatie voor bool. In de standaard library zit echter een std::vector<bool>, die die wel ruimte-efficient is. Het probleem van de operator[] wordt hier opgelost doordat die operator geen bool & teruggeeft, maar en object van het speciale type vector<bool>::reference. Dit type implementeert een operator=, die er voor zorgt dat de toegekende waarde wordt opgeslagen in de juiste bit van de std::vector<bool>.

Voor onze array<bool> zouden we dit als volgt kunnen realizeren. Eerst schrijven we get() en set() functies die de array elementen (de individuele bitjes) benaderen. De operator[] const is triviaal: die roept gewoon get() aan.

|  |
| --- |
| template< unsigned int N >  class array< bool, N > {  public:  bool get( int i ) const {  return (( a[ i / 8 ] >> ( i % 8 ) ) & 0x01 ) != 0;  }    void set( int i, bool b ){  if( b ){  a[ i / 8 ] |= ( 0x01 << ( i % 8 ));  } else {  a[ i / 8 ] &= ~ ( 0x01 << ( i % 8 ));  }  }    bool operator[]( int i ) const {  return get( i );  }  // hier komt de reference klasse  private:  unsigned char a[ ( N + 7 ) / 8 ];    }; |

De operator[] (zonder const) moet een object teruggeven waaraan een bool geassigned kan worden, dus die klasse moet een assifgnment operator hebben die een bool acceptert en zorgt dat die bool op de juiste plek in de array terecht komt. Dit kan via de set() operator, maar dan moethet object dus wel de beschikking hebben over het array object, en de index. Die beide dingen moeten we dus aan de constructor van de referenec klasse meegeven en in het object opslaan.

|  |
| --- |
| class reference {  private:  array & subject;  int n;  public:  reference( array & subject, int n ): subject( subject ), n( n ){}    reference & operator=( bool b ){  subject.set( n, b );  return \*this;  }    operator bool(){  return subject.get(n );  }  };    reference operator[]( int i ){  return reference( \*this, i );  } |

Tot slot moeten we zo’n reference object ook aan de rechter kant van een statement gebruiken, als bool waarde. Om dit mogelijk te maken is er de operator bool(), dit is een conversie operator. Als een reference object wordt gebruikt op een plek waar een bool vereist is, dan wordt automatisch deze conversie aangeroepen.

Er is hier alleen het allereenvoudigste gebruik van templates beschreven. Templates vormen een programmeertaal op zich, die in complexiteit kan wedijveren met de rest van C++.

# constexpr

Neem aan dat we een sinus functie willen tekenen, voor het gemakt vertikaal op de hwlib::cout. Dit kan met een simpel programma dat gebruik maakt van de sin() functie uit de standaard library. Dit programma rekent (op de Due) uit wat de (afgeronde en geschaalde) waarden van de sinus is in stappen van 10 graden, en tekent op die plek een \*.

|  |
| --- |
| #include <math>    double radians\_from\_degrees( int degrees ){  return 2 \* 3.14 \* ( degrees / 360.0 );  }  int scaled\_sine\_from\_degrees( int degrees ){  return 30 \* ( 1.0 + sin( radians\_from\_degrees( degrees )));  }  int main( void ){  . . .  for( int angle\_degrees = 0; angle\_degrees < 360; angle\_degrees += 10 ){  for( int i = 0; i < scaled\_sine\_from\_degrees( angle\_degrees ); ++i ){  hwlib::cout << " ";  }  hwlib::cout << "\*\n";  }  } |

De Cortex-M3 processor van de Due heeft geen floating point hardware, dus die floating point berekeningen worden gedaan door library code die automatisch door de compiler wordt meegenomen. De omvang hiervan is niet triviaal, en de floating point berekeningen zijn ook veel langzamer dan integer berekeningen.[[7]](#footnote-7) Het sinus programma is 9988 bytes, zonder de afgebeelde code (maar wel met met opstarten en gebruik van hwlib::cout) is het maar 2440 bytes. Hoe vaak de sin() wordt aangeroepen doet er niet zo veel toe, een programam dat 1 keer een sin() aanroept is al 9964 bytes.

Een eigenschap van de compiler libraries is dat die niet in kleine brokjes zijn opgedeeld: zodra je een functie uit zo’n library gebruikt wordt de hele library meegelinkt, ook de code die je niet gebruikt. Dit wordt gedaan om het linken van grote programma’s die veel libraries gebruiken te versnellen. Er zijn compiler/linker opties die er voor zorgen dat het compileren/linken wel fijnmaziger werkt (bmptk gebruikt automatisch die opties[[8]](#footnote-8)), maar de compiler libraries zijn niet met die opties gecompileerd.

We kunnen de sin() functie uit de library vervangen door een zelf geschreven functie. Met googelen kan je uitvinden dat een sinus benaderd kan worden met een Taylor reeks.[[9]](#footnote-9)



Met hulpfuncties voor machten en faculteit kunnen we dit eenvoudig naar C vertalen:

|  |
| --- |
| double sin( double a ){  return  a  - pow( a, 3 ) / fac( 3 )  + pow( a, 5 ) / fac( 5 )  - pow( a, 7 ) / fac( 7 )  + pow( a, 9 ) / fac( 9 )  - pow( a, 11 ) / fac( 11 )  + pow( a, 13 ) / fac( 13 );  } |

|  |
| --- |
| double pow( double g, int n ){  double result = 1;  while( n > 0 ){  result \*= g;  --n;  }  return result;  }  double fac( int n ){  double result = 1;  while( n > 0 ){  result \*= n;  --n;  }  return result;  } |

Met deze zelfgeschreven sin() is de executable nog maar 4932 bytes. Maar de sinus waarden worden nog wel op de Due uitgerekend tijdens het tekenen van de sinus. Als dat een probleem is (bv. omdat het tekeken snel moet gebeuren) kunnen we de berekening verplaatsen naar het begin van de main, zeg maar de initializatie van de applicatie. Dit maakt het printen sneller, maar heeft geen (positieve) invloed op de omvang van de excutable.

|  |
| --- |
| int main( void ){  . . .    const int table[ 36 ] = {  scaled\_sine\_from\_degrees( 0 ),  . . .  scaled\_sine\_from\_degrees( 350 )  };    for( int angle\_degrees = 0; angle\_degrees < 360; angle\_degrees += 10 ){  for( int i = 0; i < table[ angle\_degrees / 10 ]; ++i ){  hwlib::cout << " ";  }  hwlib::cout << "\*\n";  }  } |

In dit soort gevallen (er moeten waarden uitgerekend worden, en dat mag niet tijdens het runnen gebeuren) werd (en wordt) vaak gebruik gemaakt van excel sheets, waaruit dan een kolom met waarden wordt gecopieerd naar de C of C++ source code. Dit is natuurlijk omslachtig en foutgevoelig als er iets veranderd moet worden aan de berekening. Een andere (al wat betere) omlossing is om een applicatie te schrijven die een file met de waarden genereert die wordt meegenomen met het compileren. Dit kan je automatiseren in het Makescript. Klinkt dit bekend?

Vanaf C++ 0x11 is er een manier om dit geheel binnen C++ te doen dmv. het keyword constexpr. Dit wordt op twee manieren gebruikt. Ten eerste kan je het aan de initializatie van een (constante) variabele hangen om aan te geven dat de waarde tijdens het compileren al berekend \*moet\* worden. Dit heeft als practische consequentie dat de betreffende non-const variabele in het DATA segment terecht komt (en zijn initiele waarde in ROM), en een const variabele in het code of read-only segment (ROM). Zonder de constexpr is de compiler vrij om de waarde tijdens het compileren of tijdens het runnen te berekenen.

|  |
| --- |
| const int x = 15 + 27; // dit mag een run-time berekening zijn  constexpr const int x = 50 – 8; // dit \*moet\* een compile-time berekening zijn |

Voor het berekenen van een waarde tijdens het compileren mocht je (voor C++ 0x11) gebruik maken van literal waarden, constanten, en ingebouwde operatoren. Met C++ 0x11 zijn daar constexpr functies bijgekomen. Een functie die is gedeclareerd als constexpr moet aan een paar eisen voldoen waardoor de compiler in staat is die functie (mits de waarden van de argumenten bekend zijn) tijdens het compileren te uitrekeken. Voor C++ 0x14 zijn die eisen redelijk relaxed en intuitief (bv. geen globale dingen gebruiken). In C++ 0x11 goldt de veel strengere eis dat zo’n functie slechts een return statment mocht bevatten.

De meeste library funcies waarvan je zou denken dat ze constexpr zouden moeten zijn (bv. sinus) zijn dat in de praktijk niet, omdat

* volgens de (antieke) definitie veel library functies een fout aangeven door de globale variable errno een waarde te gebven, en een constexpr functie mag geen globale dingen gebruiken
* constexpr relatief nieuw is (2014 voor de makkelijke versie) en veel libraries gewoon nog niet zijn aangepast, mede omdat daar op grotere system (PC’s etc) niet zo veel behoefte aan is.

Om onze sinus tabel tijdens het compileren te laten berekenen moeten we dus het array constexpr maken. In C++ 0x11 moet het array ‘constexpr const’ zijn: constexpr betekent ‘tijdens het compileren berekend’, const betekent ‘kan na de initializatie niet meer wijzigen’. In C++ 0x14 impliceert constexpr dat de variable ook const is, daar zou const dus weggelaten kunnen worden.

|  |
| --- |
| int main( void ){  . . .  constexpr const int table[ 36 ] = {  scaled\_sine\_from\_degrees( 0 ),  . . .  scaled\_sine\_from\_degrees( 350 )  };  . . .  } |

De initiele waarden worden uitgerekend door de functie scaled\_sine\_from\_degrees(), dus die moet constexpr zijn, en alle functies die hij aanroept dus ook. Gelukkig voldeden die functies al aan de eisen, dus dat is alleen een kwestie van constexpr er voor zetten.

|  |
| --- |
| constexpr double pow( double g, int n ) . . .  constexpr double fac( int n ) . . .  constexpr double sin( double a ) . . .  constexpr double radians\_from\_degrees( int degrees ) . . .  constexpr int scaled\_sine\_from\_degrees( int degrees ) . . . |

De constexpr qualifier kan ook toegepast worden op functies van klassen, inclusief constructors. Een voorbeeld is de hwlib::position klasse.

|  |
| --- |
| class location final {  public:  int x, y;    constexpr location( int x, int y ): x( x ), y( y ){}  constexpr location operator+( const location rhs ) const {  return location( x + rhs.x, y + rhs.y );  }    constexpr location operator-( const location rhs ) const {  return location( x - rhs.x, y - rhs.y );  }  }; |

De constructor en beide operatoren van deze klasse zijn constexpr, dus berekeningen op waarden van deze klasse kunnen al tijdens het compileren gebeuren.

|  |
| --- |
| #include "hwlib.hpp"  namespace hwlib {  constexpr location operator/( const location & lhs, int rhs ){  return location( lhs.x / rhs, lhs.y / rhs );  }  }  constexpr auto origin = hwlib::location( 0, 0 );  constexpr auto oled\_size = hwlib::location( 128, 64 );  constexpr auto middle = origin + oled\_size / 2; |

Merk op dat er nog geen operator/ bestond voor location. Die is \*binnen de hwlib namespace\* toegevoegd. Anders dan een klasse is een namspace niet afgesloten, je kan er altijd dingen aan toevoegen.

De C++ 0x11 bepereking dat een constexpr functie alleen een return statement mag bevatten is minder bezwaarlijk dan het lijkt omdat er in dat statement de ? operator en recursie gebruikt mag worden. Voor sommige functies levert dit misschien zelfs een leesbaardere versie op.

|  |
| --- |
| // C++ 0x14 versie  double fac( int n ){  double result = 1;  while( n > 0 ){  result \*= n;  --n;  }  return result;  }  // C++ 0x11 versie  double fac( int n ){  return ( n == 0 )  ? 1.0  : n \* fac( n – 1 );  } |

De lookup table die we hebben gebruikt om sinussen op te zoeken werd geinitializeerd met een reeks waarden die we helemaal uit moesten schrijven. Dat wordt natuurlijk ondoenlijk bij grotere aantallen waarden. We kunnen een lookup klasse maken, die in zijn constructor een for loop bevat die alle waarden uitrekent en in een tabel stopt.[[10]](#footnote-10)

|  |
| --- |
| class sine\_lookup {  private:  int values[ 360 ];  public:  constexpr sine\_lookup:  values()  {  for( int i = 0; i < 360; ++i ){  values[ i ] = scaled\_sine\_from\_degrees ( i );  }  }  constexpr int get( int n ) const {  return values[ n ];  }  };  auto sinusses = sine\_lookup();    for( int angle\_degrees = 0; angle\_degrees < 360; angle\_degrees += 10 ){  for( int i = 0; i < sinusses.get( angle\_degrees ); ++i ){  hwlib::cout << " ";  }  hwlib::cout << "\*\n";  } |

Deze klasse doet wat hij moet doen, maar hij is erg specifiek voor dit geval: hij slaat 360 int waarden op die uitgerekend zijn door de functie scaled\_sine\_from\_degrees(). De eerste twee aspecten kunnen we algemener maken door er template parameters van te maken.

|  |
| --- |
| template< int N, typename T >  class sine\_lookup {  private:  T values[ N ];  public:  constexpr lookup():  values()  {  for( int i = 0; i < N; ++i ){  values[ i ] = scaled\_sine\_from\_degrees ( i );  }  }  constexpr T get( int n ) const {  return values[ n ];  }  };  auto sinusses = sine\_lookup< 360, int >(); |

Het derde aspect is iets lastiger, omdat we geen functie kunnen meegeven als template parameter. We kunnen echter wel een functie meegeven als parameter van de constructor. Om alle mogelijkheden open te houden kunnen we zelfs vermijden dat we aangeven wat het type van die parameter is, door van de constructor een template functie te maken. We kunnen nu bij het initializeren van de lookup aangeven welke functie er gebruikt moet worden.

|  |
| --- |
| template< int N, typename T >  class lookup {  private:  T values[ N ];  public:  template< typename F >  constexpr lookup( F function ):  values()  {  for( int i = 0; i < N; ++i ){  values[ i ] = function( i );  }  }  constexpr T get( int n ) const {  return values[ n ];  }  };  auto sinusses = lookup< int, 360>( scaled\_sine\_from\_degrees ); |

Omdat we het type van de ‘functie’ een template parameter hebben gemaakt, zijn we niet beperkt tot functies. Er zijn andere dingen in C++ die aangeroepen kunnen worden alsof ze een functie zijn. Een interessant geval is de lambda (geintroduceerd in C++ 11). Dat is een soort functie die je kunt schrijven op de plek waar hem als parameter meegeeft.[[11]](#footnote-11)

|  |
| --- |
| auto sinusses = lookup< 360, int >(  []( int x )-> int { return 30 \* ( 1.0 + sin( radians\_from\_degrees( x ))); }  ); |

# RAII

Het komt vaak voor dat een functie voor het vervullen van zijn taak een resource nodig heeft, die hij vlak voor het returnen weer moet vrijgegeven. Een voobeeld is het claimen van exclusief gebruik van de console output dmv. een mutex semafoor.[[12]](#footnote-12)

|  |
| --- |
| void print( int n, const char \*s ){  cout\_mutex.wait();  hwlib::cout << n;  for( auto p = text; \*p != '\0'; ++p ){  hwlib::cout << \*p;  sleep( 10 \* rtos::ms );  }  cout\_mutex.signal();  } |

Dit lijkt triviaal, maar het is een veel voorkomende fout om te vergeten de resource vrij te geven, vooral als er meerdere resources zijn en/of de functie op meerdere manieren kan eindigen. Een manier om dit aan te pakken is het Resource Allocation Is Iniotialization (RAII) pattern.[[13]](#footnote-13) Dit betekent dat een resource altijd wordt geallokeerd in de constructor van een lokaal object, en dat de resource wordt vrijgegeven in de destructor van dat object. Een destructor is een klasse functie die automatisch wordt aangeroepen als een object op het punt staat vrij gegeven te worden.

|  |
| --- |
| class mutex\_resource {  private:  rtos::mutex & mutex;    public:  mutex\_resource( rtos::mutex & mutex ):  mutex( mutex )  {  mutex.wait();  }    ~mutex\_resource(){  mutex.signal();  }  };  void print( int n, const char \*s ){  auto exclusive\_use = mutex\_resource( cout\_mutex );  hwlib::cout << n;  for( auto p = text; \*p != '\0'; ++p ){  hwlib::cout << \*p;  sleep( 10 \* rtos::ms );  }  // exclusive\_use is destructed here  } |

Mocht je geen zin hebben om voor een resource een aparte klasse te maken dan kan je een algemene resource klasse gebruiken, waaraan je de allocatie en deallocatie als lambda’s meegeeft. Hierbij moet je een truuk gebruiken: je kan een template klasse niet instatieren zonder expliciet aan te geven wat de template parameters zijn, maar bij een template functie kan dat wel. Er wordt hier dus een functie gebruikt om de typen van de beide parameters te bemachtigen, en die als template parameters door te geven aan de template klasse.

|  |
| --- |
| template< typename C, typename D >  class resource\_class {  private:  D destruct;  public:  resource\_class( C create, D destruct ):  destruct( destruct )  {  create();  }    ~resource\_class(){  destruct();  }  };  template< typename C, typename D >  resource\_class< C, D > resource( C create, D destruct ){  return resource\_class< C, D >( create, destruct );  }  void print( int n, const char \*s ){  auto exclusive\_use = resource(  [&](){ cout\_mutex.wait(); },  [&](){ cout\_mutex.signal(); }  );  hwlib::cout << n;  for( auto p = text; \*p != '\0'; ++p ){  hwlib::cout << \*p;  sleep( 10 \* rtos::ms );  }  } |

# Lempel-Ziv compressie

Een eenvoudige maar effectieve manier om data te comprimeren is om vaker voorkomende reeksen tekens twe vervangen door een (kortere) marker. Dit bespaart iedere keer het verschil in lengte tussen de reeks en de marker, maar

* Je hebt wel extra ruimte nodig om de vertaal-tabel op te slaan.
* Wat doe je als de marker voorkomt in de oorspronkelijke text?

|  |
| --- |
| Oorspronkelijke bericht: ”Hello hello brave new world! Do you like Trello?”  Vervang “ello” door “@”  Gecomprimeerd bericht: ”H@ h@ brave new world! Do you like Tr@?” |

De truuk van Lempel-Ziv compressie is dat je data \*tot nu toe\* wordt gebuikt als vertaaltabel: een marker geeft aan hoeveel posities terug in de text je moet kijken voor de vervanging, en hoe lang die vervanging is. (Het punt hoe om te gaan met het marker teken in de oorspronkelijke text blijft.)

|  |
| --- |
| Oorspronkelijke bericht: ”Hello hello brave new world! Do you like Trello?”  Een vervanging is aangegeven als @[d;n]  Gecomprimeerd bericht: ”Hello h@[-6,5]brave new world! Do you like Tr@[-37,4]?” |

Het is natuurlijk alleen de moeite waard een vervanging te doen als die korter is dan het fragment dat wordt vervangen.

Het is vaak niet handig de complete boodschap te decomprimeren, bv. omdat daar niet voldoende ruimte voor is. Om dit te ondervagen kan je het aantal posities dat een verbanging terug mag kijken bepreken. Dit betekent ook dat de veravnging zelf kleiner kan zijn.

De volgende code iomplementeert een simpele vorm van Lempel-Ziv compressie van een text. Als marker wordt ‘@’ gebruikt, die wordt gevolgd door twee bytes die de offset (hoeveel plekken terug) en de lengte van de vervanging aangeven. Beide waarden moeten in de range 0..80 liggen. Om rare controle-tekens in the gecomprimeerde text te vermijden wordt bij die waarde ‘ ’ (de ASCII code voor spatie) opgeteld.

De compressor leest eerst alle text in een buffer van N characters, door herhaald aanroepen van de write() functie. Het einde van de input wordt aangegeven doordat de read() functie ‘\0’ teruggeeft. Na het comprimeren wordt de gecomprimeerde text weggeschreven door herhaald aanroepen van de write() functie. Om allerlei soorten read() en write() functies aan te kunnen is de compress() functie een functie template.

|  |
| --- |
| template< int N >  class lz\_compressor {  . . .  public:  template< typename D, typename S >  void compress( D write, S read ){  read\_buffer( read );  write\_compressed\_buffer( write );  }  }; |

Om deze compressor te gebruiken kan je bv twee files openen en passende lambda’s meegeven voor de read en write functies. In het voorbeeld moet je bij die lamda’s aangeven dat de file variabele binnen de lambda gebruikt wordt een moet worden ‘doorgegeven’ by reference.

|  |
| --- |
| lz\_compressor< 4096 > compressor;    std::ifstream f1;  f1.open( . . . );  if( ! f1.is\_open()){  std::cerr << "input file not opened";  return -1;  }    std::ofstream f2;  f2.open( . . . );  if( ! f2.is\_open()){  std::cerr << "output file not opened";  return -1;  }    compressor.compress(  [ &f2 ]( char c ){ f2.put( c ); },  [ &f1 ]()-> int { auto c = f1.get(); return f1.eof() ? '\0' : c; }  ); |

Het vullen van de buffer is eenvoudig: de read() functie die de aanroeper heeft meegegeven wordt aangeroepen totdat die ‘\0’ teruggeeft.Daarbij moet nog gecheckt worden of er geen ‘@’ tekens voorkomen, en of de buffer niet overschreden wordt. Om met alle soorten read() functies te kunnen werken is read\_buffer() een template functie.

|  |
| --- |
| char buffer[ N ];  int used;  template< typename F >  void read\_buffer( F read ){  used = 0;  for(;;){  auto c = read();  if( used == N ){  std::cerr << "buffer overflow";  break;  }  if( c == '@' ){  std::cerr << "text contains @";  break;  }  if( c == '\0' ){  break;  }  buffer[ used++ ] = c;  }  } |

Bij het wegschrijven van de gecomprimeerde tekts moet op iedere plek in de tekst gekeken worden wat de langste match is tussen wat er vanaf die plek staat met de voorgaande tekts. Een marker met waarden neemt 3 tekens in beslag, dus een vervanging is alleen zinnig als die match ten minste 3 tekens is. Als dat zo is, dan wordt er een marker met de waarden voor de offset en lengte weggeschreven (en de overeenkomstige tekens worden overgeslagen). Zo niet, dan wordt 1 character geschreven.

|  |
| --- |
| template< typename F >  void write\_compressed\_buffer( F write ){  for( int i = 0; i < used; ){  auto match = find( i );  if( match.length >= 2 ){  write( '@' );  write( ' ' - match.offset );  write( ' ' + match.length );  i += match.length;  } else {  write( buffer[ i++ ] );  }  }  } |

Bij het zoeken naar een match mag maar 80 posities terug gekeken worden, en de langste ‘match’ moet gebruikt worden.[[14]](#footnote-14) Hierbij is het handig om een match datatype te hebben, met een operator< om de beste match te kunnen selecteren, en een operator<< voor debuggen.

|  |
| --- |
| // a match found in the previously processed characters:  // the start offset (negative), and the number of characters  struct match {    int offset;  int length;    match( int offset = 0, int length = 0 ):  offset( offset ), length( length ) {}    friend std::ostream & operator<<( std::ostream & lhs, const match & m ){  return lhs << "[" << m.offset << ";" << m.length << "]";  }    bool operator<( const match & rhs ) const {  return length < rhs.length;  }  }; |

Het zoeken naar een match begint 80 posities voor de huidige positie, maar natuurlijk niet voor het begin van de buffer. Het zoeken eindigt net voor de huidige positie. Op iedere plek wordt gekeken hoeveel opeenvolgende tekens hetzelfde zijn.

|  |
| --- |
| match find( int index ){  match best( 0, 0 );  for( int i = std::max( 0, index - 80 ); i < index - 1; ++i ){  auto length = std::min(  80,  count\_identical\_prefix\_length( buffer + i, buffer + index ));  best = std::max( best, match( index – i - 1, length ) );  }  return best;  } |

# Template metaprogramming

Een specializatie van een template kan voor een bepaald type zijn, maar ook voor een bepaalde waarde. Dit kan gebruikt worden om (tijdens compileren) waarden van elkaar te onderscheiden en er zelfs mee te rekenen. Deze techniek, het gebruik van template specializatie om dingen te doen tijdens het compileren, wordt template metaprogramming genoemd.

Het onderstaande voorbeeld gebruikt een template om van een integer waarde te bepalen of die nul is of niet nul. De basis aanname is dat een integer waarde niet nul zal zijn, en daarnaast is er een specializatie van het ene geval dat wel nul is.

|  |
| --- |
| template< int N >  struct is\_nul {  static constexpr const char \* text = "is not nul";  };  template<>  struct is\_nul< 0 >{  static constexpr const char \* text = "is nul";  };  int main(){    const auto x = 0;  const auto y = 1;    std::cout << x << " " << is\_nul< x >::text << "\n";  std::cout << y << " " << is\_nul< y >::text << "\n";    } |

## Recursive specialization

In het vorige voorbeeld is het enige onderdeel van het template een (text) constante. Dit is typisch voor template metaprogramming: de klassen die er in voorkomen worden nooit gebruikt om er variabelen van te maken, maar alleen om de statische constante eigenschappen er van te gebruiken. Zo’n eigenschap kan je recursief definieren, bv. om te bepalen of een positief getal even is of niet. Nul is even, 1 is niet even, de ‘evenheid’ van alle andere (positieve) getallen is gelijk aan de evenheid van dit getal min 2.

|  |
| --- |
| template< int N >  struct is\_even {  static constexpr bool value = is\_even< N - 2 >::value;  };    template<>  struct is\_even< 0 >{  static constexpr bool value = true;  };  template<>  struct is\_even< 1 >{  static constexpr bool value = false;  };  int main( void ){    const auto x = 5;  const auto y = 90;    std::cout << x << " is even? " << is\_even< x >::value << "\n";  std::cout << y << " is even? " << is\_even< y >::value << "\n";    } |

Andere klassieke voorbeelden zijn het bepalen van de faculteit of de fibonnachi functie. Merk op dat in al deze gevallen er geen berekening plaats vindt tijdens het runnen.

|  |
| --- |
| #include <iostream>  template< int N >  struct fac {  static constexpr int value = N \* fac< N - 1 >::value;  };  template<>  struct fac< 0 >{  static constexpr int value = 1;  };  int main( void ){    const auto x = 12;    std::cout << "fac< " << x << " >::value = " << fac< x >::value << "\n";    } |

|  |
| --- |
| #include <iostream>  template< int N >  struct fac {  static constexpr int value = N \* fac< N - 1 >::value;  };  template<>  struct fac< 0 >{  static constexpr int value = 1;  };  int main( void ){    const auto x = 12;    std::cout << "fac< " << x << " >::value = " << fac< x >::value << "\n";    } |

## loop

Het komt in bit-banging code vaak voor dat je voor ieder bit in een byte iets moet doen. De standaard manier om dat te doen is natuurlijk met een for loop.

|  |
| --- |
| int n = 0;  for( int i = 0; i < 8; ++i ){  hwlib::cout << ++n << "\n";  } |

Een andere manier is om de for loop te verbergen in een template, en de body van de for loop (de code tussen de { } ) mee te geven als lambda.

|  |
| --- |
| template< int N, typename T >  void loop( const T & body ){  for( int i = 0; i < N; ++i ){  body();  }  }    int n = 0;    loop< 10 >( [&](){  hwlib::cout << ++n << "\n";  });    } |

Dit ziet er nogal omslachtig uit, maar het heeft en aantal (potentiele) voordelen:

* De loop variabele is in de aanroep niet meer zichtbaar: je hoeft er geen naam voor te kiezen, en die naam leidt bij het lezen van de code ook niet af.
* Het type van de loop variabele wordt niet meer genoemd in de loop. De template kan op grond van het aantal iteraties kiezen wat voor type (char, short, int, long, long long) gebruikt wordt.
* Op 8-bit chips kan het beter zijn om twee geneste loops te gebruiken die 8-bit counters gebruiken, ipv. Een enkele loop met een 16-bit cunter.
* Bij sommige combinaties van compilers en target chips is het beter om de loop te schrijven als “do { . . . } while( --n );” omdat je dan 1 jump vermijdt.

Als de loop een ‘performance bottleneck’ is dan kan het een goed idee zijn de loop te unrollen. Dit kan gebeuren in de template.

Voor het unrollen van een loop moeten we een \*partiele\* specializatie maken (specfiek voor 0 iteraties, maar nog steeds een template voor de body). Helaas staat C++ partiele specializatie van functies niet toe, maar we kunnen vrijwel hetzelfde bereiken door een klasse te specializeren en de functies om te schrijven tot constructors van die klassen.

|  |
| --- |
| template< int N >  struct loop\_unrolled{  template< typename Body >  loop\_unrolled( const Body & body ){  body();  loop\_unrolled< N - 1 > dummy( body );  }  };  template<>  struct loop\_unrolled< 0 >{  template< typename Body >  loop\_unrolled( const Body & body ){}  }; |

Helaas moeten we ook nog iets aanpassen in de aanroep. We willen dat de body() calls in-line worden, en dat kan je alleen aangeven bij de lambda, dus in de aanroep.

|  |
| --- |
| loop\_unrolled< 10 >( [&]() \_\_attribute\_\_ ((always\_inline)) {  hwlib::cout << ++n << "\n";  }); |

## SFINAE

Met specializatie kan je aangeven welke variant van een template gebruikt moet worden voor een bepaald type of waarde van een of meer van de template parameters. Er is nog een tweede manier om te bepalen welke variant van een template gebruikt wordt die bekend is onder de ongelukkige afkorting Subsitution Failure Is Not An Error (SFINAE). Stel dat we twee varianten van een template willen maken, een voor integer typen, en een andere voor floating point typen. Dit kunnen we doen door een template predicaat te maken dat test of een type een floating point type is. Zo’n predicaat heeft een bepaald onderdeel als zijn argument een floating point type is, en hij heeft dat niet als zijn argument geen floating point type is.

|  |
| --- |
| template< typename T >  struct is\_floating {  };  template<>  struct is\_floating< float > {  typedef void marker;  };  template<>  struct is\_floating< double > {  typedef void marker;  }; |

Als je dit predicaat gebruikt met een niet-floating type en de marker opvraagt dan krijg je een compiler foutmelding, want de basis versie heeft die marker niet.

|  |
| --- |
| is\_floating< double >::marker x; // OK  is\_floating< int >::marker y; // compiler error |

Maar als je de marker van dit pedicaat opvraagt in de expansie van een template specializatie en de marker bestaat niet, dan veroorzaakt dit \*geen\* compiler foutmelding, het enige effect is dat die specializatie niet gebruikt wordt. Met andere woorden: Subsitution Failure (de …::marker bestaat niet) Is Not An Error (maar veroorzaakt wel dat de betreffende specializatie niet gebruikt wordt).

|  |
| --- |
| template< typename T, typename M = void >  struct what\_is\_it {  static constexpr const char \* text  = "not a floating point type ...";  };  template< typename T >  struct what\_is\_it< T, typename is\_floating< T >::marker > {  static constexpr const char \* text  = "a floating point type!!";  }; |

|  |
| --- |
| std::cout  << "double : "  << what\_is\_it< double >::text  << "\n";  std::cout  << "int : "  << what\_is\_it< int >::text  << "\n"; |

1. Vroeger was de preprocessor een aparte executable die een output file aanmaakte, die vervolgens door de compiler werd gecompileerd. Tegenwoordig is de preprocessor doorgaans geintegreerd in de compiler, en de tussen-text wordt nooit naar een file geschreven (tenzij je daar met een command line optie om vraagt). [↑](#footnote-ref-1)
2. Het file formaat van een library file lijkt wel wat op een .zip file: het bevat een aantal object files. [↑](#footnote-ref-2)
3. argc[0] bestaat ook, dit is is de naam van de executable. [↑](#footnote-ref-3)
4. Het lijkt de gewoonte te zijn om voor template parameters losse hoofdletters te gebruiken. [↑](#footnote-ref-4)
5. De copy constructor is nodig om de return waarde by value te kunnen teruggeven. [↑](#footnote-ref-5)
6. Een beter keuze zou zijn om hier een exception te genereren, maar dat wordt pas in V2CPSE2 behandeld. [↑](#footnote-ref-6)
7. Op een PC is de situatie heel anders: daar heeft de processor floating point hardware, zodat er geen extra code nodig is, en floating point berekeningen zijn doorgaans net zo snel als integer berekeningen. [↑](#footnote-ref-7)
8. -fdata-sections -ffunction-sections [↑](#footnote-ref-8)
9. Er zijn vaak betere (snellere) benaderingen dan een Taylor reeks, maar dat is wel de meest bekende. Let er als je zo’n reeks gebruikt goed op voor welke waarden de reeks goed werkt, de Taylor reeks voor sinus met 3 elementen werkt prima tussen -180 en +180 graden, maar heet slecht daarbuiten. [↑](#footnote-ref-9)
10. Om dit ‘at compile time’ te doen moet die constructor constexpr zijn, en dat vereist C++ 0x14. [↑](#footnote-ref-10)
11. Lambda’s zijn geen onderdeel van de CPSE1 stof en worden in CPSE2 in wat meer detail behandeld. Helaas zijn lambda’s geen constexpr’s, dus voor het maken van een compile-time opzoektabel zijn ze niet geschikt. [↑](#footnote-ref-11)
12. Dit soort problemen komen vooral voor bij het gebruik van allocatie op de heap en het gebruik van excepties. Beiden komen in een volgend vak aan de orde. [↑](#footnote-ref-12)
13. RAII staat wat mij betreft op een gedeelde eerste plaats (met SFINAE) voor de slechtste afkorting voor een een C++/informatica term. [↑](#footnote-ref-13)
14. De lengte van een match moet ook nog beperkt worden tot 80 tekens, maar in de voorbeeld implementatie gebeurt dat automatisch omdat een match nooit voorbij de huidige positie kijkt. [↑](#footnote-ref-14)