**50**

HBO ICT

PropedeusE - TI

*Reader*

Object Oriented Programming in C++

**TICT-V1OOPC-15**



Studiejaar 2018-2019

|  |  |
| --- | --- |
|  |  |
| **Cursuseigenaar** | Wouter van Ooijen |
| **Auteur(s)** | Wouter van Ooijen |
| **Datum** | 2019-04-01 |
| Versie | 3.3 |
|  | |

© Institute for ICT, Hogeschool Uterecht, 2017 - 2019

Contents

[2. Inleiding 4](#_Toc6212732)

[3. C++ Geschiedenis 6](#_Toc6212733)

[4. Klassen en compositie 8](#_Toc6212734)

[4.1. Een window 8](#_Toc6212735)

[4.2. Teken een lijn 8](#_Toc6212736)

[4.3. Een lijn struct 9](#_Toc6212737)

[4.4. Constructors 10](#_Toc6212738)

[4.1. Reference parameters en reference variabelen 12](#_Toc6212739)

[4.2. Class 12](#_Toc6212740)

[4.3. Verdeling over header en source files 14](#_Toc6212741)

[4.4. Compositie 14](#_Toc6212742)

[4.5. Associatie 17](#_Toc6212743)

[5. const, std::array<>, macro’s 18](#_Toc6212744)

[5.1. const variabelen 18](#_Toc6212745)

[5.2. const parameters 18](#_Toc6212746)

[5.3. std::array<> 20](#_Toc6212747)

[5.4. Macro’s 23](#_Toc6212748)

[6. Abstracte Data Typen (ADTs) 25](#_Toc6212749)

[6.1. Getters en setters 25](#_Toc6212750)

[6.2. Operatoren 28](#_Toc6212751)

[6.3. Default constructor 30](#_Toc6212752)

[6.4. Default parameter waarden 31](#_Toc6212753)

[6.5. Meer operatoren 32](#_Toc6212754)

[7. Tekst uitvoer 37](#_Toc6212755)

[8. Unit tests 41](#_Toc6212756)

[8.1. Waarom testen? 41](#_Toc6212757)

[8.2. Unit tests 41](#_Toc6212758)

[8.3. Catch2 43](#_Toc6212759)

[9. Doxygen 46](#_Toc6212760)

[10. For, auto 49](#_Toc6212761)

[10.1. For 49](#_Toc6212762)

[10.2. Auto 51](#_Toc6212763)

[11. Overerving 53](#_Toc6212764)

[11.1. Een bewegende bal 53](#_Toc6212765)

[11.2. Overerving 54](#_Toc6212766)

[11.3. Virtual en abstract 58](#_Toc6212767)

[11.4. Pointers 61](#_Toc6212768)

[11.5. Stuiteren 62](#_Toc6212769)

[12. Dingen en waarden 65](#_Toc6212770)

[13. Micro-controllers 67](#_Toc6212771)

[13.1. Wat is een micro-controller 67](#_Toc6212772)

[13.2. Arduino 69](#_Toc6212773)

[13.1. De ontwikkelomgeving 69](#_Toc6212774)

[14. GPIO 70](#_Toc6212775)

[14.1. GPIO als output 70](#_Toc6212776)

[14.2. GPIO als input 73](#_Toc6212777)

[14.3. Een abstracte GPIO superklasse 76](#_Toc6212778)

[14.4. Namespace 77](#_Toc6212779)

[14.5. Hwlib 78](#_Toc6212780)

[14.6. Kitt 80](#_Toc6212781)

[15. SPI : 74HC595 83](#_Toc6212782)

[16. Decorator en adapter 88](#_Toc6212783)

[16.1. Patterns 88](#_Toc6212784)

[16.2. Decorator 88](#_Toc6212785)

[16.3. Adapter 93](#_Toc6212786)

[17. Hardware interfacing via I2C 96](#_Toc6212787)

[17.1. I2C 96](#_Toc6212788)

[17.2. OLED 98](#_Toc6212789)

[17.3. NVI 101](#_Toc6212790)

[1. Appendix: Weekschema 103](#_Toc6212791)

[2. Appendix : Verschillen tussen de C++ standaarden 104](#_Toc6212792)

[3. Appendix : Index en endnotes 105](#_Toc6212793)

# Inleiding

In de cursus V1OOPC worden de beginselen van het Object Oriented programmeren in C++ behandeld, en een aantal bijbehorende aspecten van software engineering, zoals het gebruik van UML diagrammen voor klassen, het testen op unit niveau met Catch2, en het documenteren van interfaces met Doxygen. Er wordt van uitgegaan dat de student in vorige cursussen ervaring heeft opgedaan met C++ (procedureel programmeren) en Python.

De programmeertaal C++ wordt gebruikt voor toepassingen waar de run-time snelheid (en soms ook het gebruik van andere resources, bijvoorbeeld geheugen) van doorslaggevend belang is. Zulke toepassingen kan je vinden bij ‘groot-verbruikers’ van computertijd als google, bij fast-traders voor wie een milliseconde het verschil tussen winst en verlies kan betekenen, en bij besturingssystemen (Windows, Linux) en in gadgets.

Het toepassingsgebied van C++ valt voor een groot deel samen met dat van C. Dat is niet toevallig: C++ is een poging om aan C een aantal moderne concepten toe te voegen (vooral Object Orientatie, Abstract Data Types, templates, en excepties) zonder concessies te doen aan de performance (snelheid en omvang van een applicatie).

Vergeleken met zogenaamde ‘gemanagde talen’ zoals Python of Java is C++ een lastigere taal om in te programmeren, omdat de programmeur zelf verantwoordelijk is voor veel details, met name geheugen management. C++ is bedoeld voor toepassingen waarbij de voordelen (performance) zwaarder wegen dan deze nadelen (lastiger te programmeren).

In voorgaande (en een gelijktijdige) cursussen heeft de student geleerd om te gaan met Python, procedureel C++, en de beginselen van UML en hardware interfacing. Deze cursus bereidt voor op de V1IPASS cursus, waarmee de propedeuse wordt afgesloten, en waarin de student het aangeleerde zelfstandig in de praktijk moeten brengen.

Er wordt 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, bmptk build-scripts en GCC compilers. (Er is beperkte ondersteuning voor werken op Linux systemen, dit zal meer zelfredzaamheid van de student vergen dan gebruik van Windows.)

De student moet zelf een Arduino Due en nog wat hardware (USB kabel, ‘prikbordje’, verbindingsdraadjes, OLED display) aanschaffen[[1]](#footnote-0). Deze hardware wordt ook in IPASS en een tweedejaars C++ cursus gebruikt. Bij tijdige aanschaf (via Aliexpress) zijn de totale kosten maximaal € 30. Details staan in een appendix. De gebruikte software tools en de Arduino Due sluiten aan bij de IPASS themaopdracht waarmee het propedeuse jaar wordt afgesloten en het aansluitende C++ vak in het tweede leerjaar.

De cursus wordt gegeven door middel van klassikale instructie (‘hoorcolleges’), klassikaal werken (‘werkcolleges’) en zelf werken (‘huiswerk’).

In deze cursus worden een aantal aspecten van C++ bewust weggelaten om de stof behapbaar te houden. Deze aspecten komen in volgende cursussen aan de orde. Het gaat hier met name om object lifetime (assignment, copy constructor, pointers, heap), templates, en excepties.

Er is in deze reader gekozen voor Nederlands voor de lopende tekst (om het voor de eerstejaars student zo leesbaar mogelijk te houden), en voor Engels voor de code voorbeelden (om de student 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 gebruikt voor hetzelfde ding (bv. bal versus ball, overerving versus inheritance).

De tekst van deze reader is geschreven om sequentieel gelezen te worden. Om een specifiek onderwerp terug te kunnen vinden kan je de Index raadplegen.

Voor C programmeurs zijn een aantal verschillen met C aangegeven in voetnoten. Dit is geen onderdeel van de lessstof.

Voor zover relevant wordt gebruik gemaakt van de mogelijkheden die de C++17 standaard biedt. Let op dat veel voorbeelden die je op het internet en in boeken vindt maken nog geen gebruik van (alle) C++17 features, waardoor ze er op details wat anders uit kunnen zien. In een appendix zijn de belangrijkste verschillen tussen de C++ versies samengevat. Soms worden de verschillen tussen C en C++ kort aangegeven. Dit behoort niet tot de lesstof.

# C++ Geschiedenis



Bjarne Stroustrup, inventor of C++

De taal C++ is vanaf 1979 ontwikkeld door Bjarne Stroustrup omdat hij de run-time snelheid van C wilde combineren met de Object Oriented features van Simula. Oorspronkelijk werd C++ gerealiseerd als een pre-compiler: de uitvoer van de C++ compiler was een C programma, dat door een C compiler werd compileerd tot een executable. Om niet een geheel nieuwe taal uit te moeten vinden en om het programmeurs met C ervaring makkelijk te maken C++ te gebruiken werd vrijwel de hele C taal (zoals die in 1979 in gebruik was) opgenomen in C++. Alle specifieke C++ eigenschappen kan je dus zien als uitbreidingen op C. Vandaar de naam C++: C, maar dan 1 stap verder. In latere jaren werd de relatie tussen C en C++ wat complexer, omdat aan beide talen nieuwe features werden toegevoegd, die daarna (soms) ook door de andere taal werden overgenomen. Maar nog steeds is het zo dat de meeste (realistische) C programma’s door een C++ compiler als legaal C++ gecompileerd zullen worden.

Het heeft lang geduurd voor er een officiele C++ standaard was (C++98). Daarvoor was een bepaald boek van Stroustroup de ‘de facto’[[2]](#footnote-1) standaard, maar iedere compiler implementeerde toch een iets andere versie van de taal. De eerste standaard in 2003 (C++03) bracht niet veel nieuws, eigenlijk alleen wat ‘bug fixes’. De 2011 standaard (C+11) bracht heel veel nieuwe features en verschoof de aandacht van een voornamelijk OO taal naar andere, op templates gebaseerd programmeerstijlen. De 2014 en 2017 standaarden brachten (in vergelijking met de veranderingen in C+11) wel interessante nieuwe features, maar geen grote veschuiving. De verwachting is dat de volgende versie van de standaard in 2020 zal uitkomen en weer een veel nieuwe features zal brengen die nieuwe programmeerstijlen mogelijk zullen maken.

De relatief snelle opeenvolgingen van C++ standaarden wordt door sommigen verwelkomd omdat het de programmeurs nieuwe mogelijkheden biedt, maar wordt door anderen verafschuwd omdat het betekent dat ze om een programma (dat de nieuwe features gebruikt) te lezen allerlei nieuwe features moeten leren, ook al willen ze die zelf niet gebruiken. In vergelijking met C++ is de taal C veel stabieler gebleven.[[3]](#footnote-2)

C en C++ zijn de belangrijkste talen voor het programmeren van applicaties die het uiterste vergen van de machine waar ze op draaien. Meestal betekent dit dat de applicatie ‘resource constrained’ is: er is een resource, die zo goed mogelijk benut moet worden. Vaak is dit CPU tijd, maar het kan ook geheugen zijn, of reactietijd. C en C++ bieden de programmeur vrijwel volledige controle over wat de computer waar hun programma op draait precies doet. Dit in tegenstelling tot ‘gemanagde’ talen als Python en Java, die de programmeur veel werk uit handen nemen (bv. geheugenbeheer), ten koste van geheugengebruik, run-time snelheid en voorspelbaarheid.

# Klassen en compositie



Piet Mondriaan: *Composition* in Brown and Gray

|  |
| --- |
| behandelde onderwerpen |
| * struct, class * private, public * constructor * hpp, cpp * associatie: compositie * reference |

## Een window

Bij het aanmaken van een window moet je de omvang in x en y richting meegeven en de schaal: omdat pixels op een PC scherm nogal klein zijn, worden de pixels met die schaal factor ‘opgeblazen’. De gebruikte waarden komen redelijk overeen met het schermpje dat we later gaan gebruiken met de Arduino Due.

|  |  |
| --- | --- |
| #include "hwlib.hpp"  #include "line.hpp"  int main(int argc, char \*\*argv){  hwlib::window w( 128, 64, 5 );  . . .  } | |
| Een window aanmaken |  |

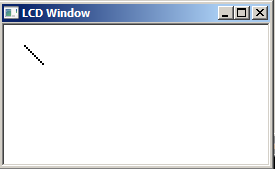
## Teken een lijn

Stel dat we in een window vormen willen afbeelden, te beginnen met een heel eenvoudige vorm: een lijn. Aan een lijn-print-functie moet je doorgeven wat hij precies moet doen, dit kan door de coördinaten van het beginpunt en eindpunt van de lijn mee te geven als 4 (integer) parameters.

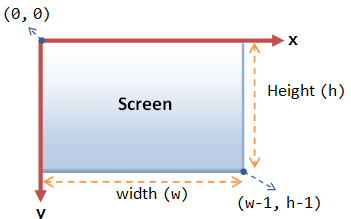
De naam van de functie (print\_line) geeft aan[[4]](#footnote-3) wat de functie doet. Beide delen zijn nuttig: als je ‘print’ weglaat weet je niet wat er met de lijn die je specificeert moet gebeuren (opslaan? uitwissen? roteren? ...); als je ‘line’ weglaat weet je niet wat die 4 getallen betekenen (een rechthoek? gemiddelde temperaturen van 4 dagen? …).

|  |  |
| --- | --- |
| void line\_print( window & w, int start\_x, int start\_y, int end\_x, int end\_y );  . . .[[5]](#footnote-4)  line\_print( w, 10, 10, 20, 20 ); | |
| specificeer een lijn via de x en y waarden van zijn begin- en eindpunten | 04-01[[6]](#footnote-5) |

Met deze code (en natuurlijk de bijbehorende implementatie van line\_print) wordt inderdaad een lijn geschreven in een window.



Bij een grafisch scherm is het de gewoonte dat (0,0) linksboven is, en dat de y coordinaten naar beneden lopen (dus anders dan in een grafiek).



Voor een window met size w (width, horiziontaal) bij h (hight, vertikaal) valt de pixel (0,0) dus net binnen een window, de pixel (w, h) valt er net buiten.

## Een lijn struct

Vaak is het handiger om alle gegevens van een lijn bij elkaar zetten, zodat ze als 1 variabele (of parameter) kunnen worden behandeld, net als alle bitjes van een integer. Dit kan door de gegevens, in dit geval de 4 coordinaat waarden, bijelkaar te plaatsen in een struct. Zo’n struct definieert een nieuw datatype[[7]](#footnote-6), in dit geval een lijn. Je dan lijn variabelen maken, lijn parameters doorgeven aan een functie, een functie kan een lijn waarde returnen, en je kan een array van lijnen maken.

Het volgende fragment laat zien hoe een line struct wordt gedefinieerd, en gebruikt. Je ziet dat de print functie nog maar 1 parameter heeft. Het ‘lijn’ deel van de functie naam kunnen we nu ook weglaten, want de parameter \*is\* een lijn.[[8]](#footnote-7)

|  |  |
| --- | --- |
| struct line {  int start\_x;  int start\_y;  int end\_x;  int end\_y;  };  void print( window & w, const line & x );  . . .  line diagonal\_line = { 10, 10, 40, 20 };  print( diagonal\_line ); | |
| specificeer een lijn via een lijn datatype | 04-02 |

Let op dat er je een struct afsluit met }; Als je de ; vergeet krijg je een foutmelding op de volgende C++ regel. Dat kan een mysterieuze melding zijn als die volgende regel in een andere file staat.

De functie print hoort echt bij de lijn struct. Dit kunnen we uitdrukken door de lijn struct en de print functie te combineren. We declareren dan de print functie in de struct in plaats van er buiten. Zo’n functie wordt een *member functie* of *methode* genoemd. Zo’n member functie wordt aangeroepen met de syntax “object.functie(…)”, waarbij het object impliciet wordt doorgegeven, in het volgende voorbeeld dus diagonal\_line.print(). In de parameter lijst van print komt de line dus niet meer voor.[[9]](#footnote-8)

|  |  |
| --- | --- |
| struct line {  int start\_x;  int start\_y;  int end\_x;  int end\_y;  void print( window & w )[[10]](#footnote-9);  };  . . .  line diagonal\_line = { 10, 10, 40, 40 };  diagonal\_line.print( w ); | |
| de print functie ondergebracht in het lijn datatype | 04-03 |

## Constructors

Als je een member functie schrijft met dezelfde naam als de struct dan is dit (per definitie) een constructor. Een constructor maakt zijn struct klaar voor gebruikt, vaak aan de hand van parameters die hij mee krijgt. In dit geval is de constructor bijna triviaal: de parameters worden gekopieerd naar de overeenkomstige data elementen van de lijn. Een constructor heeft geen return type (zelfs niet void).

Bij het aanmaken van de lijn wordt nu een iets andere syntax gebruikt: de constructor parameters wordt meegegeven bij het creëren van de variabele.[[11]](#footnote-10) Deconstructor geeft de initiële waarde aan de onderdelen van de variabele. Voor de window moet dit op een wat afwijkende manier omdat dit een reference variabele is.

|  |  |
| --- | --- |
| struct line {  window & w;  int start\_x;  int start\_y;  int end\_x;  int end\_y;  line( window & w, int p\_start\_x, int p\_start\_y, int p\_end\_x, int p\_end\_y ):  w( w )  {  start\_x = p\_start\_x;  start\_y = p\_start\_y;  end\_x = p\_end\_x;  end\_y = p\_end\_y;  }  void print();  };  . . .  line diagonal\_line( w, 10, 10, 50, 50 );  diagonal\_line.print(); | |
| initializeren dmv. een constructor | 04-04 |

De lijn heeft nu een constructor waarmee hij aangemaakt kan worden en een print functie die de lijn kan printen. Dat is eigenlijk alles wat een gebruiker moet weten: dat er in de lijn vier integers worden opgeslagen is voor de gebruiker niet van belang. We kunnen dit uitdrukken door de vier integers private te maken. Private geldt voor alle dingen die erna komen, dus we moeten nog wel zorgen dat de constructor en de print functie wel toegankelijk blijven. Dit doen we door er het label public voor te zetten. Code buiten de struct kan nu niet meer bij de attributen, maar de methoden van de struct nog wel.

In de line struct staat een window reference attribuut toegevoegd die in de initializatielijst van de constructor zijn waarde krijgt. Dit moet op die plaats, je kan een reference niet later (in de body) een waarde geven.

Merk op dat in de initializatie lijst bv. voor het initializeren van w de syntax w( w ) wordt gebruikt. Er zijn twee dingen die w heten: het attribuut en de parameter van de constructor. De compiler geeft de parameter voorrang, dus in principe betekent w de parameter, *behalve* bij de attribuut naam in een initializatie lijst: dat kan alleen maar een attribuut zijn, dus dan wordt w geinterpreteerd als de het attribuut w. Daardoor betekent w( w ) in een initializatielijst dus ‘initializeer het attribuut w (de eerste w) met de waarde van de parameter w (de tweede w)’.

|  |  |
| --- | --- |
| struct line {  private:  window & w;  int start\_x;  int start\_y;  int end\_x;  int end\_y;  public:  line( window & w, int p\_start\_x, int p\_start\_y, int p\_end\_x, int p\_end\_y ){  . . .  }  void print();  };  . . .  line diagonal\_line( w, 10, 10, 100, 30 );  // dit geeft nu een compilatie fout:  // error: 'int line::start\_x' is private  line.start\_x = 7;  diagonal\_line.print(); | |
| onderscheid tussen prive en publiek | 04-05 |

## Reference parameters en reference variabelen

Het is van belang dat het scherm object niet ‘by value’ wordt doorgegeven. Als je dat doet dan wordt er geprint worden op een kopie van het scherm, wat in veel gevallen niet het juiste resultaat zal leveren (het oorspronkelijke scherm object wordt dan niet veranderd). [[12]](#footnote-11)

Een reference declareer je door een & voor de naam van de parameter of object te plaatsen. Als je vervolgens die naam gebruikt in een expressie of om er een waarde aan toe te kennen, dan gebruik je automatisch het object waar de reference naar refereert.[[13]](#footnote-12)

Voor een reference moet je op het moment dat de reference wordt aangemaakt aangeven waarnaar hij refereert, en gedurende de lifetime van de reference blijft hij naar dat object refereren. Eer is geen mogelijkheid om een reference later naar een ander object te laten wijzen.

## Class

Het combineren van data en functies in een struct is legaal C++, maar wordt zelden gebruikt, omdat er in C++ een alternatief is voor struct: class. Het enige verschil tussen struct en class is dat de default (als je zelf niets aangeeft) bij een struct public is, en bij een class private. Maar het is een ingeburgerde (en daardoor aan te raden) gewoonte om class te gebruiken in plaats van struct als je onderdelen private maakt en/of functies binnen de class plaatst.

Deze vorm van programmeren wordt Object Oriented (OO) genoemd: in plaats van losse gegevens en functies worden de gegevens en de daarbij behorende functies gecombineerd in een klasse, en je programma maakt gebruik van objecten (instanties van klassen) die ieder van een bepaalde klasse zijn. Een klasse biedt zijn gebruikers toegang tot bepaalde functies en data onderdelen (het publieke deel), maar schermt andere onderdelen af (het prive deel). Een belangrijk voordeel hiervan is dat het prive deel gewijzigd kan worden zonder dat de rest van de applicatie hier iets van merkt. In ons geval zouden we bv. de coördinaten van een klein LCD scherm kunnen opslaan als unsigned char om ruimte te besparen, of in een ander geval als doubles als dat voor onze grafische processor beter uitkomt.

In UML geven we een klasse aan als een rechthoek met drie delen: de naam, de data elementen (attributen), en de functies. Een - voor een onderdeel geeft aan dat het onderdeel private is (niet voor de buitenwereld zichtbaar), een + geeft aan dat het public is. Het type van een attribuut en het return type van een functie staan achteraan. Een constructor herken je ook hier aan de naam (gelijk aan de naam van de klasse), en aan het ontbreken van een return type.

In de line class zijn de attributen (variabelen) private, en de methoden (functies) public. Dit is de meest voorkomende situatie (zeker voor eenvoudige klassen), maar private functies en (in mindere mate) public variabelen komen ook voor.

|  |
| --- |
|  |
| UML diagram van de lijn klasse |

|  |  |
| --- | --- |
| class line {  private:  window & w;  int start\_x;  int start\_y;  int end\_x;  int end\_y;  public:  line(window & w, int p\_start\_x, int p\_start\_y, int p\_end\_x, int p\_end\_y );  void print();  }; | |
| C++ declaratie van de lijn klasse | 04-06 |

## Verdeling over header en source files

De declaratie (interface) van een klasse staat vrijwel altijd in een header file: een klasse is bedoeld om in andere onderdelen van een applicatie gebruikt te worden en die andere onderdelen moeten de klasse declaratie kunnen includen. De definitie (implementatie) van een klasse functie kan in de klasse declaratie staan (dus in de header file), zoals we hebben gedaan bij de lijn constructor. Dit wordt over het algemeen alleen gedaan met heel simpele functies. Niet-triviale functies worden in de bijbehorende implementatie file gezet. Voor C++ wordt meestal de extensie .hpp gebruikt voor headers en .cpp voor implementatie files. Samengevat:

* Een header file heeft een een #ifndef/#define/#endif die zorgt dat er geen problemen ontstaan als een header meer dan 1 keer ge-include wordt,
* De header wordt file altijd ge-#include door de bijbehorende implementatie file.
* De header wordt file ge-#include door alle andere files die gebruik maken van de services die in de header file zijn gedeclareerd.

In code voorbeelden worden de #include regels vaak weggelaten.

|  |  |
| --- | --- |
| #ifndef LINE\_HPP  #define LINE\_HPP  class line {  . . .  };  #endif // LINE\_HPP | |
| bescherming tegen multiple-inclusion in de file line.hpp | 04-06 |

Voor de lijn klasse staat de implementatie van de print functie dus in de line.cpp file. Buiten de klasse declaratie moet je voor de naam van de functie zijn klasse naam en :: zetten, anders heb je het over een losse functie die niets met een klasse te maken heeft. Binnen de klasse definitie zelf hoeft dat niet.

|  |  |
| --- | --- |
| #include ”line.hpp”  void line::print(){  . . .  } | |
| implementatie van een member-functie (= methode) in de file line.cpp | 04-06 |

## Compositie

Een object kan samengesteld zijn uit deelobjecten. Als we een (niet opgevulde) rechthoek willen kunnen printen dan kunnen we dat doen door de vier lijnen die de omtrek van de rechthoek vormen te printen. De constructor kan deze lijn objecten aanmaken, en in de print functie hoeven we ze dan alleen maar te printen. Het rechthoek object bevat dan vier lijn objecten. In UML wordt dit aangegeven door een lijn met aan de kant van de ‘eigenaar’ een opgevuld wiebertje. Dit heet compositie: de rechthoek is samengesteld uit vier lijnen. Compositie (A bevat B, B is een deel van A) is de sterkste relatie (associatie) tussen klassen: de deelobjecten kunnen alleen bestaan als onderdeel van het omvattende object en dat omvattende object is verantwoordelijk voor het aanmaken van zijn deelobjecten.

Net als bij attributen geeft een + of – voor de naam van het deelobject aan of de compositie public (voor code buiten de klasse benaderbaar) of private (niet door vreemden benaderbaar) is. Een

|  |
| --- |
|  |
| UML diagram van een rechthoek die vier lijnen bevat |

In C++ zijn zijn de vier compositie relaties zichtbaar als attributen van de rechthoek, van het type lijn. Om dit te kunnen doen moet de declaratie van lijn (line.hpp) opgenomen worden in de definitie van rechthoek.

|  |  |
| --- | --- |
| #include ”line.hpp”  class rectangle {  private:  window & w;  line left, right, top, bottom;  public:  rectangle( window & w, int start\_x, int start\_y, int end\_x, int end\_y );  void print();  }; | |
| C++ declaratie van een rechthoek die vier lijnen bevat (file rectangle.hpp) | 04-07 |

In de rechthoek zijn nu vier lijnen opgenomen. In de declaratie van lijn is een constructor gedeclareerd. Dat betekent dat we bij het aanmaken van een lijn verplicht zijn deze constructor aan te roepen. Dit moet in de constructor van rechthoek, net voor de body (het stuk tussen de krulhaken), in de initializatie lijst. Daarmee zijn de lijnen geïnitialiseerd en er is niets meer te doen in de body zelf, dus die is leeg. Dit patroon komt vaak voor in C++: het meeste ‘werk’ in een constructor wordt gedaan in de initialisatielijst, en de body is vaak leeg.[[14]](#footnote-13)

|  |  |
| --- | --- |
| #include ”rectangle.hpp”  rectangle::rectangle( window & w, int start\_x, int start\_y, int end\_x, int end\_y ):  w( w ),  left( start\_x, start\_y, start\_x, end\_y + 1 ),  right( end\_x, start\_y, end\_x, end\_y + 1 ),  top( start\_x, start\_y, end\_x + 1, start\_y ),  bottom( start\_x, end\_y, end\_x + 1, end\_y )  {} | |
| de constructor van een rechthoek (file rectangle.cpp) | 04-07 |

Merk op dat er in de coordinaten soms +1 staat. Kan je beredeneren waarom?

Omdat de lijnen al geconstrueerd zijn in de constructor van rechthoek hoeft er in de print functie maar weinig gedaan te worden: de vier lijnen worden geprint.

|  |  |
| --- | --- |
| void rectangle::print(){  left.print();  right.print();  top.print();  bottom.print();  } | |
| de print functie van een rechthoek (ook in de file rectangle.cpp) | 04-07 |

Tot nu toe zijn we er aan voorbij gegaan waar een lijn of rechthoek eigenlijk op afgebeeld wordt. Dit kan worden aangegeven door een scherm object mee te geven. Van dit object interesseert ons (als programmeurs die scherm objecten willen implementeren) alleen de publieke interface. Die kan heel beperkt zijn: wis een scherm (maak het wit), en maak een pixel zwart.[[15]](#footnote-14)

|  |
| --- |
| D:\2017-2018\vakken\V1OOPC\images\classes\window.png |
| UML diagram van een window klasse |

De print functies moeten een scherm object hebben om hun werk te kunnen doen. Er zijn twee manieren om het scherm object ter beschikking te stellen:

* We kunnen een scherm object meegeven aan de constructor van een af te beelden object (bv. een lijn). Het scherm object wordt dan opgeslagen in het af te beelden object en is dus tijdens het printen beschikbaar.
* We kunnen een scherm object meegeven aan iedere print aanroep.

Welke oplossing er gekozen wordt is een kwestie van ontwerp. De eerste oplossing is (vaak) makkelijker, de tweede is flexibeler.

## Associatie

In UML komt een reference tussen objecten die ieder een eigen ‘leven’ hebben (het ene object is niet een logisch deel van het andere) doorgaans overeen met een ‘gewone’ associatie, weergegeven met een open wiebertje aan van de klasse die de reference bevat, en de naam van de referentie aan de kant van de klasse waar hij aan refereert. Als we in een lijn opslaan op welk scherm hij afgebeeld moet worden krijgen we dus de volgende UML diagram:

|  |
| --- |
|  |
| UML diagram voor een lijn die (een referentie naar) zijn window opslaat |

Merk op dat (net als bij een compositie) de window niet apart wordt vermeld in de lijst met attributen: de associatie met de naam w geeft aan dat het de lijn klasse een attribuut w van het type window heeft.

# const, std::array<>, macro’s

[](http://www.google.nl/url?sa=i%26rct=j%26q=%26esrc=s%26source=images%26cd=%26cad=rja%26uact=8%26ved=0ahUKEwiIoInN-djLAhVFng4KHXusAUIQjRwIBw%26url=http://www.mathend.com/c-const-left-or-right/%26psig=AFQjCNHTCKuJYZVzTxC8mf3nOBP9Kqvu-Q%26ust=1458896062938563)

|  |
| --- |
| behandelde onderwerpen |
| * const variabelen * const parameters * std::array<> * macro’s |

## const variabelen

Je kan je een variabele[[16]](#footnote-15) of parameter const declararen om aan te geven dat de waarde van dit object (na de initiele initializatie) niet meer zal veranderen. Dit maakt lezen van de code makkelijker, omdat de lezer zich bij het lezen van een expressie waarin de variabele wordt gebruikt niet hoeft af te vragen of de waarde in de tussentijd veranderd is. Daarnaast helpt het de compiler effeciëntere code te genereren.

|  |  |
| --- | --- |
| int main(){  const int answer = 42;    // Imaginge a long and very comple piece of code    std::cout << "The answer is " << answer << "\n";  } | |
| const: je kan er zeker van zijn dat de waarde niet veranderd | 05-01 |

## const parameters

Je kan const ook gebruiken om aan te geven dat een parameter van een functie binnen die functie niet veranderd zal worden. Het wijzigen van een gewone (niet-reference) parameter kan in een functie al geen gevolgen hebben voor de meegegeven variabele zelf, omdat er een kopie van wordt doorgegeven. Maar de functie kan die kopie wél wijzigen, wat voor verwarring kan zorgen als je de code leest.

|  |  |
| --- | --- |
| void f( int x ){  . . .  x = x + 5; // did the author realize that he is modifying a copy?  . . .  }  . . .  int i = 10;  f( i );  // i is still 10 | |
| je kan toekennen (assignen) aan de parameter van een functie, maar bedoel je dat ook? | 05-02 |

Als je een functie parameter als const declareert dwing je af dat die parameter in de functie niet gewijzigd wordt. Dit maakt het lezen van je code makkelijker.[[17]](#footnote-16)

|  |  |
| --- | --- |
| void f( const int x ){  . .  x = x + 5; // compilation error  . . .  } | |
| een const parameter kan je niet wijzigen | 05-03 |

In C++ wordt een parameter standaard ‘by copy’ overgedragen. Als een object groter is dan een paar bytes is dat niet efficient: er moet ruimte gemaakt worden voor de kopie, de inhoud moet gecopieerd worden, en uiteindelijk moet de ruimte ook weer vrijgegeven worden. En dat terwijl de functie body ook gewoon gebruik had kunnen maken van het originele object. Het is dus (voor objecten van enige omvang) efficienter om ‘by reference’ over te dragen. Maar dat heeft twee naadelen:

* De functie kan nu niet meer worden aangeroepen met een const object of met een expressie.
* De aangeroepen functie kan het object nu wél wijzigen.

|  |  |
| --- | --- |
| void f( int & x ){}  const int x = 10;  f( x ); // compilation error  f( 3 + 5 ); // compilation error | |
| Je kan geen const of expressie meegeven aan een reference parameter | 05-04 |

Beide problemen worden ondervangen door de parameter ‘by const reference’ over te dragen. De aangeroepen functie krijgt dan een reference naar het object (een reference heeft dezelfde omvang als een pointer), maar kan via die reference het object niet wijzigen.

|  |  |
| --- | --- |
| void f( const int & x ){}  void g( const int & x ){  x = 7; // compilation error  }  int main(){  const int x = 10;  f( x ); // OK  f( 3 + 5 ); // OK  } | |
| een object doorgeven by const reference | 05-05 |

Samenvattend:

* de eerste keuze voor het doorgeven van een parameter is ‘by const reference’. Reference omdat zo wordt voorkomen dat er een copie wordt gemaakt. Const omdat je dan constanten en expressies kan meegeven, en je in de functie zelf niet ‘per ongeluk’ de waarde kan veranderen.
* Als een object zo klein is dat een copie doorgeven efficienter is dan het doorgeven van een reference dan wordt de reference vaak weggegelaten. Dit is bv. het geval bij (losse) integers en characters.
* Als het doel van de functie is om de waarde van een parameter aan te passen dan moet je die parameter uiteraard by reference doorgeven (dus zonder const).

## std::array<>

Een std::array<> bevat een aantal waarden van het zelfde type. Anders dan bij een std::vector<> moet je aangeven wat het aantal is, en dit aantal kan gedurende de levensduur van het array niet veranderen. Dit is een flinke beperking, maar daar staat tegenover dat het allokeren en benaderen van een std::array<> sneller gaat dan een std::vector<>. Op een klein embedded systeem is het vaak niet mogelijk een std::vector<> te gebruiken omdat die gebruik maakt van de heap, en die is op zo’n systeem vaak niet aanwezig. std::array<> is dan dus het enige alternatief.

|  |  |
| --- | --- |
| #include <array>  int main(){  std::array< int, 10 > a;    for( unsigned int i = 0; i < a.size(); i++ ){  a[ i ] = i \* i;  }    for( unsigned int i = 0; i < a.size(); i++ ){  std::cout << "a[ " << i << " ]= " << a[ i ] << "\n";  }  } | |
| Gebruik van std::array<> | 05-07 |

De omvang (aantal elementen) van een std::array<> moet je expliciet aangeven. Dat kan je doen door het getal (bv. 10 in het vorige voorbeeld) daar op te nemen, maar het is beter een const te maken, dan wordt je gedwongen een zinnige naam te geven aan het aantal, en als je het op meerdere plekken moet gebruiken loop je niet de kans dat het getal op de ene plek wordt veranderd en op een andere plek niet. [[18]](#footnote-17)

|  |  |
| --- | --- |
| const unsigned int squares\_table\_size = 10;  int main(){  std::array< int, squares\_table\_size > squares;    for( unsigned int i = 0; i < squares\_table\_size; i++ ){  squares[ i ] = i \* i;  }    for( unsigned int i = 0; i < squares.size(); i++ ){  std::cout << "squares[ " << i << " ]= " << squares[ i ] << "\n";  }  } | |
| Een const als std::array size | 05-08 |

Als je een std::array<> als parameter wil gebruiken, dan moet wat je meegeeft precies het zelfde zijn als de parameter, dus zowel het type van de elementen als het aantal elementen. Als dat niet zo is krijg je een compiler error.

|  |  |
| --- | --- |
| const unsigned int word\_size = 8;  void print( const std::array< char, word\_size > & word ){  for( unsigned int i = 0; i < word.size(); i++ ) {  std::cout<< word[ i ];  }  }  int main(){  std::array< char, word\_size > word = { 'P', 'a', 't', 'a', 't', 'j', 'e', '!' };  print( word );    std::array< char, 5 > short\_word = { 'P', 'a', 't', 'a', 't' };  print( short\_word ); // compiler error  } | |
| Een std::array<> als parameter | 05-09 |

Als je een std::array<> doorgeeft, maar niet alle elementen zijn geldig, dan moet je dit op een andere manier doorgeven, bv. met een extra parameter doorgeven, of door speciale waarden te gebruiken om ongeldige elementen te markeren.

|  |  |
| --- | --- |
| const unsigned int word\_size = 8;  void print(  const std::array< char, word\_size > & word,  unsigned int valid  ){  for( unsigned int i = 0; i < valid; i++ ) {  std::cout<< word[ i ];  }  }  int main(){  std::array< char, word\_size > word = { 'P', 'a', 't', 'a', 't', 'j', 'e', '!' };  print( word, 8 );    std::array< char, word\_size > short\_word = { 'P', 'a', 't', 'a', 't' };  print( short\_word, 5 );  } | |
| Gebruik van een extra parameter om de geldige lengte door te geven | 05-10 |

Soms is het mogelijk een speciale waarde te gebruiken om aan te geven dat een element (en de elementen die daar na komen) niet geldig zijn. Dit wordt vaak een ‘ sentinel’ genoemd. Voor ASCII characters is het ASCII nul character ( ‘\0’ ) een veel gebruikte sentinel. Dit is zelfs in de taal ingebouwd: een string literal tussen dubbel-quotes is de reeks characters tussen de quotes, met een ASCII nul er achter. De string ”hello” neemt dus 6 bytes in beslag: de 5 letters plus de nul.

|  |  |
| --- | --- |
| void print( const std::array< char, word\_size > & word ){  for( unsigned int i = 0; i < word.size(); i++ ) {  if( word[ i ] == '\0' ){  return;  }  std::cout<< word[ i ];  }  }  int main(){  std::array< char, 8 > short\_word = { 'P', 'a', 't', 'a', 't', '\0' };  print( short\_word );  } | |
| Gebruik van een sentinel | 05-11 |

## Macro’s

In C was het lang de gewoonte om vaste getallen als #define macro te declareren[[19]](#footnote-18), en deze mogelijkheid bestaat ook nog in C++. Macro’s zijn echter vervelende dingen: ze zijn een letterlijke (textuele) vervbanging, en houden zich niet aan de normale zichtbaarheids regels: je kan in een functie geen lokale macro definieren. In onderstaande code gaat dit (daardoor) mis.

|  |  |
| --- | --- |
| #define ARRAY\_SIZE 100  int array[ ARRAY\_SIZE ];  void f(){  #undef ARRAY\_SIZE  #define ARRAY\_SIZE 20  char my\_array[ ARRAY\_SIZE ];  . . .  }  for( unsigned int i = 0; i < ARRAY\_SIZE; i++ ){ // wrong ARRAY\_SIZE  array[ i ] = 0;  } | |
| Oude C stijl: gebruik van macro’s als constanten | 05-12 |

Om het wat afwijkende karakter van macro’s aan te geven is het een gewoonte macro namen in HOOFDLETTERS te schrijven.

In C++ wordt in zo’n geval een const declaratie gebruikt: een variabele die bij zijn declaratie een waarde krijgt, die daarna niet meer veranderd kan worden. Zo’n const variabele houdt zich wel aan de zichtbaarheidsregels, net als iedere andere variabele.

|  |  |
| --- | --- |
| const unsigned int array\_size = 1000;  int array[ array\_size ];  void f(){  const int array\_size = 20;  char my\_array[ array\_size ];  . . .  }  for( unsigned int i = 0; i < array\_size; i++ ){  array[ i ] = 0;  } | |
| C++ stijl: gebruik const voor constante waarden | 05-13 |

# Abstracte Data Typen (ADTs)



|  |
| --- |
| behandelde onderwerpen |
| * ADTs * Public, of get/set * Operatoren * Overloading, meerdere constructors * Default parameter waarden * Operatoren als methoden of als losse functies * Lijst van operatoren en hun prioriteiten * Vuistregels voor user-defined operatoren |

Een Abstract Data Type (ADT) is door de gebruiker geschreven datatype dat zich zo veel mogelijk gedraagt en gebruikt kan worden als een van de ingeboude datatypen (int, float, etc.), en daarbij zijn implementatie verbergt.

## Getters en setters

Tot nu toe hebben we om een plaats op een scherm aan te geven steeds twee integer waarden gebruikt. Dit leidt tot lange parameter en attribuut lijsten, en als we willen rekenen met zulke coordinaten (bv. twee coordinaten bij elkaar optellen) dan moeten we aparte bewerkingen doen op de x en y elementen.

|  |  |
| --- | --- |
| class line {  private:  . . .  int start\_x;  int start\_y;  int end\_x;  int end\_y;  public:  . . .  // move the line  void shift( int x, int y ){  start\_x += x;  start\_y += y;  end\_x += x;  end\_y += y;  }  }; | |
| een lijn klasse met een shift() functie | 06-01 |

Het is in zo’n geval handig om een xy (2d int vector) klasse te maken voor een plek op het scherm, die een x en y coordinaat bevat. Dan bevat een lijn twee van zulke xy vectoren: het startpunt en het eindpunt van de lijn.

|  |
| --- |
|  |
| UML voor een lijn klasse die twee xy waarden bevat |

|  |
| --- |
| class xy {  private:  int x;  int y;  public:  xy( int x, int y ):  x( x ), y( y )  {}  };  class line {  private:  xy start;  xy end;  . . .  }; |
| een xy klasse, en een lijn klasse met xy waarden ipv. losse integers |

Omdat de x en y private zijn moeten we nog wat aanpassingen maken om er nuttig gebruik van te kunnen maken. Een manier is om zogenaamde getter en setter methoden toe te voegen: klasse functies die als enige taak hebben de waarde van een prive attribuut op te halen of te wijzigen.

|  |  |
| --- | --- |
| class xy {  private:  int x;  int y;  public:  . . .  int x\_get(){ return x; }  int y\_get(){ return y; }  void x\_set( int new\_x ){ x = new\_x; }  void y\_set( int new\_y ){ y = new\_y; }  };  xy a( 1, 2 ), b( 3, 4 );  // add b to a  a.x\_set( a.x\_get() + b.x\_get() );  a.y\_set( a.y\_get() + b.y\_get() ); | |
| xy vectoren bij elkaar optellen door middel van getters en setters | 06-02 |

Een andere optie is om de attributen x en y publiek toegankelijk te maken. Dit maakt werken met een locatie object veel makkelijker, maar het dataype is nu wel een belangrijk deel van zijn ‘abstractie’ kwijt:

* We kunnen nu niet meer de implementatie van de klasse veranderen (bv. van twee integers naar carthesische coordinaten: een hoek en een afstand) zonder dat dit gevolgen heeft voor de gebruikers van de klasse.
* We kunnen geen samenhangt tussen de attributen meer garanderen. Als we bv. aan een ‘kleine’ xy de eis willen stellen dat de afstand tot de oorsprong maximaal 100 is, dan zouden we die met x\_set() en y\_set functies kunnen afdwingen: als de afstand groter zou worden, dan wordt de waarde niet veranderd, of er wordt een foutmelding gegegeven. Met publieke attributen kan dit niet meer.

De keuze tussen ofwel publieke attributen, ofwel getters en setters, is lastig. Er zijn in de software engineering verschillende meningen hierover, varierend van ‘altijd alle attributen public maken’ (dit is in feite zo in Python) tot ‘altijd getters en setters gebruiken’ (deze mening komt veel voor onder bv. Java programmeurs). Als je in een getter of setter meer zou doen dan alleen de waarde ophalen of wegschrijven, dan is het duidelijk dat je geen publieke attributen moet gebruiken. Als dat niet zo is, dan moet je het gemak van publieke attributen afwegen tegen de toekomstige flexibiliteit die getters en setters bieden. Helaas laat de toekomst zich lastig voorspellen.

Voor de xy klasse is besloten om de x en y attributen publiek te maken, omdat

* er in de getters/setters niets anders gebeurt dan de waarde opvragen en wegschrijven, en
* de kans dat er ooit overgeschakeld zal worden op een andere representatie te verwaarlozen is, want de pixels op een klein grafisch scherm worden altijd aangeduid met integer coordinaten.[[20]](#footnote-19)

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  . . .  };  xy a( 1, 2 ), b( 3, 4 );  // add b to a  a.x += b.x;  a.y += b.y; | |
| xy waarden bij elkaar optellen via publieke attributen | 06-03 |

## Operatoren

In de vorige code moesten we voor het optellen van twee lokaties twee statements gebruiken, één voor de x en één voor de y. We kunnen dit makkelijker maken, want in C++ zijn operatoren zoals + eigenlijk gewoon methoden (klasse functies). Als een passende operator+ is geclareerd zijn de volgende twee expressies legaal in C++ en betekenen exact hetzelfde:[[21]](#footnote-20)

|  |
| --- |
| a + b  a.operator+( b ) |
| twee manieren om dezelfde operator+ methode aan te roepen |

Als we dus twee lokaties bij elkaar op willen kunnen tellen dan moeten we daar de passende **operator+** voor schrijven.

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  . . .  xy operator+( const xy & rhs ) const {  return xy( x + rhs.x, y + rhs.y );  }  };  xy a( 1, 2 ), b( 3, 4 );  // calculations with xy’s  a = a + b; | |
| een operator+ te maken | 06-04 |

De operator+ heeft twee parameters zijn van het type xy: de eerste (linker) parameter is het object zelf, de tweede (rechter) is hier rhs (Right Hand Side) genoemd. De rhs parameter wordt overgedragen by reference (de &), en meer specifiek by const reference: de operator+ belooft plechtig dat hij die parameter niet zal wijzigen, en de compiler zal hem daaraan houden. Daardoor kan de operator ook aangeroepen worden met als argument een constante of een expressie.

Een + is symmetrisch, dus we willen dat de lhs (Left Hand Side, het object zelf dus) ook ‘by const reference’ moet worden overgedragen, maar die lhs zit niet in de parameter lijst, dat is het object zelf. Dit object wordt (als er niets anders is aangegeven) in feite ‘by reference’ overgedragen: de functies van de klasse kunnen de variabelen in het object lezen en schrijven. Als we willen dat de functies van de klasse de variabelen van de klasse kunnen lezen *maar niet schrijven*, dan moeten we het object ook als ‘const’ markeren. Volgens de C++ syntax moet dat meteen ná de parameterlijst (voor de { open-krulhaak).

De body van de operator is bijna triviaal: het resultaat van de functie is een waarde van het type xy. Die waarde construeren we door de xy constructor aan te roepen met de juiste x en y waarden.

Behalve de tweezijdige (diadic) + is er ook een eenzijdige (monadic) +. Dit is een veel minder nuttige operator, die er eigenlijk alleen is omdat de eenzijdige - wél een nuttige operator is. De eenzijdige operatoren hebben geen rhs (de enige parameter is het object zelf, dat impliciet wordt doorgegeven), dus hun parameterlijst is leeg.

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  . . .  Xy xyr+() const {  return xy( x, y );  }  };  vector a( 1, 2 ), b( 3, 4 );  a = + b; | |
| een eenzijdige (monadic) operator+ | 06-05 |

Een xy vector is natuurlijk niet het enige data type waarvoor je een optel-operator zou willen hebben. (En met de diadic en monadic operatoren heeft xy er al twee!) Dat leidt er toe dat er veel operator+ functies kunnen zijn. In C++ is het geen probleem als er meerdere functies met dezelfde naam zijn, mits de compiler bij iedere aanroep van een functie aan het aantal en de typen van de parameters kan zien welke functie er wordt aangeroepen.[[22]](#footnote-21) In het volgende stukje code worden drie + operatoren aangeroepen. De eerste heeft twee parameters van het type xy dus de aangeroepen functie is de zojuist beschreven ( xy + xy ). De tweede aanroep heeft twee parameters van het type int, dus de aangeroepen functie is de ingebouwde ( int + int ) operator. De derde aanroep heeft parameters van het type xy en int. Tenzij iemand een dergelijke functie gemaakt heeft zal de compiler hier een foutmelding geven omdat hij geen passende operator+ kan vinden.

|  |  |
| --- | --- |
| xy a( 1, 2 ), b( 3, 4 ), c( 5, 6 );  int x = 1, y = 2, z;  c = a + b;  z = x + y;  z = x + b; // compilation error | |
| drie verschillende + operatoren naast elkaar | 06-06 |

Functies met dezelfde naam worden ‘overloaded’ genoemd. Overloading is in C++ mogelijk voor alle functies en methoden. Het wordt veel gebruikt bij operatoren en bij constructors.

## Default constructor

Tot nu toe hebben we bij het aanmaken van een xy steeds aangegeven wat de (initiele) waarde is. Stel dat we dat niet willen doen: we willen gewoon een xy variabele aanmaken, zonder initiele waarde. Met de xy decaratie die we tot nu toe hadden kan dat niet, want voor de compiler is het aanmaken van een variabele het aanroepen van een constructor, en als je bij het aanmaken geen waarden meegeeft is dat dus een constructor die zonder parameters wordt aangeroepen, de zgn. default constructor. En die is er in ons geval niet. Maar zoals zo vaak in C++ is dat geen probleem: (bijna) alles kan, maar je moet het wel zelf doen. In dit geval kunnen we een constructor toevoegen zonder parameters. (We overloaden dan dus de constructor.) Die nieuwe constructor hoeft niets te doen, zowel de initializatielijst als de body zijn dus leeg. De attributen van een xy die met deze constructor is aangemaakt hebben dus geen gedefinieerde waarde, net als een (lokale) integer of float die je aanmaakt.

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  xy( int x, int y ):  x( x ), y( y )  {}  xy(){}  };  xy a;  xy b( 1, 2 ); | |
| Een xy klasse met twee constructors | 06-07 |

Een object met ongedefinineerde waarden is tijdens het runnen van je programma snel aangemaakt, maar als je die waarden per ongeluk gebruikt vertoont je programma ‘undefined behaviour’[[23]](#footnote-22). Dit kan je (ten koste van wat snelheid) voorkomen door in de constructor de x en y wél een waarde te geven. Welke waarde doet er niet zo veel toe, meestal wordt 0 gebruikt.[[24]](#footnote-23)

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  xy( int x, int y ):  x( x ), y( y )  {}  xy():  x( 0 ), y( 0 )  {}  }; | |
| deze default constructor zet de attributen wel op een waarde | 06-08 |

## Default parameter waarden

We hebben nu twee constructors, maar dat is niet nodig als we een andere C++ feature gebruiken. We kunnen in de declaratie van een functie voor een parameter een default waarden aangeven. Bij het aanroepen van zo’n functie mogen de parameters die een default waarde hebben weggelaten worden. Dan worden van links naar rechts de parameters die wél worden meegegeven gebruikt, en voor de overige parameters worden de default waarden gebruikt.

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  xy( int x = 0, int y = 0 ):  x( x ), y( y )  {}  };  xy a;  xy b( 1 );  xy c( 2, 3 ); | |
| een constructor met default parameters | 06-09 |

Het heeft dus geen zin om na een parameter met een default waarde nog een parameter te hebben die geen default waarde heeft[[25]](#footnote-24), en dit mag dus ook niet.

Een verschil tussen de twee aparte constructors en een enkele contructor met twee default parameters is dat je die enkele constructor ook kan aanroepen met één parameter. Of dit wenselijk is moet je als auteur van de klasse zelf beoordelen.

Als je een constructor zonder parameters definieert en ook nog een constructor met voor alle parameters een default, dan zijn er twee constructors die beide zonder parameters kunnen worden aangeroepen. Als je dat doet krijg je bij zo’n aanroep[[26]](#footnote-25) een foutmelding van de compiler die aangeeft dat hij niet weet welke van de twee hij moet aanroepen. Zoals eerder gezegd, geldt dit principe ook voor andere (niet-constructor) functies: pas bij het aanroepen bepaalt de compiler of hij, aan de hand van de naam en de paraneters, precies weet welke functie er aangeroepen moet worden. Zijn er meerdere kandidaten, dan volgt een foutmelding.

Default waarden voor parameters moet je aangeven bij de declaratie van een functie (of methode). Als je een aparte definitie hebt (vaak in een andere file), dan herhaal je de default waarden daar niet. De achterliggende reden is dat de default waarden iets te maken hebben met het aanroepen van een functie (en dat doet de compiler met alleen kennis van de declaratie), en niet met de implementatie van de functie (die ziet alleen de waarden van de parameters, niet of die afkomstig zijn van een default of van een door de aanroeper meegegeven waarde).

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  xy( int x = 0, int y = 0 );  };  xy::xy( int x, int y ):  x( x ), y( y )  {} | |
| default parameters staan in de declaratie, niet in de definitie | 06-09 |

## Meer operatoren

We hebben gezien hoe een operator+ en een operator- voor xy’s kan worden gemaakt. Van een echte ADT verwacht de gebruiker veel meer operatoren. Als een ADT een operator+ en een operator= (assignment) heeft, dan verwacht je ook een operator+=. Die moet, anders dan een operator+, zijn linker argument wijzigen. Dat argument moet dus ‘by reference’ worden doorgegeven (dat is automatisch zo) maar anders dan bij de operator+ niet ‘als const’. De C++ gewoonte is dat het resultaat van een assignment-achtige operator (=, +=, -=, etc.) een reference is naar het object zelf. Anders dus dan bij de operator+, daar is het resultaat een waarde. Het rechter argument (rhs) wordt voor beide operatoren op de zelfde wijze doorgegeven: by const reference.

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  xy operator+( const xy & rhs ) const;  xy & operator+=( const xy & rhs );  }; | |
| een xy klasse met operatoren + en += | 06-10 |

De implementatie van de operator+= is simpel: tel de x en y waarden van de rhs op bij de x en y attributen van het object. Daarna moeten we het object zelf returnen. De C++ feature die dat mogelijk maakt is de this pointer, die wijst steeds naar het object. Wat we moeten returnen is niet een pointer naar het object maar het object zelf, dus we returnen \*this.[[27]](#footnote-26)

|  |  |
| --- | --- |
| xy & xy::operator+=( const xy & rhs ){  x += rhs.x;  y += rhs.y;  return \*this;  } | |
| implementatie van operator+= | 06-10 |

De operator+ en operator+= doen gedeeltelijk hetzelfde: twee xy waarden optellen. Het verschil is waar ze het resultaat laten: += bergt het op in de lhs, + geeft de nieuwe waarde terug als functie resultaat. Als wat zo’n koppel operatoren doet wat minder triviaal is dan is het een goed idee om dit maar één keer op te schrijven.[[28]](#footnote-27) De eenvoudigste manier is om de operator+= te implementeren zoals we al zagen, en dan de operator+ te implementeren door een kopie te maken van onze eigen waarde, daar de operatror+= op los te laten, en dan die kopie te returnen. Dit lijkt misschien wat omslachtig, maar de winst is dat deze drie regels helemaal onafhankelijk zijn van wat ‘optellen’ voor de klasse precies inhoudt: deze drie regels vind je dus in vrijwel alle operator+ implementaties.

|  |  |
| --- | --- |
| xy xy::operator+( const xy & rhs ) const {  xy temp = \*this;  temp += rhs;  return temp;  } | |
| implementatie van operator+ door gebruik te maken van operator+= | 06-10 |

Je kunt veel (maar niet alle) operatoren naar keuze als methode (klasse functie) schrijven of als ‘vrije’ functie (los van een klasse). Het voordeel van een methode is dat die automatisch bij de private attributen van de klasse kan, een losse functie kan dat niet. Maar een losse functie kan je schrijven voor een klasse waarvan je de declaratie niet kan aanpassen, bv. omdat die in een bibliotheek zit die jij niet kan wijzigen. De vuistregel is dat je een operator als methode schrijft, tenzij dat niet kan.

|  |  |
| --- | --- |
| class xy {  public:  int x;  int y;  . . .  xy operator+( const xy & rhs ) const {  return xy( x + rhs.x, y + rhs.y );  }  };  xy operator-( const xy & lhs, const xy & rhs ){  return xy( lhs.x - rhs.x, lhs.y + rhs.y );  } | |
| een methode operator+ en een losse functie operator- | 06-11 |

In C++ kunnen heel veel operatoren door de gebruiker worden gedefinieerd, maar de lijst van operatoren ligt vast, en hun onderlinge prioriteit ook. Je kunt dus geen nieuwe operator $ of ^^ maken. De volgende tabel geeft de belangrijkste operatoren die je in C++ zelf kan definiëren. Slechts een klein deel hiervan zal in deze cursus behandeld worden.

De meeste operatoren kan je op twee manieren schrijven:

* als losse functie, alle parameters zijn dan expliciet in de parameterlijst vermeldt, of
* als klasse functie (methode), de eerste parameter is dan het object zelf, en alleen de parameters die daarna komen staan expliciet in de parameterlijst.[[29]](#footnote-28)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C++ Operatoren | | | | |
| Gebruik | Function syntax | Method syntax | Algemene betekenis |  |
| + a | operator+( a ) | operator+() | Unary plus | Evenzo voor - resp. -- |
| a + b | operator+( a, b ) | operator+( b ) | Binary plus |
| a += b | operator+=( a, b ) | operator+=( b ) | Plus-assignment |
| a ++ | operator++( a ) | operator++() | Postfix increment (geeft de oude waarde terug) |
| ++ a | operator++( a, x )[[30]](#footnote-29) | operator++( x ) | Prefix increment (geeft de nieuwe waarde terug) |
| a \* b | operator\*( a, b ) | operator\*( b ) | Vermenigvuldiging | Evenzo voor / % <<>>[[31]](#footnote-30)& | ^ |
| a \*= b | operator\*=( a, b ) | operator\*=( b ) | Vermenigvuldiging-assignment |
| a || b | operator||( a, b ) | operator||( b ) | Logische or (zie opmerking over short-circuit character) | Evenzo voor && |
| ! a | operator!( a ) | operator!() | ontkenning |  |
| a < b | operator<( a, b ) | operator<( b ) | Kleiner dan | Evenzo voor <= > >= == != |
| a = b | [[32]](#footnote-31) | operator=( b ) | Assignment |  |
| a[ … ] | 32 | operator[]( … ) | Array element selectie |  |
| a( … ) | 32 | operator()( … ) | Functie aanroep |  |
| \* a | operator\*( a ) | operator\*() | Pointer dereference |  |
| & a | operator&( a ) | operator&() | Adres van |  |
| a -> | 32 | operator->() | Pointer dereference plus . |  |

Aangezien een operator niet zoals een methode of functie een lange, betekenisvolle naam kan hebben is het erg belangrijk dat de gebruiker van een operator krijgt wat hij verwacht. Dit beperkt het gebruikt van operatoren tot bv. dingen waar je mee kan rekenen, zoals bv. meetwaarden, xy waarden, en imaginaire getallen. Twee strings bij elkaar optellen heeft voor de meeste mensen ook nog wel een intuïtieve betekenis, en een string met een integer vermenigvuldigen eventueel ook nog (s \* 5 is dan het zelfde als s + s + s + s + s). Maar strings van elkaar aftrekken of een string delen door een getal is voor de bijna alle argeloze lezers van zo’n stukje code betekenisloos. De operatoren << en >> hebben een standaard betekenis (tekst-output en -input) die nog besproken wordt.

De ingebouwde && en || operatoren (logische *and* en *or*) hebben een short-circuit eigenschap. Eerst wordt de linker waarde bepaald. Als met die waarde het resultaat van de operator al vast staat (voor welke combinaties van operatoren en waarden is dat zo?) dan wordt de rechter waarde niet meer uitgerekend. Dit kan van groot belang zijn als het bepalen van die rechter waarde een pobleem zou opleveren. Een door de gebruiker gedefinieerde && of || operator heeft nooit deze short-circuit eigenschap: beide waarden worden eerst uitgerekend, en dan pas wordt de (door de gebruiker gedefinieerde) operator functie aangeroepen.

Na al die waarschuwingen vraag je je misschien af of het überhaupt wel verstandig is om zelf operatoren te definiëren. Een paar vuistregels:

* Voor alle typen die redelijkerwijs als tekst weergegeven kunnen worden is een operator<< nuttig, al is het alleen maar voor debuggen. Een operator>> is dan wellicht ook nuttig (als het redelijkerwijs mogelijk is het type van een console of een tekst file in te lezen).[[33]](#footnote-32)
* Voor typen waar je mee kan rekenen is het heel handig om de reken-operatoren ( + - \* / ) te definiëren. Een dergelijk type kan je meestal ook ordenen (vergelijken), dus definieer dan ook de vergelijkings-operatoren ( ><>= <= == != ). De operator= zal je voor dit soort typen vaak niet zelf hoeven te definiëren omdat de automatisch door de compiler aangemaakte operator= voldoet.
* Zorg dat je operatoren zich op dezelfde manier tot elkaar verhouden als de integer operatoren: als je operator+ een 1 accepteert als rechter argument, definieer dan ook de operatoren voor ++a en a++, zodanig dat die ook 1 bij a optellen, en laat a++ de oude waarde van a teruggeven, en ++a de nieuwe waarde. Als je een operator+ definieert, maak dan ook een operator+=, zodanig dat net als bij integers de volgende twee statements het zelfde effect hebben:

|  |
| --- |
| a = a + b;  a += b; |

* De array element selectie (operator[]) kan gebruikt worden om een onderdeel dat opgeslagen is op te vragen.
* Het definiëren van de short-circuit operatoren && en || ligt niet voor de hand omdat die dan niet het short-circuit karakter kunnen hebben dat een naieve lezer ervan verwacht.
* De functie aanroep en de adres en dereference operatoren worden veel gebruikt in containers en zgn. smart pointers. Het is goed om te weten dat in zulke gevallen gebruik wordt gemaakt van deze mechanismen, maar het zelf schrijven van zulke constructies (behalve om het even uit te proberen) kan je beter overlaten aan de zeer ervaren programmeurs die zulke libaries voor je hebben geschreven.

# Tekst uitvoer

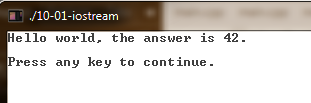


Text uitvoer, bestaat er dan nog iets anders?

|  |
| --- |
| behandelde onderwerpen |
| * tekst uitvoer met behulp van operator<< * zelf schrijven van operator<< * ostream manipulators |

De standaard tekst output in C++ is std::cout, die is gedeclareerd in <iostream>. Het type van std::cout is std::ostream. Tekstuele output in C++ wordt gedaan door middel van de infix operator <<. Er zijn << operatoren beschikbaar om alle standaard typen ‘naar een ostream te schuiven’.

|  |  |
| --- | --- |
| #include <iostream>  std::cout << "Hello world, the answer is " << 42 << '.' << '\n'; | |
| gebruik van std::ostream en diverse operators << | 08-01 |

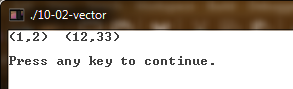


De waarden worden van links naar rechts naar de uitvoer geschreven, omdat de << operatoren (die allemaal zelfde prioriteit hebben) van links naar rechts ‘binden’. De vorige regel code moet dus eigen zo gelezen worden:

|  |
| --- |
| ((( std::cout << "Hello world, the answer is " ) << 42 ) << '.' ) << '\n'; |
| De zelfde expressie maar met haakjes |

Dit werkt omdat (volgens afspraak) alle << operatoren zo geschreven worden dat ze hun linker argument als resultaat teruggeven. Voor onze xy is er nog geen operator<< beschikbaar, maar die kunnen we makkelijk zelf schrijven.

|  |  |
| --- | --- |
| std::ostream & operator<<( std::ostream & lhs, const xy & pos ){  lhs << "(" << pos.x << "," << pos.y << ")";  return lhs;  }  xy pos1( 1, 2 ), pos2( 12, 33 );  std::cout << pos1 << " " << pos2 << "\n"; | |
| operator<< voor xy | 08-02 |



Deze functie print zijn rechter argument ‘naar’ zijn linker argument, in het formaat (x,y). Daarna returnt hij zijn linker argument, om een keten van << operatoren mogelijk te maken. Let op dat het linker argument, en ook het resultaat, ‘by reference’ worden doorgegeven. Een ostream kan allerlei status informatie en buffers bevatten. Als we ‘by copy’ zouden overdragen zouden we een kopie van die dingen doorkrijgen, daar in schrijven, en daarna zou die kopie (met alle gewijzigde informatie erin) weer verdwijnen. Dat is niet wat we willen, we willen naar de ‘echte’ output schrijven, niet naar een tijdelijke kopie ervan.

Om te bepalen in welk formaat uitvoer wordt geschreven zijn er een aantal uitvoer manipulatoren beschikbaar in de namespace std:: in de header #include<iomanip>. Een manipulator schrijf je naar de uitvoer, en hij beïnvloedt hoe alle volgende items worden geschreven (behalve setw(n), die werkt alleen op het eerstvolgende item!). Als je dus zeker wil zijn hoe je item geschreven wordt, dan moet net ervoor je alle relevante manipulatoren gebruiken.

De basis bepaalt met welk grondtal (basis) de volgende integer waarden worden geschreven. De default is 10. Er is helaas (nog) geen officiele manier om zo binair te printen.

|  |  |
| --- | --- |
| **Manipulator** | **Effect** |
| dec | Schrijf integer getallen als decimaal. Dit is de default. |
| hex | Schrijf integer getallen als hexadecimaal. |
| oct | Schrijf integer getallen als octaal. |
| setbase( n ) | Schrijf getallen met n als basis, n moet 8, 10 of 16 zijn. |

|  |  |
| --- | --- |
| int n = 100;  std::cout << "100 (decimal) : " << n << "\n";  std::cout << "100 (hexdecimal) : " << std::hex << n << "\n";  std::cout << "100 (octal) : " << std::oct << n << "\n";  std::cout << "100 (still octal) : " << n << "\n";  std::cout << "100 (setbase(10)) : " << std::setbase( 10 ) << n << "\n";  std::cout << "\n"; | |
| Effect van de ‘base’ manipulatoren | 08-03 |

Een vlag schakelt een bepaald effect aan of uit. Default staan deze vlaggen uit.

|  |  |  |
| --- | --- | --- |
| **Aanzetten** | **Uitzetten** | **Effect** |
| boolalpha | noboolalpha | Schrijf een bool value als true / false (in plaats van 1 / 0). |
| showbase | noshowbase | Geef voor een integer zijn basis aan, op de zelfde manier als je een literal in de code zou noteren (12, 0x12, 012). |
| showpos | noshowpos | Schrijf een + voor een positief getal. |
| uppercase | nouppercase | Gegenereerde letters in floating point en integer waarden (0x, 1f, etc.) worden als hoofdletters geschreven. |

|  |  |
| --- | --- |
| int n = 100;  bool f = false, t = true;  std::cout << "bool, default : "  << f << " " << t << "\n";  std::cout << "after boolalpha : "  << std::boolalpha  << f << " " << t << "\n";  std::cout << "100, (dec,hex,oct) : "  << std::dec << n << " "  << std::hex << n << " "  << std::oct << n << "\n";  std::cout << "after showbase : "  << std::showbase  << std::dec << n << " "  << std::hex << n << " "  << std::oct << n << "\n";  std::cout << "after uppercase : "  << std::uppercase  << std::dec << n << " "  << std::hex << n << " "  << std::oct << n << "\n";  std::cout << "boolalpha+uppercase : "  << std::uppercase  << f << " " << t << "\n"; | |
| Effect van vlaggen | 08-03 |

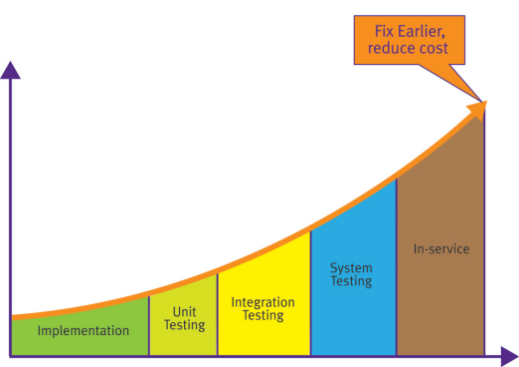
De formaat manipulatoren bepalen hoe breed het veld is waarin geschreven wordt en waar in dat veld de eerstvolgende geschreven waarde komt te staan.

|  |  |
| --- | --- |
| **Manipulator** | **Effect** |
| Left | Als een te schrijven item wordt aangevuld dan gebeurt dit rechts (het item komt dus links in zijn veld). |
| right | Als een te schrijven item wordt aangevuld dan gebeurt dit links (het item komt dus rechts in zijn veld). |
| setfill( c ) | Als een te schrijven item wordt aangevuld dan gebeurt dit met het letterteken c. De default is een spatie. |
| setw( n ) | Een te schrijven item zal ten minste n posities in beslag nemen. De default is 0. |

|  |  |
| --- | --- |
| std::string s = "hello";  std::cout << "default : "  << "[" << s << "]\n";  std::cout << "setw(20) : "  << "[" << std::setw(20) << s << "]\n";  std::cout << "setw(20)+left : "  << std::left << "[" << std::setw(20) << s << "]\n";  std::cout << "setw(20)+setfill('-') : "  << std::setfill('-') << "[" << std::setw(20) << s << "]\n";  std::cout << "setw(20) : "  << "[" << std::setw(20) << s << "]\n"; | |
| Effect van veld manipulatoren | 08-03 |

Er zijn nog meer manipulatoren, bv. om te bepalen met hoeveel decimalen een floating point waarde geschreven wordt. Lees <http://www.cplusplus.com/reference/library/manipulators/> voor meer informatie.

# Unit tests



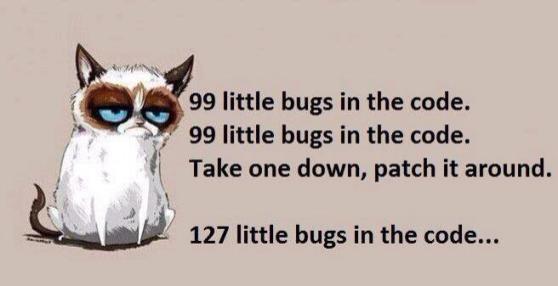
reparatie kosten per fout afgezet tegen de ontwikkelfasen:

een fout repareren is duurder in een latere fase

|  |
| --- |
| behandelde onderwerpen |
| * unit test * assert * Catch2 |

## Waarom testen?

Het is een tragisch ‘fact of life’ dat software fouten (bugs) bevat. Het is natuurlijk een goed voornemen om geen fouten te maken tijdens het schrijven, maar dat zal het aantal fouten nooit tot nul reduceren. Uit onderzoek is gebleken dat het herstellen van een fout (veel) duurder is in latere fasen van de ontwikkeling van een applicatie (een reden is dat het verbeteren van een fout vaak een nieuwe fout introduceert, en als je pech hebt meer dan één), dus het is verstandig fouten zo vroeg mogelijk te vinden.

[](http://blog.typemock.com/wp-content/uploads/2014/03/GrumpyCatBugs1.jpg)

Voor technische software geldt dit misschien nog wel sterker dan voor andere software: als een bug pas tijdens het gebruik van een kerncentrale aan het licht komt is er misschien geen centrale meer om je gewijzigde software te gebruiken. (Als je dichtbij woonde is er misschien ook geen ‘jij’ meer om die wijziging te maken.) Het is dus verstandig software zo vroeg mogelijk te testen. Er is zelfs een ontwikkelmethode (Test Driven Development) die er op is gebazeerd dat je pas code mag schrijven of wijzigen als je eerst een test hebt geschreven die (met de code die je tot nu toe hebt) faalt.

## Unit tests

De kleinste eenheden code die we schrijven worden units genoemd, en het testen daarvan dus unit testing. Een unit test heeft meestal een simpel patroon:

* Zet een paar dingen (objecten) klaar
* Laat de te testen code iets doen met die dingen en/of een nieuw ding maken
* Verifieer dat de oude dingen en/of de nieuwe dingen de verwachte waarden hebben

Neem als voorbeeld de operator+ voor twee xy waarden. Om die te testen moeten we twee xy’s hebben met bekende waarden die we bij elkaar optellen, wat een nieuwe xy oplevert. Na het optellen checken we dat de nieuwe xy de verwachte waarde heeft, en (dat wordt vaak vergeten) dat de oude xy’s nog steeds hun oude waarden hebben. Om te checken of aan een bepaalde voorwaarde is voldaan kan je assert() gebruiken. Een assert met een false argument levert een run-time foutmelding op. Althans, als je compileert voor de debug: als je compileert voor ‘release’ (of de macro NDEBUG definieert) dan worden alle asserts genegeerd.[[34]](#footnote-33)

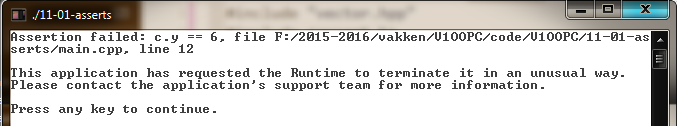
Richt je test zo in dat het waarchijnlijk is dat realistische fouten in de te testen code ook aan het licht komen. Als je alleen test dat (0,0) + (0,0) gelijk is aan (0,0) dan zijn er voor de hand liggende fouten die je niet zal ontdekken. De volgende test is al wat beter, maar wat zou deze test niet ontdekken en wat zou jij er aan veranderen om dat wel te ontdekken?

|  |  |
| --- | --- |
| #include <assert.h>  xy a( 1, 2 );  xy b( 4, 3 );  xy c = a + b;  assert( a.x == 1 ); assert( a.y == 2 );  assert( b.x == 4 ); assert( b.y == 3 );  assert( c.x == 5 ); assert( c.y == 5 ); | |
| recht-toe-echt aan test van xy::operator+ | 09-01 |

De vorige test bevat reeks asserts op de attributen van een xy. Je kan het jezelf (en je lezer) wat makkelijker maken door een operator== te definieren voor xy en die te gebruiken.[[35]](#footnote-34)

|  |  |
| --- | --- |
| bool operator==( const xy &lhs, const xy &rhs ){  return ( lhs.x == rhs.x ) && ( lhs.y == rhs.y );  }  xy a( 1, 2 );  xy b( 4, 3 );  xy c = a + b;  assert( a == xy( 1, 2 ) );  assert( b == xy( 4, 3 ) );  assert( c == xy( 5, 5 ) ); | |
| testen met gebruik van operator== | 09-02 |

Als één van de test faalt krijg je een foutmelding, maar die vermeldt alleen dat een assert faalde op een bepaald regelnummer.

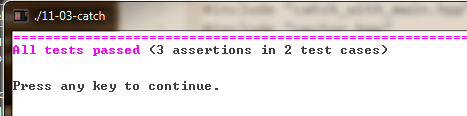


## Catch2

Het zou best handig zijn als de foutemelding ook aangaf wat de beide waarden zijn. Je kan zelf een macro schrijven die dat doet, maar dat werk is natuurlijk al lang een keer gedaan. Een library die zoiets (een vaak nog veel meer) voor je doet heet een ‘unit test framework’. Wij gebruiken een heel simpel unit test framwork: Catch2[[36]](#footnote-35). Een eenvoudige Catch2 applicatie bevat een reeks TEST\_CASEs met daarin aanroepen van REQUIRE. Een main is meestal niet nodig, de main die de header catch.hpp levert als CATCH\_CONFIG\_MAIN is gedefinieerd is doorgaans voldoende. Een TEST\_CASE leest als een soort functie definitie, maar op de plaats van de parameter lijst staat een string literal die de test identificeert.

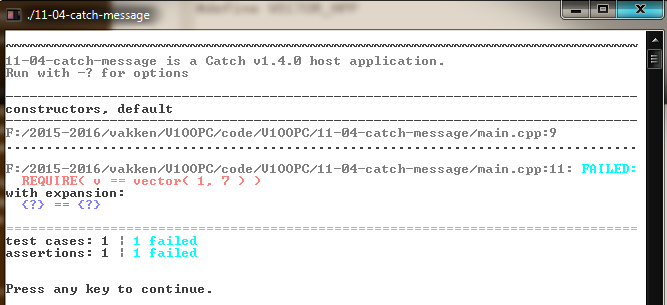
|  |  |
| --- | --- |
| **#include "ostream"**  **#include "xy.hpp"**  **#define CATCH\_CONFIG\_MAIN**  **#include "catch.hpp"**  **bool operator==( xy lhs, xy rhs ){**  **return ( lhs.x == rhs.x ) && ( lhs.y == rhs.y );**  **}**  **std::ostream & operator<<( std::ostream & lhs, xy rhs ){**  **return lhs << "(" << rhs.x << "," << rhs.y << ")";**  **}**  **TEST\_CASE( "constructors, two\_parameters" ){**  **xy v( 3, 4 );**  **REQUIRE( v.x == 3 );**  **REQUIRE( v.y == 4 );**  **}**  **TEST\_CASE( "constructors, default" ){**  **xy v;**  **REQUIRE( v == xy( 0, 0 ) );**  **}** | |
| een eenvoudige Catch applicatie | 09-03 |

De uitvoer van Catch voor een run waarin geen fouten zijn gevonden vermeldt het aantal assertions en het aantal tests dat is uitgevoerd.



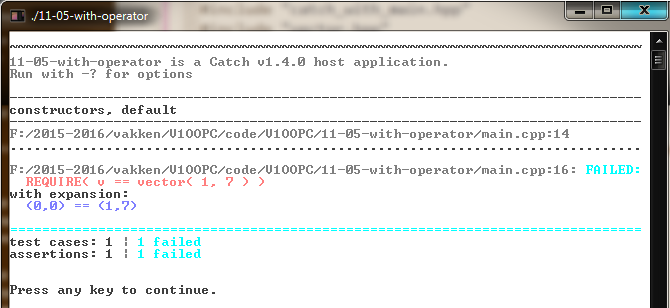
Als een assertion faalt dan wordt dit gemeldt.

|  |  |
| --- | --- |
| **TEST\_CASE( "constructors, default" ){**  **xy v;**  **REQUIRE( v == xy( 1, 7 ) );**  **}** | |
| een test case die een fout zal opleveren | 09-04 |



Er wordt nu gemeld dat er een fout is opgetreden en waar, maar het zou nog handiger zijn als we ook zien welke waarden er aan beide kanten van de operator== staan. Dit kan Catch2 voor je doen, maar dan moeten er voor die waarden wel een operator<< bestaan.

|  |  |
| --- | --- |
| **std::ostream & operator<<( std::ostream & lhs, xy pos ){**  **lhs << "(" << pos.x << "," << pos.y << ")";**  **return lhs;**  **}**  **TEST\_CASE( "constructors, default" ){**  **xy v;**  **REQUIRE( v == xy( 1, 7 ) );**  **}** | |
| falende test case met een operator<< | 09-05 |



Tot nu toe hebben we operator<< alleen gebruikt om te schrijven naar de standaard uitvoer. Om een operator<< te testen is het handig om te weten dat je die via een omweg ook kunt gebruiken om naar een string te schrijven, want dan kan je na afloop de inhoud van die string checken. Dit doe je door naar een (lege) std::stringstream te printen, en daarna de waarde van die stringstream op te vragen met de s.str() methode.

|  |  |
| --- | --- |
| **TEST\_CASE( "operator<<" ){**  **std::stringstream s;**  **xy v( 1, 2 );**  **s << v;**  **REQUIRE( s.str() == "(1,2)" );**  **}** | |
| test de operator<< dmv. std::stringstream | 09-06 |

# Doxygen



nee, niet dat soort Doxygen

|  |
| --- |
| behandelde onderwerpen |
| * Doxygen |

Om een library te kunnen gebruiken moet je weten wat je met de verschillende onderdelen ervan kan doen. Je kan natuurlijk de code gaan lezen om dit uit te vinden, maar het is handiger als de onderdelen van de library die voor extern gebruik bedoeld zijn gedocumenteerd zijn in een overzichtelijk formaat. Programmeurs zijn lui, dus ze hebben tools uitgevonden die met een minimale inspanning een redelijke library documentatie kunnen opleveren. De truuk is hierbij dat de tool zoveel mogelijk informatie uit de code (hpp file) zelf haalt (bv. de headers van de functies, en de onderlinge relaties tussen klassen), en dat de programmeur commentaar toevoegt dat door de tool wordt herkend en meegenomen.[[37]](#footnote-36) Voor C++ is Doxygen veelgebruikte tool.

Doxygen is een command-line tool die je moet runnen in de directory waar je de header files staan waar je documentatie voor wil genereren.[[38]](#footnote-37) In die directory moet een Doxyfile staan die specificeert hoe de documentatie gegenereerd wordt. De Doxyfile is een lange text file met heel veel uitleg. In de meeste gevallen zijn de defaults die er in staan prima. Als je doxygen runt (met de Doxyfile die bij de voorbeelden zit) wordt een subdirectory html aangemaak met daarin de documentatie. Het startpunt van de documentatie is de index.html file in die subdirectory.

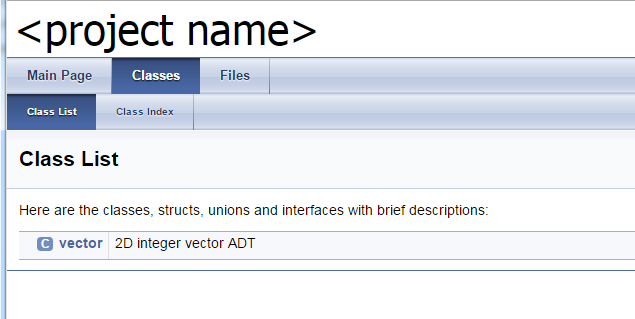
In de header files waarvoor documentatie gegenereerd moet worden moet commentaar staan dat begint met ///. Bovenin de file moet een regel ///@file staan om aan te geven dat deze file Doxygen input is. Bij alle te documenteren onderdelen (namespaces, klassen, niet-prive attributen en functies, losse functies, losse variabelen[[39]](#footnote-38)) zet je

* /// \brief, gevold door een regel die in telegramstijl (geen grammaticaal volledige zin) aangeeft wat het onderdeel doet. Deze regel komt terecht in een overzicht van alle onderdelen, dus een lezer moet er snel uit kunnen opmaken of hij het onderdeel wil gaan gebruiken.
* /// \details, gevolgd door een stuk text (met volledige zinnen) dat aangeeft hoe het onderdeel gebruikt moet worden. Dit moet genoeg informatie bevatten voor een gebruiker om het onderdeel te kunnen gebruiken. Hoeveel dat is hangt af van de complexiteit van (het gebruik van) het onderdeel. In heel eenvoudige gevallen is de naam en de telegramstijl regel al voldoende en kan je dit stuk text weglaten.

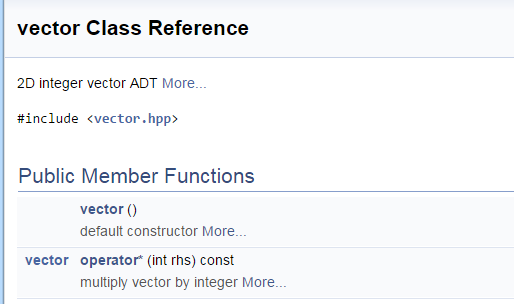
|  |  |
| --- | --- |
| **/// @file**  **/// \brief**  **/// 2D integer vector ADT**  **/// \details**  **/// This is 2D vector ADT that stores its two components as (signed) integers.**  **/// The x and y components are public attributes.**  **/// The appropriate constructors and operators are provided.**  **class xy {**  **public:**  **/// \brief**  **/// default constructor**  **/// \details**  **/// This constructor does not initialize the x and y attributes.**  **xy();**  **/// \brief**  **/// multiply xy by integer**  **/// \details**  **/// This operator multiplies an xy by an integer, yielding an xy.**  **/// The resulting xy points in the same direction as the original,**  **/// but its length is rhs times longer than the original.**  **/// In other words, the x and y attributes of the result are**  **/// the x and y of the original, multiplied by the rhs.**  **xy operator\*( int rhs ) const;**  **. . .**  **};** | |
| een deel van een met Doxygen gedocumenteerde xy declaratie | 10-01 |

Voor functies met veel parameters kan je ook een beschrijving per parameter geven.

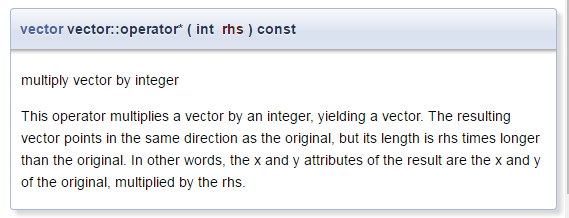
Doxgen geneert hieruit een overzicht van alle (in dit geval dus maar één) klassen. De <project name> staat in de Doxyfile, die kan je daar dus aanpassen.



Voor iedere klasse genereert Doxygen een pagina met bovenaan de beschrijvng van de klasse, en een overzicht van de dingen (attributen, methoden) in de klasse.



Die dingen linken naar de langere beschrijving die verderop in dezelfde pagina staan.



Dit is het meest eenvoudige gebruik van Doxygen. Er zijn veel meer mogelijkheden, bv.

* formatteren van de gegenereerde text
* genereren van documentatie in document (printbare) vorm
* documentatie per parameter
* invoegen van plaatjes
* invoegen van fragmenten uit andere files (handig voor code voorbeelden)
* documentatie die niet bij een bepaald onderdeel hoort, bv. een algemene inleiding op de hoofdpagina.

# For, auto



Laat de auto het werk voor je doen

|  |
| --- |
| behandelde onderwerpen |
| * for( : ) * auto: pre-C++0x11 betekenis * auto: C++0x11 betekenis * use & or const & in for( : ) |

## For

Je kan je een array doorlopen via indexering. Je geeft dan met een (integer) index aan welk array element je wilt benaderen. Dat gebeurt doorgaans in een for loop, waarin je expliciet aangeeft wat de bovengrens is. Dit is een uitgelezen ‘kans’ om een zogenaamde ‘off by one error’ te veroorzaken: je vergeet 1 array element te benaderen, of je benadert juist 1 element te veel (dat helemaal niet bestaat). Als je erg creatief bent doe je dat zelfs beide.

|  |  |
| --- | --- |
| int main(){    const unsigned int array\_size = 100;  std::array< int, array\_size > array;  for( unsigned int i = 1; i <= array\_size; ++i ){  array[ i ] = 0;  }  return 0;  } | |
| Code voor het op 0 zetten van alle entries in een array; wat is hier fout?[[40]](#endnote-0) | 11-01 |

|  |  |
| --- | --- |
| const unsigned int array\_size = 100;  std::array< int, array\_size > array;    for( unsigned int i = 0; i < array\_size; ++i ){  if( array[ i ] > array[ i + 1 ] ){  std::cout << "not sorted\n";  fouten++;  }  } | |
| Code voor het controleren of een array gesorteerd is; wat is hier fout?[[41]](#endnote-1) | 11-02 |

Er is (sinds C++ 0x11) een alternatieve vorm van de for loop, speciaal voor het benaderen van alle elementen in een array, de zg. range-based-for. Hierbij hoef je zelf niet aan te geven hoeveel elementen er in het array zitten, dat zoekt de compiler zelf wel uit (aan de hand van de declaratie van het array). In het volgende voorbeeld krijgt de loop variabele achtereenvolgens alle waarden uit het array.

|  |  |
| --- | --- |
| const int array\_size = 100;  int array[ array\_size ];  int sum = 0;  for( int element : array ){  sum += element;  } | |
| C++ code voor het optellen van de elementen van een array | 11-03 |

Je moet wel opletten: je loop variabele krijgt achtereenvolgens alle **waarden** uit het array. De volgende code is dus niet de goede manier om de elementen van een array op 0 zet: je loop variabele is telkens een kopie van je array element, en je ze die kopie op 0.

|  |  |
| --- | --- |
| const int array\_size = 100;  std::array< int, array\_size > array;  for( int x : array ){  x = 0;  } | |
| een niet geslaagde poging om een array op 0 te zetten[[42]](#endnote-2) | 11-04 |

Dit probleem kan je oplossen door als loop variabele geen integer maar een integer reference te gebruiken. Dan wordt je loop variabele achtereenvolgens geinitializeerd als reference naar ieder array element, en als je aan een reference 0 toekent dan ken je toe aan de achterliggende ’echte’ variabele, in dit geval dus de array elementen. Het verschil in code is maar een enkel & teken, maar het verschil in effect is enorm. Dit lijkt op de situatie bij het doorgeven van een parameter: als je niets doet krijg je een copie.

|  |  |
| --- | --- |
| const int array\_size = 100;  std::array< int, array\_size > array;  for( int & x : array ){  x = 0;  } | |
| een array op 0 te zetten via een reference loop variabele | 11-05 |

## Auto

In C++ vóór de C++0x11 standaard werd auto gebruikt om aan te compiler aan te geven dat een lokale variable het beste op de stack gealloceerd kon worden. Andere mogelijkheden waren static (globaal) en register (in een CPU register). In de onderstaande code heeft de programmeur aangegeven dat de pointer p het beste in een CPU register kan worden gealloceerd, en sum op de stack.

|  |
| --- |
| int add( int \*arg\_p; int n ){  register int \* p = arg\_p;  auto int sum = 0;  while( n > 0 ){  sum += \*p;  ++p;  --n;  }  } |
| pré-C++0x11 gebruik van register en auto |

Compilers zijn al lang veel beter dan menselijke programmeurs in het bepalen of een variabele op de stack of in een register moet worden gealloceerd, en die afweging hangt sterk af van het de specifieke CPU waar code voor gegenereerd wordt, met name het aantal registers dat de CPU heeft. De default (geen register, static of auto) is dat een lokale variabele op de stack wordt gealloceerd, dus auto werd eigenlijk al nooit meer gebruikt.

In C++0x11 heeft het auto keyword een nieuwe betekenis gekregen: als je een variabele creëert met een initiele waarde (je roept dus een constructor aan), dan kan je als type van die variabele auto geven. De compiler vervangt jouw auto dan door het type van de expressie. Als die expressie const en/of een referentie is dan wordt dit **niet** meegenomen. Je kan wel zelf const en/of & toevoegen.

|  |
| --- |
| auto pi = 3.14; // pi is een double  const auto p1( &pi ); // p1 is een const double  auto & p2( pi ); // p2 is een double & |

Een populair gebruik van auto is bij het declararen van de loop variabele in een for( : ) statement. Let daarbij wel op: een kale auto betekent in dit geval dat je een variabele krijgt, die iedere keer dat de lus van de loop begint, wordt geconstrueerd aan de hand van de waarde van het array element, maw. een **kopie**. Je kan aan die kopie een waarde toekennen, maar dat heeft geen effect op het array element. Als je dat wel wil, dan moet je de loop variabele als reference deklareren.

Het is daarom aan te raden de loop variabele **altijd als reference te declareren**, en als je niet de bedoeling had de array elementen te wijzigen, dan ook als const. Je bedoeling is dan duidelijk voor de lezer van je code, en als je dan per ongeluk toch assignt aan de loop variabele krijg je een foutmelding van de compiler.

|  |  |
| --- | --- |
| const int array\_size = 100;  std::array< int, array\_size > a;  . . .  for( auto & x : a ){  x = 5; // a is changed  }  . . .  for( const auto & x : a ){  x = 5; // compilation error  } | |
| gebruik altijd auto & voor de for( : ) loop parameter, eventueel met const | 11-07 |

Het gebruik van auto (in plaats van het expliciet aangeven van het type van de variabele) wordt (sterk) aangeraden als dit *de leesbaarheid verhoogt* of *duplicatie voorkomt*. Helaas is niet iedereen het er over eens waanneer dit het geval is, maar de tendens is om overal waar dit mogelijk is auto te gebruiken.

In het volgende voorbeeld moet een functie een reeks array elementen optelen. In de meeste gevallen zal je voor de som (variabele waarin de waarden worden opgeteld) het zelfde type (short, int, long, float, double, etc.) willen gebruiken als de array elementen. Als je voor som het type expliciet spexcifieert, dan staat dit type dus op 2 plaatsen. Dit brengt een risico met zich mee bij wijzigen, en de lezer moet zich afvragen of het toevallig of zo bedoeld is dat de beide typen het zelfde zijn.

Als je het type van de som als auto opgeeft kan dit niet mis gaan, en het is voor de lezer duidelijk dat het de bedoeling is dat sum hetzelfde type moet zijn als de elementen van het array.

|  |  |
| --- | --- |
| void add1( long x[], . . . ){  long sum = x[ 0 ];  . . .  }  void add2( long x[], . . . ){  auto sum = x[ 0 ];  x = 0;  . . .  } | |
| Gebruik van auto om duplicatie te voorkomen |  |

# Overerving

|  |
| --- |
| https://upload.wikimedia.org/wikipedia/commons/4/49/Sexlinked_inheritance_white.jpg |

*Overerving* (bij fruitvliegjes)

|  |
| --- |
| behandelde onderwerpen |
| * (public) inheritance (overerving) * protected * virtual, abstract, override * pointer |

## Een bewegende bal

Stel dat we een bal op het scherm willen afbeelden die (eenparig) beweegt. Een bal (afgebeeld als een cirkel) zit al in de code die gebruikt is bij de opgaven van vorige week, maar die heeft geen attribuut dat een snelheid opslaat, en zal al helemaal niet uit zichzelf bewegen. We kunnen natuurlijk code schrijven die buiten de bal om onthoudt welke snelheid de bal heeft, en aan de hand daarvan de posititie met vaste intervallen aanpast. Maar volgens de Object Oriented principes is het bijwerken van de positie een verantwoordelijkheid van het bal object, dus hoort de snelheid, en de code die de positie aanpast, **in** de bal thuis, niet er buiten.

We kunnen een bal klasse schrijven met alle gegevens en functies van een circkel er in, plus de gegevens en functies voor bewegen, maar dan hebben we die circkel aspecten (middelpunt, diameter, draw() functie) op twee plekken in ons programma staan: in cirkel en in bal. Dat is een slecht teken voor de kwaliteit van je code, en het is dubbel werk voor jou.[[43]](#footnote-39)

|  |
| --- |
| D:\2017-2018\vakken\V1OOPC\images\inheritance\circle-and-ball.png |
| UML diagram voor aparte cirkel en bal klassen |

|  |  |
| --- | --- |
| class circle {  private:  xy midpoint;  int radius;  public:  circle( window & w, xy midpoint, int radius );  void draw();  };  class ball {  private:  xy midpoint;  int radius;  location speed;  public:  ball( window & w, xy midpoint, int radius, xy speed );  void draw();  void update();  };  void ball::update(){  midpoint += speed;  } | |
| C++ code voor aparte cirkel en bal klassen, met een update functie (dit kan beter met overerving) | 12-01 |

We kunnen nu een bal aanmaken, tekenen, zichzelf laten updaten, en dan weer tekenen.

|  |  |  |
| --- | --- | --- |
| window w( 128, 64, 2 );  ball b( w, xy( 30, 30 ), 20, xy( 10, 20 ) );  b.draw();  b.update();  b.draw(); |  | |
| Een ‘bewegende’ bal. | | 12-01 |

## Overerving

Wat we hier kunnen gebruiken is het OO principe van **overerving**: als een subklasse erft van een superklasse, dan heeft die subklasse alle eigenschappen (attributen en functies) van de superklasse, en waarschijnlijk nog wat meer (anders heeft het weinig zin om een subklasse te maken). In ons geval maken we de circkel de superklasse, en daarvan leiden we de subklasse bal af. In UML geven we dit aan met een open pijl[[44]](#footnote-40), in C++ met “: public circle”. Met deze opzet erft bal alle dingen die in circkel zitten, dus de duplicaties in bal kunnen weg.[[45]](#footnote-41)

|  |
| --- |
| D:\2017-2018\vakken\V1OOPC\images\inheritance\ball-from-circle.png |
| UML diagram voor aparte cirkel en bal klassen |

|  |  |
| --- | --- |
| class circle {  private:  window & w;  location midpoint;  int radius;  public:  circle( window & w, const xy & midpoint, int radius );  draw();  };  class ball : public circle {  private:  xy speed;  public:  ball(window & w, const xy & midpoint, int radius, const xy & speed );  void update();  }; | |
| C++ code voor een bal afgeleid van cirkel | 12-02 |

In de constructor van bal moeten we de constructor van circkel aanroepen, en de juiste parameters doorgeven.

|  |  |
| --- | --- |
| ball::ball( window & w, const xy & midpoint, int radius, const xy & speed ):  circle( w, midpoint, radius ),  speed( speed )  {} | |
| de constructor van bal moet de constructor van zijn superklasse aanroepen | 12-02 |

Er is nu wel een probleem met de update functie: dat is een functie van bal, maar hij probeert de positie, wat een variabele van circkel is, te wijzigen. Die variabele is prive, dus dat zal de compiler niet toestaan.

|  |  |
| --- | --- |
| void ball::update(){  midpoint = midpoint b+ speed; // compiler error  } | |
| prive variabele van een superklasse zijn niet toegankelijk in een subklasse klasse | 12-02 |

We kunnen die variabele natuurlijk publiek maken, maar dat is misschien wat te veel van het goede. Een tussenoplossing is om de variabele protected te maken: dan is hij wel bereikbaar voor de klasse zelf en voor afgeleide klassen, maar niet voor andere code.

|  |  |
| --- | --- |
| class circle {  protected:  xy midpoint;  . . .  };  class ball : public circle {  . . .  void update(){  midpoint = midpoint + speed; // no compiler error  }  }; | |
| protected: ook toegankelijk in subklassen | 12-03 |

Nu is alles op zijn plaats voor een bewegende bal. De applicatie creeert de window en de bal, en daarna komt een oneindige lus die steeds de window leeg maakt, de bal tekent, event wacht, en de bal verplaatst naar zijn nieuwe positie. Merk op hoe weinig details er in de main staan: de window en de bal moeten gecreeerd worden met al hun details (plaats, omvang, snelheid, etc), maar daarna zijn ze zelfstandig: een bal weet zelf hoe hij zich afbeeldt (en zelfs waarop), en hoe hij beweegt. Zelfs het detail dat de bal beweegt is niet zichtbaar: de bal kan op de update reageren zoals hij wil, hij zou ook bv. van kleur of omvang kunnen veranderen.

|  |  |  |
| --- | --- | --- |
| int main(){  window w( 128, 64, 2 );  bal b( w, xy( 10, 10 ), 8, xy( 2, 2 ));  for(;;){  w.clear();  b.draw();  wait\_ms( 100 );  b.update();  }  } |  | |
| een bewegende bal | | 12-04 |

C++ is in principe een statisch (compile-time) getypeerde taal: bij een functie parameter moet je aangeven van welk type die parameter is, en je kan die functie alleen aanroepen met het een parameter van dat type, anders compileert je code niet. Het voordeel is dat er tijdens het runnen van je programma geen type checks meer hoeven plaats te vinden, wat de snelheid ten goede komt. Maar soms is het handig om een functie te kunnen schrijven die kan werken met verschillende typen parameters. Dit kan in C++ als die verschillende typen een gemeenschappelijk supertype hebben: je kan dan het supertype als parameter type gebruiken. Alle subtypen van dat supertype zijn dan toegestaan als parameter.

|  |  |  |
| --- | --- | --- |
| class animal { . . . };  void sell( animal & a );  class duck : public animal { . . . };  void fly( duck & d );  duck d;  fly( d ); // OK: duck required, duck provided  sell( d ); // OK: animal required, duck provided  animal a;  sell( a ); // OK: animal required, animal provided  fly( a ); // ERROR: duck required, animal provided | D:\2017-2018\vakken\V1OOPC\images\inheritance\duck-from-animal.png | |
| een subklasse object is acceptabel als een superklasse object wordt gevraagd | | 12-05 |

De regel dat een subklasse object overal gebruikt kan worden waar een superklasse vereist is geldt ook voor pointers en references. Een superklasse pointer of superklasse reference kan ook naar een subklasse object wijzen, maar andersom kan niet.[[46]](#footnote-42)

|  |  |
| --- | --- |
| class animal { . . . };  class duck : public animal { . . . };  duck d;  duck \* dp;  animal a;  animal \* ap;  ap = & a;// OK: animal pointer required, animal pointer provided  ap = & d; // OK: animal pointer required, duckpointer provided  dp = & d; // OK: duck pointer required, duckpointer provided  dp = & a; // ERROR: duck pointer required, animal pointer provided | |
| een superklasse pointer kan ook naar een subklasse object wijzen | 12-05 |

|  |  |
| --- | --- |
| class animal { . . . };  class duck : public animal { . . . };  duck d;  animal a;  animal & ar1 = a;// OK: animal reference refers to an animal  animal & ar2 = d; // OK: animal reference refers to a duck  duck & dr1 = d; // OK: duck reference refers to a duck  duck & dr2 = a; // ERROR: rerence can’t refer to a plain animal | |
| een superklasse reference kan ook aan een subklasse object refereren | 12-05 |

## Virtual en abstract

Als je in C++ een object van een afgeleide klasse gebruikt via een pointer of reference naar zijn superklasse, dan kan je alleen bij de eigenschappen van de superklasse. Het subklasse deel is er nog wel, maar het is voor jouw code onzichtbaar. Dit gaat zo ver dat als je in de superklasse en de subklasse een functie hebt met dezelfde naam, je dan toch de superklasse functie gebruikt.

|  |  |  |
| --- | --- | --- |
| class animal {  public:  void print(){  std::cout <<"I am an animal.\n";  }  };  class duck : public animal {  public:  void print(){  std::cout <<"I am a duck.\n";  }  };  animal a;  duck d;  a.print(); // I am an animal  d.print(); // I am a duck  animal & ar = d;  ar.print(); // I am an animal <======== !!  void f( animal & x ){  x.print();  }  f( a ); // I am an animal  f( d ); // I am an animal <======== !! | D:\2017-2018\vakken\V1OOPC\images\inheritance\functions.png | |
| welke functie wordt aangeroepen? | | 12-06 |

Dat is vaak niet wat je zou willen: het is bv. handig als het een verzameling objecten (bv. dingen die we op het scherm afbeelden) allemaal eenzelfde functie hebben die we kunnen aanroepen, maar dat wat ze dan in die functie doen verschillend is. Naast de bewegende bal zou ik ook muren op het scherm willen afbeelden, die niet bewegen, maar op een andere manier reageren op het verstrijken van tijd. Vervolgens wil ik deze scherm objecten onthouden in een array, zodat ik ze in een loop kan bijwerken en afbeelden.

Om dit te kunnen doen moeten gebruiken we weer inheritance, maar wat we doen is het omgekeerde doen van specailizeren: we moeten generalizeren. De dingen die de verschillende scherm objecten gemeenschappelijk hebben brengen we onder in een gemeenschappelijk supper klasse (‘common base class’):

* een beginpunt (locatie)
* het feit dat ze afgebeeld kunnen worden
* het feit dat ze bijgewerkt kunnen worden.

Die klasse kunnen we noemen naar wat zijn objecten zijn (scherm objecten?), of zoals in dit geval, naar wat je er mee kan (je kan ze tekenen). Anders dan bij de bal die erft van een cirkel, is de superklasse nu iets volkomen theoretisch: een drawable zelf is niets, het is **abstract**: het is alleen het gemeenschappelijke deel van zijn **concrete** subklassen. Je kunt van een drawble niet aangeven wat de body van draw() zou moeten zijn: dit kan alleen voor de echte (concrete) objecten. Verder moet, als we bv. de draw() of update() methode van een schermobject aanroepen, de methode in het echte object worden aangeroepen. Om dit te bereiken moeten we twee C++ features gebruiken:

* De draw functie is **virtual**. Dit betekent dat als we deze functie aanroepen, we de functie van het ‘echte’ object aanroepen. Dit is iets langzamer en kost iets meer geheugen dan het aanroepen van een gewone functie, vandaar dat het in C++ niet de default is.[[47]](#footnote-43)
* De draw() functie is behalve virtual ook nog **abstract**. Dit betekent dat er in drawable helemaal geen (drawable-specifieke) draw() functie is. Dat is ook wel logisch, hoe zou je een object willen tekenen als je niet weet wat voor object het is?

De update() functie is wel virtual, want als je een object (waar je verder niets van weet, alleen dat het een object is) wil bijwerken dan moet je de update functie van dat eigenlijke object aanroepen. Maar bijwerken is voor veel objecten (bv. de gewone lijn, cirkel en rechthoek) een lege operatie. Daarom is update() niet abstract: er is een (lege) implementatie gegeven. De ‘gewone’ subobjecten hoeven die dus niet te her-definieren. De bal doet dat wel, want die moet wel bewegen.

In UML wordt abstract aangegeven doordat de naam in italic staat. Virtual zie je niet in UML, omdat het een implementatie detail is (dat in de meeste talen niet speelt).

|  |
| --- |
| D:\2017-2018\vakken\V1OOPC\images\inheritance\drawables.png |
| schermobjecten met een gemeenschappelijk superklasse |

In C++ wordt virtual aangegeven met het virtual keyword voor de functie, en abstract met = 0 op de plaats van de functie body. Als je in een afgeleide klasse een functie implementeerd die in de superklasse virtual is, dan geef je dan aan met ‘override’ achter de parameter lijst.[[48]](#footnote-44)

|  |  |
| --- | --- |
| class drawable {  protected:  window & w;  xy location;  public:  drawable( window & w, const xy & location );  virtual void draw() = 0;  virtual void update(){}  };  class circle : public drawable {  protected:  int radius;  public:  circle( window & w, const xy & midpoint, int radius );  void draw() override;  };  class ball : public circle {  private:  xy speed;  public:  ball( window & w, const xy & midpoint, int radius, const xy & speed );  void update() override;  }; | |
| C++: drawable, circle, ball | 12-07 |

Als een klasse een of meer abstracte functies heeft (in het besproken geval geldt dit dus alleen voor drawable) dan is die klasse als geheel abstract. Van een abstracte klasse kan je geen objecten (variabelen) aanmaken. Je kan wel pointers en referenties maken die wijzen naar zo’n abstract type, maar die kunnen alleen wijzen naar bestaande objecten van een afgeleid, niet-asbtract type.[[49]](#footnote-45)

|  |  |
| --- | --- |
| class abstract {  public:  virtual void draw() = 0;  };  abstract obj; // ERROR: object is abstract  obj.draw(); // and what would this call anyway?  class concrete : public abstract {  public:  void draw() override {}  };  concrete brick;  brick.draw(); // call the concrete draw()  abstract & p = brick;  p->draw(); // again, call the concrete draw | |
| een variabele van een abstracte klasse? | 12-08 |

## Pointers

Een pointer is in C++ de voorloper van een reference. Een pointer variabele declareer je met een \* voor de naam (vergelijkbaar met de & voor de naam bij een reference). Een pointer is lastiger te gebruiken dan een reference:

* Een pointer kan je niet niet zomaar kan initializeren met een variabele (zoals bij een reference), je moet expliciet aangeven dat je het adres van die variabele wil hebben, door een & voor de variabele te zetten.
* Een pointer kan je niet zomaar gebruiken alsof het de variabele is waarb hij naar wijst, je moet er een \* voor zetten.

|  |  |
| --- | --- |
| int n; // integer, waarden 0, 1, etc.  int &r = n; // reference naar n  int \*p = &n; // pointer naar n  n = 5;  std::cout << n << ” ” << r << ” ” << \*p << ”\n”; // three ways to print n | |
| Pointers en references |  |

Een belangrijk verschil met references is dat pointers gedurende hun bestaan naar verschillende oobjecten kunnen wijzen (die moeten wel van het zelfde type zijn), en dat je arrays en vectors van pointers kan maken (een arrays en vectors van references kunnen niet in C++). Je doet dit door een nieuw adres toe te kennen aan de pointer zelf (dus aan p, niet aan \*p).

|  |  |
| --- | --- |
| int a = 42;  int b = 8;  int \*p = &a;  std::cout << a << ” ” << b << ” ” << \*p << ”\n”;  p = &b;  std::cout << a << ” ” << b << ” ” << \*p << ”\n”; | |
| Assigning a pointer |  |

Net als bij references kan een superklasse pointer wijzen naar een object van een afgeleide klasse.

## Stuiteren

We kunnen nu het voorbeeld van de bewegende bal uitbreiden met vier muren. Daarmee komt het aantal scherm-objecten al op vijf, en in een realistische applicatie zullen het er nog veel meer worden. Het is dan onhandig (en foutgevoelig) als we voor ieder object dat we toevoegen de draw en update loops moeten aanpassen. Een oplossing hiervoor is om de scherm-objecten (of eigenlijk referenties naar de schermobjecten) op te slaan in een array. Dan hoeven we in de loop niet meer te weten wat voor objecten er zijn, die worden buiten de loop klaargezet. Voor een klein aantal objecten is dat misschien wat meer werk dan ze expliciet aanroepen, maar het levert nu al een veel overzichtelijker programma op.

|  |
| --- |
| int main(){  window w( 128, 64, 2 );  bal b( w, xy( 10, 10 ), 8, xy( 2, 2 ));  line top( . . . );  line right( . . . );  line bottom( . . . );  line left( . . . );  std::array< drawable \*, 5 > objects = { &b, &top, &left, &right, &bottom };  for(;;){  w.clear();  for( auto p & : objects ){  p->draw();  }  w.flush();  wait\_ms( 100 );  for( auto p & : objects ){  p->update();  }  }  } |

Als je een pointer hebt naar een object en je wilt een functie van dat object aanroepen, dan is de standaard syntax (\*p).f() De taal heeft hiervoor een alternatief (zie het als een afkorting): p->f().

Het is wel jammer dat de bal gewoon door een muur heen gaat. Om dat te veranderen kunnen we invoeren dat objecten op elkaar kunnen reageren. Dat wordt dan een nieuwe eigenschap van de superklasse: op een ander object reageren. Die functie heeft 1 argument: het scherm-object waar mee ge-interact wordt. De interactie die we willen realiseren is dat een bal afketst op een muur. Daarvoor moeten we bepalen of een bal een muur raakt. Een niet geheel perfecte (maar wel makkelijke) manier om dit te doen is ieder object een ‘bounding box’ te geven: de (kleinste) rechthoek waar het object in past. Het is redelijk simpel om van twee bounding boxes te bepalen of ze elkaar overlappen.Dit brengen we onder in de overlaps() functie.

|  |
| --- |
| D:\2017-2018\vakken\V1OOPC\images\inheritance\interaction.png |
| UML voor een drawable met interactie |

Om dit te gebruiken moeten dus aan de superklasse een variabele en twee functies toevoegen, en de constructor aanpassen. De interact() functie moet natuurlijk virtual zijn, want de functie van het echte object moet worden aangeroepen. De default is (net als bij update) een lege body: als je in een object niets zegt over zijn interactie, dan interact hij niet.

|  |  |
| --- | --- |
| class drawable {  protected:  window w;  xy location;  xy size;  public:  drawable( window & w, const xy & location, const xy & size );  virtual void draw() = 0;  virtual void update(){}  bool overlaps( const drawable & other );  virtual void interact( drawable & other ){}  }; | |
| nieuwe eigenschappen voor de drawable klasse. | 12-10 |

Hoe gaat een bal nu interacteren met een ander object? Eigenlijk heel simpel. De interact functie van bal kijkt of er een overlap is met het andere object, en zo ja dan keert hij om door de x en y componenten van zijn snelheid te met -1 te vermenigvuldigen. (Dat is mechanisch niet correct, maar dat laat ik graag aan jullie over om te verbeteren.)

|  |  |
| --- | --- |
| class ball : public circle {  private:  xy speed;  public:  ball( window & w, const xy & midpoint, int radius, const xy & speed );  void update() override;  void interact( drawable & other ) override;  };  void ball::interact( drawable & other ){  if( this != & other){  if( overlaps( other )){  speed.x \*= -1;  speed.y \*= -1;  }  }  } | |
| de bal klasse met de interactie functie | 12-10 |

Wat we nu nog moeten doen is in de main niet alleen het scherm overnieuw tekenen en de objecten zichzelf laten bijwerken, maar ook een geneste for-lus die ieder object met ieder ander object laat interacten.

|  |  |
| --- | --- |
| int main(){  . . .  drawable \* objects[] = { . . . };  for(;;){  w.clear();  for( auto & p : objects ){  p->draw();  }  wait\_ms( 200 );  for( auto & p : objects ){  p->update();  }  for( auto & p : objects ){  for( auto & other : objects ){  p->interact( \*other );  }  }  }  } | |
| nogmaals een bewegende bal, maar nu met interactie | 12-10 |

Hiermee hebben een applicatie die een bewegende bal toont. In de opgaven gaan we hier op voortbouwen.

# Dingen en waarden



|  |
| --- |
| behandelde onderwerpen |
| * ding-achtige objecten, waarde-achtige objecten * plaats van ieder type in een klasse hiërarchie |

Je hebt nu twee soorten zaken gezien die je in C++ als klasse kan implementeren: dingen en waarden. Een ding is het klassieke OO object: het heeft een identiteit, het is iets. Denk aan een persoon, een auto, een bankrekening, een beeldscherm, etc. De eigenschappen van een ding kunnen in de loop van zijn bestaan veranderen, maar het blijft hetzelfde ding. Als je twee dingen vergelijkt op gelijkheid, dan bedoel je (meestal) ‘zijn deze twee variabelen hetzelfde ding’. Dingen zitten vaak in een klasse hiërarchie: een student is afgeleid van een persoon, en een student-assistent is weer afgeleid van student.

|  |
| --- |
|  |
|  |

Een waarde is bijvoorbeeld een getal, een kleur, een naam, het saldo op een rekening, een tijdsduur, een GPS coördinaat, etc. Een waarde is geen ding. Een variabele is niet een waarde, het bevat een waarde, en die waarde kan in de loop der tijd veranderen (anders was het geen variabele maar een constante). Als je twee waarden test op gelijkheid dan bedoel je of ze beiden dezelfde waarde bevatten, niet of ze dezelfde variabele zijn. Een waarde maakt doorgaans geen deel uit van een klasse hiërarchie, maar het is toch nuttig om een waarde als klasse te implementeren omdat je dan details van de implementatie kan verbergen, en methoden en operaties kan leveren. Zo’n klasse noemt men een Abstract Data Type (ADT).

De typische eigenschappen van een object en een ADT staan in de volgende tabel naast elkaar.

|  |  |
| --- | --- |
| **Object** | **ADT** |
| Heeft een identiteit. | Bevat een waarde. |
| Gelijkheid betekent hetzelfde ding zijn. | Gelijkheid betekent dezelfde waarde bevatten. |
| Maakt deel uit van een klasse hiërarchie. | Staat op zich zelf. |
| Geen equality of andere vergelijkingen, geen operatoren. | Heeft equality, mogelijk andere vergelijkingen, vaak operatoren. |
| Een basis klasse heeft een virtuele destructor, en vaak ook (veel) andere virtuele methoden. | Niet bedoeld om van te erven. |
| Is zelden een constante. | Constanten komen vaak voor. |
| Is vaak een fysiek ‘object’. Typisch voorbeeld: persoon. | Is vaak iets meetbaars, en heeft vaak een eenheid. Typisch voorbeeld: kleur. |
| Bevat vaak referenties naar andere objecten. | Is doorgaans ‘self contained’: bevat geen verwijzingen naar andere objecten. |

# Micro-controllers



|  |
| --- |
| behandelde onderwerpen: |
| * Micro-controller * Arduino Due * Cross-development met CodeLite en bmptk |

## Wat is een micro-controller

Een micro-controller is een enkele chip die alle essentiele onderdelen van een computer bevat:

* CPU (processor),
* code geheugen (FLASH ROM),
* data geheugen (RAM), en
* I/O.

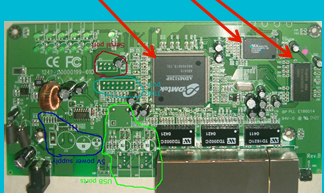
Een micro-controller bevat veel minder geheugen en heeft een langzamere CPU dan een PC (laptop of dektop), of zelfs een enkel-kaarts computer als de RaspberryPi. Maar een micro-controller is goedkoper, en (belangrijk voor batterijgevoede apparaten) kan een veel lager energieverbruik hebben.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | CPU | RAM | ROM | Prijs | Engergie-verbruik |
| PC anno 2016 | 32 of 64 bit Intel, 2-3 GHz (multicore) | 4-8 Gb | Harde schijf, bv 1 Tb | E 300 | 100 W |
| Raspberry Pi 2B | 32-bit Cortex A7, 900 MHz (4 core) | 1 Gb | flash kaart, bv 32 Gb | E 30 | 5 W |
| ATSAM3X8E  (Arduino Due) | 32-bit Cortex M3, 84 MHz | 96 kb | 512 kb | E 15 | 100 mW |
| AtMega328 (Arduino Uno) | 8-bit, 16 MHz | 2 kb | 16 kb | E 3 | 10 mW |
| PIC10F200 | 8-bit, 4 MHz (1 MIPS) | 16 b | 256 instructies | E 0.50 | 1 mW |

Micro-controllers worden gebruikt in allerlei apparaten, van smartcards en keyfobs (electrische sleutels) tot industriele besturingen.



Een micro-controller heeft niet genoeg geheugen (en vaak ook niet voldoende processor-kracht) om een echt OS (Operating System) te draaien. Als een OS wel nodig is worden losse CPU, ROM en RAM chips gebruikt. Zo’n systeem is groter, duurder, en gebruikt beduidend meer vermogen dan een micro-controller. Zulke systemen, vaak met een Linux als OS, worden gebruikt in internet modems, routers, game consoles, intelligente TVs, moderne mobieltjes (telefoons), en tablets, en (maar dan met Windows of MacOS) in desktop en laptop PCs.



Micro-controllers worden vaak ‘bare-metal’ (direct op het metaal) geprogrammeerd. Dit wil zeggen dat er behalve de applicatie (inclusief allerlei libraries) niets anders op de micro-controller draait, dus geen los OS. Voor de grotere micro-controllers zijn Real-Time OS’es beschikbaar, die wel los van een applicatie programma draaien. Zo’n RTOS is heel iets anders dan Windows of Linux: het is geoptimalizeerd om weinig resources (geheugen en CPU) te gebruiken, en heeft meestal geen desktop interface.

Een micro-controller is over het algemeen niet geschikt om direct op te werken: je kan er geen editor of compiler op draaien. Daarom worden applicaties voor een micro-controller ontwikkeld op een PC of vergelijkbaar systeem. Daar werk je dus met je editor en compiler. Het resultaat is een file die wordt ge-‘download’: in het Flash[[50]](#footnote-46) geheugen van de micro-controller wordt geladen. Deze manier van werken wordt cross-development genoemd.

## Arduino

De Arduino hardware is een micro-controller op een printje, gecombineerd met de electronica die nodig is om een PC in staat te stellen een applicatie op de micro-controller te ‘laden’ (downloaden). De oorspronkelijke Arduino Uno gebruikt een AtMega328 micro-controller. Er zijn inmiddels vele varianten en klonen van de Arduino, van uitgeklede versies met alleen de micro-controller (geen PC interface hardware voor downloaden) tot versies met een compleet Linux systeem naast de micro-controller. Alle Arduino’s gebruiken een opstelling van connectoren waar je een andere print (een zgn. shield) in kan prikken. Soms heeft zo’n shield ook weer die connectoren, je kan dan een toren van shields maken. Het is dan wel lastig om te zorgen de shields elkaar niet ‘in de weg’ zitten (bv. doordat ze de zelfde IO pinnen gebruiken).

[](http://www.google.nl/url?sa=i%26rct=j%26q=%26esrc=s%26source=images%26cd=%26cad=rja%26uact=8%26ved=0ahUKEwis0OuR2IbMAhUHnA4KHZEqBz4QjRwIBw%26url=http://cstart.org/forum/miscellaneous/an-arduino-based-solution-for-project-balloon/%26psig=AFQjCNE9QJR_kPKk6udoRP84mYO0TElTDA%26ust=1460467641837287)

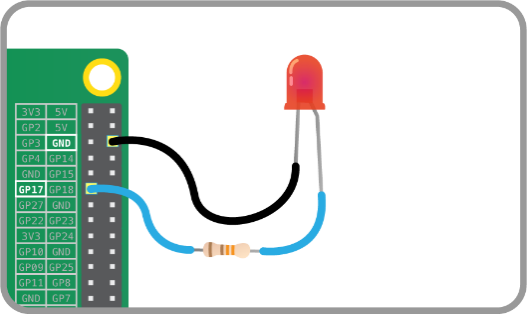
De Arduino IDE (Integrated Development Environment) biedt een redelijk laagdrempelige manier om een applicatie (in Arduino termen een ‘sketch’) te schrijven, te compileren en naar de Arduino te laden. Dit is erg polulair bij mensen die iets met een micro-controller willen doen, maar voor wie programmeren geen dagelijks werk is. Maar je wordt als gebruiker voor een deel afgeschermd van wat er echt op de processor gebreurt: de IDE voegt allerlei code toe die gedeeltelijk parallel met de gebruikerscode draait. De taal van Arduino scketches is in principe C++, maar er wordt meestal weinig gebruik gemaakt van de echte C++ features.

De Arduino versie die wij gebruiken is de Due. Deze is wat duurder dan de Uno, maar heeft een veel krachtigere micro-controller (sneller en meer geheugen) met meer IO pinnen. In China (aliexpress) kan je een Due kloon bestellen voor ongeveer € 15. De CPU van de Due micro-controller is een Cortex M3, die we programmeeren in C++. In een cursus in het tweede leerjaar zal je die ook programmeren in assembler.

## De ontwikkelomgeving

De Arduino IDE is populair, maar geoptimalizeerd voor mensen die niet vaak programmeren. Daardoor is een groot deel van de werking afgeschermd, en wordt de gebruiker beperkt tot een subset van C++. Daarom gebruiken wij voor het programmeren van de Due niet de Arduino IDE maar een editor/IDE (CodeLite), een cross-compiler (een GCC versie voor ARM procerssoren), en een aantal scripts, tools en libraries gebundeld als bmptk (Bare Metal Programming Tool Kit) en hwlib (hardware library). De details voor de installatie en gebruik hiervan vind je in de practicumhandleiding.

# GPIO



|  |
| --- |
| behandelde onderwerpen: |
| * GPIO * memory-mapped registers * volatile * namespace * LEDs en schakelaars aansluiten * hwlib pin & poort abstracties * argument-dependent lookup |

## GPIO als output

De eenvoudigste manier waarop een micro-controller in verbinding kan staan met zijn omgeving is via een General Purpose Input Output (GPIO) pin. General Purpose betekent ongeveer dat je er in je programma mee kan doen wat je wilt, maar je moet het wel helemaal zelf doen. Input Output betekent dat de pin naar keuze (van het programma) als input of als output gebruikt kan worden. Als het een input is dan kan je programma **lezen** of de pin laag of hoog is (dat kan bv. worden bepaald door een drukschakelaar). Als de pin een output is dan kan het programma **bepalen** of de pin laag of hoog is, waadoor bv. een aangesloten LED aan of uit gezet kan worden. Bij het aansluiten van een LED moet er altijd een weerstand met een geschikte waarde in serie met de LED worden aangesloten om de stroom te beperken.[[51]](#footnote-47) Zonder serie-weerstand zal het in eerste instantie wel werken, maar er is gevaar dat er te veel stroom gaat lopen waardoor de LED en/of de Due chip beschadigd kunnen raken.

|  |
| --- |
|  |
| Een LED aangesloten op een GPIO pin (groen kader is de Due chip) |

Het geheugen van een micro-controller bevat naast Flash (voor de code) en RAM (voor de run-time data) ook adressen die door de CPU gelezen en/of geschreven kunnen worden om de ‘peripherals’ van de chip aan te sturen. Voor je programma zijn dit een soort globale variabelen die eigenlijk stukjes hardware zijn. De werking van deze registers is uitputtend beschreven in de 1459 bladzijden tellende ‘AT91SAM ARM-based FLASH MCU SAM3X SAM3A series’ datasheet. Als je een peripheral van de chip wil gaan gebruiken (en je hebt geen bibliotheek beschikbaar die deze laag-niveau details voor je afhandelt) dan moet je het deel van de datasheet gaan lezen dat die peripheral beschrijft, bv. blz 970-1052 voor de PWM (pulse-breedte modulatie) peripheral.

Het gebruik van een GPIO pin beschreven in blz. 618-676 van de datasheet. Voor eenvoudig gebruik zijn daarvan maar een paar dingen nodig. De GPIO pinnen zijn gegroepeerd in **poorten**, voor onze chip vier stuks: PIOA, PIOB, PIOC en PIOD. Iedere poort bestuurt een aantal (maximaal 32) pinnen. Binnen iedere poort zijn er een aantal registers die de pinnen besturen. Bit 0 (het laagste bit) bestuurt pin 0, bit 1 bestuurt pin1, etc. Voor het aansturen van een LED zijn 3 registers nodig: OER, SODR en CODR. (Die afkortingen worden hierna uitgelegd).

Initieel zijn alle pins inputs. Als je naar het OER (Output Enable Register van een poort) schrijft, dan worden alle pinnen waar het bitje van op 1 staat outputs. De pinnen waarvan het bitje op 0 staat veranderen niet.

|  |
| --- |
|  |
| PIO\_OER beschrijving, datasheet p 633 |

Vervolgens je kan naar het SODR (Set Output Data Register) schrijven, en dan worden de pinnen waarvan het bitje in de waarde die je schrijft op 1 staan hoog gemaakt. Als de pin op dat moment een output is wordt die pin ook echt hoog (3.3 V). Is de pin geen output, dan merk je niets. Op de zelfde manier kan je een pin laag maken (mits het een output pin is) door een 1 bitje naar het (CODR) Clear Output Data Register te schrijven.

De datasheet geeft de namen van de peripheral registers en hun adressen in het geheugen (bv. 0x400E0E10 voor het OER register van IOA), maar de fabrikant levert ook een header file (sam.h[[52]](#footnote-48)) met de declaraties van de peripheral registers, die gebruikt kunnen worden alsof ze globale (volatile) variabelen zijn.

Op de print van de Due zit een LEDje aangesloten op pin B.27 (poort PIOB pin 27). Het volgende programma gebruikt de poort registers die in de header file van de fabrikant zijn gedeclareerd om deze LED te laten knipperen.De registers zijn gegroepeerd per functie, in de vorm van een struct die wordt geadresseerd door een pointer. De WDT pointer wijst naar de WatchDog Timer registers. De watchdog is er voor om als je programma vastloopt de chip te resetten. Wij gebruiken deze voorziening niet, maar omdat de watchdog standaard aan staat moeten we hem wel uitschakelen. (Anders wordt ons programma door de watchdog als ‘vastgelopen’ gezien en na een paar seconden gereset.)

Vervolgens wordt pin 27 van poort B een uitgang gemaakt, door een 1 te schrijven naar het betreffende bit (en alleen dát bit) van het Output Enable Register. De waarde die hier voor nodig is hadden we kunnen schrijven als 0x8000000 of zelfs als 134217728, maar dat is niet erg leesbaar. De expressie 0x01 << 27 geeft duidelijk aan dat er in die waarde maar 1 bitje gezet is, en dat de positie vabn dat bitje 27 is. 0x01 in plaats van 1 geeft aan dat het bitpatroon van belang is, niet de numerieke waarde.

Dan volgt een oneindige lus (typisch voor een embedded programma: er is geen Operating System om naar terug te keren). In die lus wordt de GPIO pin hoog gemaakt door 1 te schrijven naar het betreffende bit van het Set Output Data Register, en even later laag door de zelfde waarde te schrijven naar het Clear Output Data Register.

Tussen het hoog en laag maken van de pin moet enige tijd verstrijken. Hier wordt dat op de meest primitieve manier gedaan: door de CPU een for() lus een groot aantal keren te laten doorlopen. Het aantal keren hangt af van de CPU snelheid, de soort CPU, de compiler, etc. dus dit is niet de meest stabiele manier om een vertraging te realiseren, maar voor een eerste poging nu werkt het. Om te voorkomen dat de compiler de lus analyzeert en concludeert dat er niets in gebeurt (en de lus dus weggelaten kan worden) moeten we de teller als *volatile* declareren. Volataile betekent dat de compiler er niet op mag rekenen dat de variabele zich ‘normaal’ gedraagt, daardoor moet de compiler de lus intact laten. Het aantal iteraties door de lus is 100’000. Die ’ is een soort commentaar-teken binnen een constante: het wordt door de compiler genegeerd, maar het maakt het getal wel leesbaarder voor een mens (mits je het teken op de juiste plek zet…).

|  |  |
| --- | --- |
| #include "sam.h"  int main( void ){  // kill the watchdog  WDT->WDT\_MR = WDT\_MR\_WDDIS;  // make the GPIO pin an output  PIOB->PIO\_OER = 0x01 << 27;  for(;;){  // make the GPIO pin high  PIOB->PIO\_SODR = 0x01 << 27;  // wait some time  for( volatile int i = 0; i < 100'000; i++ ){}  // make the GPIO pin low  PIOB->PIO\_CODR = 0x01 << 27;  // again, wait some time  for( volatile int i = 0; i < 100'000; i++ ){}  }  } | |
| knipperen dmv. de poort registers en ‘busy wait’ delay loop | 15-01 |

## GPIO als input

Op een vergelijkbare manier kunnen we ook een pin die input is lezen. Een gemeen detail is dat we om dat te kunnen doen, eerst het clock signaal naar de GPIO peripherals moeten aanzetten. Iedere peripheral (hardware onderdeel van de micro-controller chip) heeft een bitje in de Periperal Clock registers, en dat bitje kunnen we aanzetten door een waarde naar het betreffende Peripheral Clock Enable Register te schrijven waarin dat bitje op 1 staat. Dat mag ook met meerdere bitjes tegelijk. Initieel staan de meeste peripheral clocks uit, om stroom te besparen. Voor het als output configureren van een GPIO, en het ernaar schrijven is (blijkbaar) geen clock signaal nodig, dus werkte het vorige programma zonder op dit detail te letten.[[53]](#footnote-49)

|  |
| --- |
|  |
| de Power Management Module (datasheet p528) |

De nummers van de bitjes die je moet zetten in de PMC registers staan weer elders in de datasheet, in de peripheral identifiers list.

|  |
| --- |
|  |
| Een deel van de Peripheral Identifiers List (datasheet, p38) |

Hieruit kunnen we dus opmaken dat als we de bitjes 11 … 16 aanzetten, alle GPIO peripherals geactiveerd zijn. Initieel zijn alle pinnen inputs, dus dat hoeven we niet in te stellen. Initieel hebben alle pinnen ook een ‘weak pull-up’, dwz. dat er intern een weerstand (van ~500 kΩ) is geplaatst tussen iedere GPIO pin en de 3.3V voeding. Daardoor is een pin die verder niet is aangesloten automatisch hoog, en we kunnen een schakelaar aanbrengen tussen een input pin en de massa (0V) om de pin laag te maken. Die schakelaar noemen we dan ‘active low’. De actuele waarde van een input pin kunnen we lezen in de Port Data Status Register: daarin komt iedere pin overeen met een bitje. De schakelaar verbinden we met de Due pin 7. (En de andere aansluiting van de schakelaar met de GND.) Let op: gebruik twee pinnen van de schakelaar die dicht bij elkaar zitten, of twee pinnnen die kruislings tegenover elkaar zitten, want de pinnen zijn twee-aan-twee met elkaar verbonden.

|  |
| --- |
|  |
| Een schakelaar aangesloten op Due pin D7 |

De Arduino gebruikt een volkomen andere pin nummering dan de chip zelf, dus moeten we Arduino pin D7 (= digitale pin 7) omzetten in een poort en pin nummer.

|  |
| --- |
|  |
| Arduino Due pins |

Op het pin diagram kunnen we zien dat Due pin 7 overeenkomt met Poort C pin 23 van de chip. Om die te lezen moeten we dus PIOC->PIO\_PDSR opvragen en daar het 23e bitje van nemen. Het onderstaande programma zet (voortdurend) de LED naar gelang dit bitje laag is (schakelaar ingedrukt, input bitje 0, LED pin hoog, dus LED aan) of hoog (schakelaar niet ingedrukt, input bitje 1, LED pin laag, LED uit).

|  |  |
| --- | --- |
| int main( void ){  // kill the watchdog  WDT->WDT\_MR = WDT\_MR\_WDDIS;  // enable the clock to all GPIO  PMC->PMC\_PCER0 = ( 0x3F << 11 );  // make the GPIO pin an output  PIOB->PIO\_OER = 0x01 << 27;  for(;;){  // check whether the switch is pressed  if (( PIOC->PIO\_PDSR & ( 0x01 << 23 )) == 0 ){  // make the GPIO pin high  PIOB->PIO\_SODR = 0x01 << 27;  } else {  // make the GPIO pin low  PIOB->PIO\_CODR = 0x01 << 27;  }  }  } | |
| copieer een aangesloten schakelaar naar de LED | 15-02 |

## Een abstracte GPIO superklasse

De ‘directe’ manier van aansturen maakt het lastig om code (bv. knipperen) in een library onder te bregen die onafhankelijk is van de specifieke GPIO pin(nen) die worden gebruikt. Om dit wel te bereiken kunnen we een C++ klasse maken die een GPIO pin voorstelt. Zo’n GPIO klasse moet een aantal operaties bieden:

* een mogelijkheid om de pin als input dan wel als output te laten functioneren
* een mogelijkheid om de waarde van de pin (als hij een output is) te zetten
* een mogelijkheid om de waarde van de pin (als hij een input is) op te vragen.

Het is wenselijk dat we code kunnen schrijven die werkt voor verschillende soorten GPIO pinnen, dus we gebruiken een klasse waarin deze mogelijkheden worden geboden door virtuele functies. Er is voor geen van deze functies een zinnige default te bedenken, dus ze zijn ook abstract, en daarmee is de klasse dus ook abstract. Van deze klasse leiden we een concrete klasse af die een GPIO pin implementeerd als een pin van de micro-controllerrontroller in de Due.

|  |  |
| --- | --- |
| D:\2017-2018\vakken\V1OOPC\images\GPIO\pin_in_out.png | namespace hwlib {  class pin\_in\_out {  public:  virtual void direction\_set\_input() = 0;  virtual bool get() = 0;  virtual void direction\_set\_output() = 0;  virtual void set( bool x ) = 0;  };  namespace target {  class pin\_in\_out : public hwlib::pin\_in\_out {  private:  . . .  public:  pin\_in\_out( int port, int pin ){ . . . }  void direction\_set\_input() override { . . . }  bool get() override;  void direction\_set\_output() override { . . . }  void set( bool v ) override { . . . }  };  };  }; |
| UML en C++ definitie voor een abstracte GPIO pin en een concrete ArduinoDue GPIO pin | |

## Namespace

Een namespace dient om ‘name-clashes’ te voorkomen: als je een aantal libraries gebruikt van verschillende bronnen, dan is er een is er een kans dat dezelfde naam in veschillende libraries wordt gebruikt. In C was het daarom de gewoonte de namen van alles wat in een library wordt gedeclareerd te beginnen met de naam van de library, bv. svn\_fs\_initialize() voor een functie in de svn library. Dit wordt in C++ ook wel eens gedaan, maar de moderne manier is om alle declaraties van je library onder in een namespace. Dit doe je door namespace <name> { } om je deklaraties heen te zetten. Als je iets uit een namespace wil gebruiken dan moet je de ‘scope resolution operator’ :: gebruiken.

|  |  |
| --- | --- |
| namespace my\_project {  class my\_class {  // usefull stuff here  };  };  int main( void ){  my\_project::my\_class x;  . . .  } | |
| gebruik van een namespace | 15-03 |

De (meeste) declaraties van de standaard C++ bibliotheek zitten in de namespace std::, bv. std::string en std::ostream.

## Hwlib

Hwlib is een C++ library voor de Arduino Due met oa. een abstracte GPIO klasse, en een implementatie daarvan voor de Arduino Due, zoals in de vorige tekst is besproken. De declaraties van de hwlib library zitten in de namespace hwlib. Om deze klassen te gebruiken moeten we eerst de hwlib library includen. De klasse voor een Due GPIO pin heet pin\_in\_out en zit in de target namespace binnen de hwlib namespace, dus de volledige naam is hwlib::target::pin\_in\_out. De library biedt ook een hwlib::wait\_ms() functie die het aangegeven aantal milliseconden wacht.

|  |  |
| --- | --- |
| #include "hwlib.hpp"  int main( void ){  // hwlib will kill the watchdog  // the on-board LED is connected to port B bit 27  auto led = hwlib::target::pin\_in\_out( 1, 27 );  led.direction\_set\_output();  led.direction\_flush();  while(1){  led.write( 1 );  led.flush();  hwlib::wait\_ms( 200 );  led.write( 0 );  led.flush();  hwlib::wait\_ms( 200 );  }  } | |
| blinken met gebruik van de hwlib library | 15-04 |

De hwlib library biedt overigens nog veel meer. De library is voorzien van Doxygen commentaar, maar je moet wel zelf Doxygen runnen om de html pagina’s te genereren.

De hwlib.hpp file zorgt er voor dat, afhankelijk van de chip waarmee je werkt, de juiste dingen (definities en declaraties) beschikbaar worden gemaakt. Dit wordt bepaald door de TARGET regel die in de gebruikte makefile staat. Die wordt door bmptk meegegeven als command-line parameter die bij het compileren worden meegegeven, en hwlib.hpp geberuikt die om de juiste file te includen.[[54]](#footnote-50) Voor deze cursus staan die regels in de makefile.due en makefile.native files in de workspace directory.[[55]](#footnote-51)

|  |  |
| --- | --- |
| # in makefile.due  TARGET := arduino\_due | |
| # in makefile.native  TARGET := native | |
| TARGET regels in makefiles |  |

De dingen in de hwlib::target namespace zijn afhankelijk van je gekozen target, de dingen in de rest van de hwlib namespace zijn het zelfde voor alle targets.

Het ‘copieren’ van de schakelaar naar de LED is eigenlijk nog eenvoudiger dan knipperen. Het aanzetten van de clock naar de GPIO’s en het killen van de watchdog wordt in de constructor van pin\_in\_out gedaan, dus dat hoeven we niet meer in de main te doen. De pin\_in\_out klasse heeft een tweede constructor die de pin namen van de Arduino print accepteert, dit maakt het makkelijker om te zien welke pinnen gebnruikt worden.

|  |  |
| --- | --- |
| int main( void ){  auto led = hwlib::target::pin\_in\_out( hwlib::target::pins::d13 );  led.direction\_set\_output();  led.direction\_flush();  auto sw = hwlib::target::pin\_in\_out( hwlib::target::pins::d7 );  sw.direction\_set\_input();  sw.direction\_flush();  while(1){  sw.refresh();  led.write( ! sw.read() );  led.flush();  }  } | |
| copieer een aangesloten schakelaar naar de LED met hwlib | 15-05 |

Het blinken van een LED is iets dat vaak terugkomt, dus het is handig om daar een aparte functie van te maken. Als we die functie een object van de abstracte klasse hwlib:pin\_in\_out meegeven, dan kunnen we niet alleen Due pinnen knipperen, maar ook andere soorten pinnen. Om een LED te knipperen moet je pin een output zijn, dus het is handig om de direction\_set\_output() aanroep ook in de blink functie te plaatsen, dan blijft de main mooi overzichtelijk.

Merk op dat de led parameter ‘by reference’ wordt overgedragen, en niet als ‘const’. We willen het echte object ‘knipperen’ (niet een copie ervan), vandaar ‘by reference’, en het zou raar zijn als knipperen het object niet zou mogen veranderen,[[56]](#footnote-52) vandaar *niet* ‘by const’.

|  |  |
| --- | --- |
| #include "hwlib.hpp"  void blink( hwlib::pin\_in\_out & pin ){  pin.direction\_set\_output();  pin.direction\_flush();  for(;;){  pin.write( 1 );  pin.flush();  hwlib::wait\_ms( 200 );    pin.write( 0 );  pin.flush();  hwlib::wait\_ms( 200 );  }  }  int main( void ){  auto led = hwlib::target::pin\_in\_out( 1, 27 );  ::blink( led );  } | |
| knipperen via een aparte functie | 15-06 |

De hwlib library bevat al een dergelijke knipper functie. De basis vorm daarvan vereist een parameter van het type hwlib::pin\_out. Er zijn namelijk ook pinnen die alleen maar een output kunnen zijn (bv. de pinnen van een HC595 schuifregister), en het zou onhandig zijn als een knipper functie geen gebruik zou kunnen maken van zulke pinnen. De klasse hwlib::target::pin\_out is vergelijkbaar met hwlib::target::pin\_in\_out, behalve dat de constructor de pin al als output zet. De aanroep van direction\_set\_output() is dus overbodig (de pin\_out klasse biedt die functie dan ook niet).

|  |  |
| --- | --- |
| #include "hwlib.hpp"  int main( void ){  auto led = hwlib::target::pin\_out( 1, 27 );  hwlib::blink( led );  } | |
| knipperen via hwlib::blink | 15-07 |

## Kitt

Een stapje ingewikkelder dan het knipperen van een enkele LED is het heen-en-weer oplichten van een reeks LEDs (Kitt[[57]](#footnote-53)-display, voor wie de Knightrider serie ooit gezien heeft). We kunnen dit natuurlijk ‘met de hand’ doen. Om het iets overzichtelijker te maken, gebruiken we de Due pin namen die in hwlib::target::pins gedeclareerd zijn: de klasse hwlib::target::pin\_out (en de andere pin klassen) heeft namelijk een tweede constructor die een waarde van die lijst accepteert.

|  |  |
| --- | --- |
| int main( void ){  . . .  auto led0 = hwlib::target::pin\_out( hwlib::target::pins::d7 );  auto led1 = hwlib::target::pin\_out( hwlib::target::pins::d6 );  auto led2 = hwlib::target::pin\_out( hwlib::target::pins::d5 );  auto led3 = hwlib::target::pin\_out( hwlib::target::pins::d4 );  for(;;){  led0.write( 1 ); led0.flush(); hwlib::wait\_ms( 80 ); led0.write( 0 ); led0.flush();  led1.write( 1 ); led0.flush(); hwlib::wait\_ms( 80 ); led1.write( 0 ); led0.flush();  led2.write( 1 ); led0.flush(); hwlib::wait\_ms( 80 ); led2.write( 0 ); led0.flush();  led3.write( 1 ); led0.flush(); hwlib::wait\_ms( 80 ); led3.write( 0 ); led0.flush();  led2.write( 1 ); led0.flush(); hwlib::wait\_ms( 80 ); led2.write( 0 ); led0.flush();  led1.write( 1 ); led0.flush(); hwlib::wait\_ms( 80 ); led1.write( 0 ); led0.flush(); }  } | |
| Kitt op de ‘doe-het-zelf’ manier | 19-08 |

De Due heeft maar 1 LED op de print, dus we moeten deze schakeling op een breadboard opbouwen. Let er op dat je voor iedere LED een serieweerstand gebruikt (1 kΩ : bruin-zwart-rood). Een LED heeft een langere en een kortere poot. De kortere poot verbindt je met de gnd. (Meestal is de LED behuizing aan die kant afgeplat.) Het is een goede gewoonte om voor de gnd de blauwe strips te gebruiken (de rode gebruik je voor de 3.3V).

|  |  |  |
| --- | --- | --- |
|  |  | Afbeeldingsresultaat voor led |
| Kitt opbouw |  | LED aansluitingen |

De code van de DIY kitt applicatie is erg lelijk: als het aantal LEDs veranderd moet je regels toevoegen of weghalen. (En de delay tijd is ook 6 keer apart aangegeven, maar dat zou makkelijk op te lossen zijn.) Het zou beter zijn als we de LEDs samenvoegen tot 1 object. Hiertoe heeft hwlib de poort abstracties: een poort is een reeks (maximaal 8) pinnen. De set() functie van een poort zet de pinnen van de poort volgens de bitjes in de waarde die je als parameter meegeeft: bv. 3 zet de laagste twee pinnen op 1 en de overige pinnen op 0. Je kan ook opvragen hoeveel pinnen er in de poort zitten. Daarmee kan je een kitt functie maken die onafhankelijk is van het aantal pinnen in de poort (tot het maximum van 8).

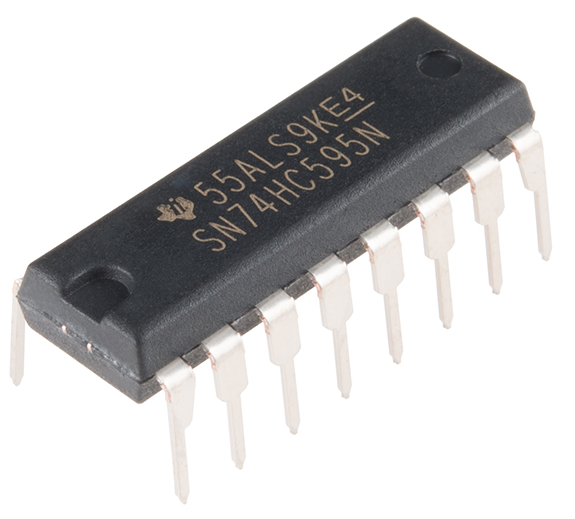
|  |  |
| --- | --- |
| void kitt( hwlib::port\_out & leds, int ms = 80 ){  for(;;){  for( unsigned int i = 0; i < leds.number\_of\_pins(); ++i ){  leds.write( 0x01 << i );  leds.flush();  hwlib::wait\_ms( ms );  }  for( unsigned int i = leds.number\_of\_pins() - 2; i > 0; --i ){  leds.write( 0x01 << i );  leds.flush();  hwlib::wait\_ms( ms );  }  }  }  int main( void ){  auto led0 = hwlib::target::pin\_out( hwlib::target::pins::d7 );  auto led1 = hwlib::target::pin\_out( hwlib::target::pins::d6 );  auto led2 = hwlib::target::pin\_out( hwlib::target::pins::d5 );  auto led3 = hwlib::target::pin\_out( hwlib::target::pins::d4 );    auto leds = hwlib::port\_out\_from( led0, led1, led2, led3 );  ::kitt( leds );  } | |
| Kitt via een poort | 19-09 |

Je moet de kitt functie aanroepen met :: ervoor, wat je kan lezen als ‘uit geen enkele namespace’. Omdat in hwlib ook al een zo’n kitt functie zit, en omdat het type van de parameter die je meegeeft uit hwlib komt, wordt die hwlib::kitt ook als ‘aanroepbaar’ gezien. Zonder de :: zou je een foutmelding krijgen omdat de compiler niet weet welke van de twee functies hij moet aanroepen.[[58]](#footnote-54)

Als je die herhaling van hwlib::target lelijk vind kan je daar een synoniem voor aanmaken, zoals in het volgende programma. Om het nog verder in te korten is daar de hwlib::kitt functie gebruikt.

|  |  |
| --- | --- |
| int main( void ){    namespace target = hwlib::target;    auto led0 = target::pin\_out( target::pins::d7 );  auto led1 = target::pin\_out( target::pins::d6 );  auto led2 = target::pin\_out( target::pins::d5 );  auto led3 = target::pin\_out( target::pins::d4 );    auto leds = hwlib::port\_out\_from( led0, led1, led2, led3 );  hwlib::kitt( leds );  } | |
| Kitt via hwlib:kitt | 19-10 |

# SPI : 74HC595



|  |
| --- |
| behandelde onderwerpen: |
| * 74HC595 chip * SPI * Dummy pin class * Kitt over verschillende ‘soorten’ pinnen |

Een Due heeft een groot aantal beschikbare I/O pinnen, maar soms is het handig om pinnen te gebruiken die niet direct aan de micro-controller zitten.[[59]](#footnote-55) Een eenvoudige chip die je kan gebruiken om ‘remote’ output pinnen te maken is het 74HC595 schuifregister. Deze chip heeft drie ingangen die aan de micro-controller moeten worden aangesloten: shift-clock (SHCP = SHift Clock Pulse), data (DS), en hold-clock (STCP = STorage Clock Pulse). Verder moet je natuurlijk GND en VCC (3.3V) aansluiten, en MR (Master Reset) aan VCC en OE (Outpout Enable) aan GND.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| De aansluitingen van een 74HC595 output schuifregister | | |

De manier waarop zo’n chip wordt aangestuurd heet SPI: Serial Peripheral Interface. Een standaard SPI bus heeft twee data lijnen: MOSI (Master Out Slave In) voor data van de master (micro-controller) naar de slave (in ons geval de HC595), en MISO (Master In Slave Out) voor data van de slave naar de master. Een reeks SPI chips kunnen de zelfde data (DS) en clock (SHCP) lijnen delen, maar ze moeten ieder een aparte select (SHCP) lijn hebben.

|  |  |
| --- | --- |
|  |  |

Het SPI protocol kan je op verschillende manieren implementeren. De Due chip heeft een peripheral (speciaal stukje hardware op de chip) dat het SPI protocol implementeert. Als je de maximale snelheid van dataoverdracht wil bereiken dan moet je deze peripheral gebruiken, maar dat is niet triviaal, en hoe je zo’n peripheral precies moet aansturen verschilt van chip tot chip. Daariom is dit in wwlib (nog) niet gedaan.[[60]](#footnote-56)

Het is redelijk makkelijk het SPI protocol in software uit te schrijven. Dit wordt wel bit-bangen genoemd. De hwlib heeft een zo’n ge-bitbangde implementatie van SPI. Omdat de constructor van deze SPI klasse 3 pinnen vereist (CLCK, MOSI, MISO) en je anders niet makkelijk kan zien welke pin wat is, zit de volgorde van de pinnen in de naam verwerkt. De eerste pin is de clock, de tweede MOSI (onze DS pin), en de derde de MISO pin, die wij niet gebruiken. Voor dit soort gevallen bevat hwlib een reeks dummy implementaties, wij kunnen hier de hwlib::pin\_in\_dummy gebruiken als derde parameter.

|  |  |
| --- | --- |
| namespace target = hwlib::target;  auto ds = target::pin\_out( target::pins::d8 );  auto stcp = target::pin\_out( target::pins::d10 );  auto spi\_bus = hwlib::spi\_bus\_bit\_banged\_sclk\_mosi\_miso(  shcp,  ds,  hwlib::pin\_in\_dummy  ); | |
| een SPI interface object aanmaken | 16-01 |

De declaratie van hwlib::pin\_dummy is overigens kinderlijk eenvoudig: de klasse erft van pin\_in, en implementeert een get() die altijd 0 teruggeeft.[[61]](#footnote-57)

|  |  |
| --- | --- |
| D:\2017-2018\vakken\V1OOPC\images\hc595\pin_in.png | class pin\_in {  public:  virtual bool get() = 0;  };  class \_pin\_in\_dummy\_class : public pin\_in {  public:  bool get() override { return 0; }  };  \_pin\_in\_dummy\_class pin\_in\_dummy;  class target::pin\_in : public pin\_in {  public:  bool get() override { . . . }  }; |
| UML en implementaties van hwlib::pin\_in | |

De constructor van de HC595 interface vereist twee parameters: een SPI bus, en een select pin. Het HC595 chip object dat je daarmee maakt is een subklasse van port\_out, dus je kan met de set() functie van de chip de 8 uitgangspinnen in 1 keer aansturen, zoals gedaan wordt door hwlib::kitt().

|  |
| --- |
|  |
| Opbouw van een HC595 met 8 LEDs |

|  |  |
| --- | --- |
| int main( void ){  . . .  namespace target = hwlib::target;  auto ds = target::pin\_out( target::pins::d8 );  auto shcp = target::pin\_out( target::pins::d9 );  auto stcp = target::pin\_out( target::pins::d10 );  auto spi\_bus = hwlib::spi\_bus\_bit\_banged\_sclk\_mosi\_miso(  shcp, ds, hwlib::pin\_in\_dummy );  auto leds = hwlib::hc595( spi\_bus, stcp );  hwlib::kitt( leds );  } | |
| Kitt op 8 LEDs aangesloten op een 74HC595 | 16-01 |

Daarnaast bevat het object ook 8 variabelen p0 .. p7 die je kan gebruiken om de 8 pinnen individueel te gebruiken, bv. om een enkele LED te laten knipperen.

|  |  |
| --- | --- |
| #include "hwlib.hpp"  int main( void ){  // kill the watchdog  WDT->WDT\_MR = WDT\_MR\_WDDIS;    namespace target = hwlib::target;  auto ds = target::pin\_out( target::pins::d8 );  auto shcp = target::pin\_out( target::pins::d9 );  auto stcp = target::pin\_out( target::pins::d10 );  auto spi = hwlib::spi\_bus\_bit\_banged\_sclk\_mosi\_miso(  shcp,  ds,  hwlib::pin\_in\_dummy  );    auto leds = hwlib::hc595( spi, stcp );    hwlib::blink( leds.p1 );  } | |
| Blink een LED aangesloten op een HC595 | 16-02 |

De ‘echte’ GPIO pinnen van de Due en de output pinnen van bv. een HC595 zijn afgeleid van hetzelfde basis type hwlib:pin\_out. Ze zijn dus uitwisselbaar, je kan ze bv. door elkaar gebruiken. Het volgende programma laat een Kitt zien op 6 Leds, 3 direct op de Due, en 3 op een HC595.

|  |
| --- |
|  |
| opbouw van 3 LEDs direct + 3 LEDs via een HC595 |

|  |  |
| --- | --- |
| int main( void ){  . . .  auto ds = target::pin\_out( target::pins::d8 );  auto shcp = target::pin\_out( target::pins::d9 );  auto stcp = target::pin\_out( target::pins::d10 );  auto spi = hwlib::spi\_bus\_bit\_banged\_sclk\_mosi\_miso(  shcp, ds, hwlib::pin\_in\_dummy  );  auto hc595 = hwlib::hc595( spi, stcp );  auto led0 = target::pin\_out( target::pins::d2 );  auto led1 = target::pin\_out( target::pins::d3 );  auto led2 = target::pin\_out( target::pins::d4 );  auto leds = hwlib::port\_out\_from(  led0,  led1,  led2,  hc595.p0,  hc595.p1,  hc595.p2  );  hwlib::kitt( leds );  } | |
| Kitt op 6 LEDs, 3 op de Due, 3 op een HC595 | 16-03 |

# Decorator en adapter

|  |  |  |
| --- | --- | --- |
| [https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSWGF8jXOAIuBc7scDXeW7UMj2p_tK91zfiGjSGgJSecjrT2T2P](http://www.google.nl/url?sa=i%26rct=j%26q=%26esrc=s%26source=images%26cd=%26cad=rja%26uact=8%26ved=0ahUKEwjh1dvcq9TLAhVDGg4KHdCNBhgQjRwIBw%26url=http://blog.chudinov.net/using-decorator-pattern-in-net-c/%26psig=AFQjCNGPR19roRJuZ_HgP-n8z2IT3NrcFg%26ust=1458737693909197) | [http://images.esellerpro.com/2295/I/246/77/ES988_1.jpg](http://www.google.nl/url?sa=i%26rct=j%26q=%26esrc=s%26source=images%26cd=%26cad=rja%26uact=8%26ved=0ahUKEwjA_Yq3idTMAhVC7hoKHc_zANQQjRwIBw%26url=http://www.sustuu.com/es988-universal-to-uk-3-pin-plug-visitor-travel-adaptor-for-usa-aus-eu-au-us-to-uk.html%26bvm=bv.121658157,d.ZGg%26psig=AFQjCNEBsWYtULqtrJVpIu62eWB4oZ02Mw%26ust=1463126420003583) | |
| behandelde onderwerpen: | |
| * Patterns * Decorator pattern * Adapter pattern | |

## Patterns

In de context van software engineering is een pattern een min of meer vaste manier om iets te doen, die in de loop der tijd zijn nut heeft bewezen. Er zijn enorm veel dingen die aanspraak maken op de term software pattern, maar er is maar een klein aantal (ongeveer 20) dat echt algemeen erkend en bekend verondersteld wordt.[[62]](#footnote-58) Het handige van de patterns is (behalve dat ze goed blijken te werken) dat je met weinig woorden (“dit is een decorator”) veel informatie geeft aan iemand die met het pattern bekend is. Dat bespaart jouw veel schrijfwerk, en de lezer veel leeswerk.

## Decorator

Het principe van het decorator pattern is dat je eerst een abstracte interface klasse hebt, bijvoorbeeld een GPIO pin. Om daar nuttig gebruik van te maken moet je natuurlijk ook een of meer concrete implementaties van die klasse hebben, bv. een GPIO pin van de Arduino Due.

Vervolgens kan je een klasse maken die een gegeven GPIO pin een beetje aanpast, bv. een inverter: als je 1 naar de pin schrijft, dan wordt de pin laag, en als je er een 0 naar schrijft dan wordt hij hoog (dus precies andersom als bij een normale pin). Je zou zo’n feature natuurlijk onder kunnen brengen in de abstracte klasse, maar dan moet je het in iedere concrete klasse implementeren. De truuk van het decorator pattern is dat je het maar 1 keer hoeft te schrijven, en dan als een soort decoratie om of over een ander object kan leggen. Een decorator is dus een implementatie van een abstracte klasse, die een (ander) object van die klasse kent, en iets wijzigt aan hoe de hij de opdrachten (functie aanroepen) die hij krijgt doorgeeft aan het onderliggende object.

|  |
| --- |
| D:\2017-2018\vakken\V1OOPC\images\patterns\decorator.png |
| UML voor een (algemene) decorator |

De C++ code voor zo’n inverter decorator is een klasse die een subklasse is van de interface. Hij krijgt in zijn constructor het ‘onderliggende’ object mee. De operaties die niet gewijzigd hoeven te worden zijn in de decorator geimplementeerd door simpleweg ‘door te verwijzen’ naar het onderliggende object. De twee operaties die wel gewijzigd zijn, set() en get(), verwijzen ook door, maar met een (in dit geval kleine) verandering.

|  |  |
| --- | --- |
| class pin\_in\_out\_invert : public hwlib::pin\_in\_out {  private:  hwlib::pin\_in\_out & slave;    public:  pin\_in\_out\_invert( hwlib::pin\_in\_out & slave ):  slave( slave ){}    void direction\_set\_input() override {  slave.direction\_set\_input();  }    void direction\_set\_output() override {  slave.direction\_set\_output();  }    bool read() override {  return ! slave.read();  }    void write( bool x ) override {  slave.write( ! x );  }  void direction\_flush() override {  slave.direction\_flush();  }    void refresh() override {  slave.refresh();  }    void flush() override {  slave.flush();  }    }; | |
| een inverter decorator voor de GPIO pin\_in\_out klasse | 17-01 |

In een eerder programma kopieerden we de een schakelaar naar een LED. Dat kopieren kunnen we onderbrengen in een functie.

|  |  |
| --- | --- |
| void copy\_pins(  hwlib::pin\_in\_out & destination,  hwlib::pin\_in\_out & source  ){  destination.direction\_set\_output();  source.direction\_set\_input();  for(;;){  source.refresh();  destination.write( source.read() );  destination.flush(); }  } | |
| kopieer de waarde van een source pin naar een destination pin | 17-01 |

Maar nu hebben we een probleem: de LED in is actief-hoog (de LED is aangesloten tussen de pin en ground), maar de switch pin is actief-laag (de schakelaar is aangesloten tussen de pin en ground, dus de pin is laag als de schakelaar is ingedrukt). Dit kunnen we oplossen door aan de copy\_pins functie niet de ‘originele’ LED en schakelaar pinnen mee te gaven, maar een van beiden te inverteren.

|  |  |
| --- | --- |
| int main( void ){  . . .  auto led = target::pin\_in\_out( target::pins::led );  auto sw\_pin = target::pin\_in\_out( target::pins::d7 );  // LED on when switch is \*not\* pressed  // copy\_pins( led, sw\_pin );  // LED on when switch is pressed  auto sw = pin\_in\_out\_invert( sw\_pin );  copy\_pins( led, sw );  } | |
| copieer schakelaar naar LED | 17-01 |

Een ander voorbeeld van een decorator is pin\_out\_all. Deze klasse krijgt in zijn constructor een aantal pin\_out objecten mee, en als je pin\_out\_all.set() aanroept worden set() functies van al die ‘slaaf’ pin\_out objecten aangeroepen. Het is lastig om in C++ een variabel aantal parameters mee te geven, maar voor een beperkt maximum aantal kan je hetzelfde effect bereiken door voor de parameters een geschikte (dummy) default te kiezen. Het is in C++ niet mogelijk een array van references te maken, dus worden de adressen van de ‘slaaf’ pin\_out’s opgeslagen in een array.

|  |  |
| --- | --- |
| class pin\_out\_all : public hwlib::pin\_out {  private:  std::array< hwlib::pin\_out \*, 4 > list;    public:  pin\_out\_all(  hwlib::pin\_out & p0,  hwlib::pin\_out & p1 = hwlib::pin\_out\_dummy,  hwlib::pin\_out & p2 = hwlib::pin\_out\_dummy,  hwlib::pin\_out & p3 = hwlib::pin\_out\_dummy  ):  list{ &p0, &p1, &p2, &p3 }  {}    void write( bool v ) override {  for( auto p : list ){  p->write( v );  }  }    void flush() override {  for( auto p : list ){  p->flush();  }  }  }; | |
| de pin\_out\_all decorator | 17-02 |

Met deze decorator kunnen we hwlib::blink gebruiken om meer dan 1 pin tegelijk te blinken.

|  |  |
| --- | --- |
| int main( void ){  . . .  auto led0 = target::pin\_out( target::pins::d7 );  auto led1 = target::pin\_out( target::pins::d6 );  auto leds = pin\_out\_all( led0, led1 );  hwlib::blink( leds );  } | |
| blink 2 LEDs tegelijk | 17-02 |

Met deze twee decorators (pin\_out\_all en pin\_out\_invert, een variatie op pin\_in\_out\_invert), kunnen we allerlei combinaties maken, bv het patroon xx--/--xx/xx--/--xx.

|  |  |
| --- | --- |
| int main( void ){  . . .  auto led0 = target::pin\_out( target::pins::d7 );  auto led1 = target::pin\_out( target::pins::d6 );  auto led2\_pin = target::pin\_out( target::pins::d5 );  auto led3\_pin = target::pin\_out( target::pins::d4 );  auto led2 = pin\_out\_invert( led2\_pin );  auto led3 = pin\_out\_invert( led3\_pin );  auto leds = pin\_out\_all( led0, led1, led2, led3 );  hwlib::blink( leds, 500 );  } | |
| blink 4 LEDs in twee clusters | 17-03 |

## Adapter

Een decorator biedt een ‘laagje’ over een object, zodat het zich iets anders gaat gedragen, maar de interface van de decorator en het ‘slaaf’ object is dezelfde. Bij adapter pattern is het gedrag van het ‘slaaf’ object en de adapter hetzelfde, maar de interface (bv. de naam van de functies, de types en volgorde van de parameters) is anders.

Een voorbeeld van een adapter is de implementatie van een (abstract) grafisch scherm op bv. windows. De window\_window klasse is een implementatie van de abstracte window klasse. De abstracte windows klasse hoeft maar 2 functies te implementeren: een window maken, en een pixel schrijven. Het wissen van de window kunnen we in de abstracte window klasse oplossen door alle pixels uit te zetten. De window\_window::draw functie maakt gebruik van de SetPixel functie in de Microsoft Windows library die hetzelfde doet, maar met heel andere parameters.

|  |
| --- |
| D:\2017-2018\vakken\V1OOPC\images\patterns\adapter.png |
| Een adapter: een abstracte window, en een implementatie daarvan op Microsoft Windows |

Behalve het veranderen van de parameters (de boolean waarde moet vervangen worden door twee RGB waarden) hoeft bv. de window\_window::draw functie eigenlijk niets te doen. (De code bevat nog twee for loops omdat we op windows de pixels wat groter willen maken.) Dit is typisch voor een adapter: de *interface* van de adapter is anders dan van het onderliggende object, maar de *functie* (wat er eigenlijk gebeurt) is meestal hetzelfde.

|  |  |
| --- | --- |
| void window\_window::draw( const xy & pixel, bool v ){  for( int dx = 0; dx < scale; dx++ ){  for( int dy = 0; dy < scale; dy++ ){  SetPixel(  hdc,  scale \* pixel.x + dx,  scale \* pixel.y + dy,  v ? RGB( 0x00, 0x00, 0x00 ) : RGB( 0xFF, 0xFF, 0xFF )  );  }  }  } | |
| window\_window::draw implementatie | 17-04 |

Een window is trouwens ook een goede kandidaat voor decorators. De mirror decorator splits de window in twee sub-windows, en print naar beide windows, maar geinverteerd.

|  |  |
| --- | --- |
| class window\_mirror : public hwlib::window {  private:  hwlib::window & w;  public:  window\_mirror( window & w ):  window( hwlib::xy( w.size.x / 2, w.size.y ) ),  w( w )  {}    void write\_implementation( hwlib::xy pixel, hwlib::color c ) override {  if( pixel.x <= ( w.size.x / 2 )){  w.write( pixel, c );  w.write( hwlib::xy( ( w.size.x - 1 ) - pixel.x, pixel.y ), c );  }  }    void flush() override {  w.flush();  }  }; | |
| een mirror decorator | 17-05 |

|  |  |
| --- | --- |
| int main(){  hwlib::target::window big( hwlib::xy( 128, 64 ), 2 );  auto w = window\_mirror( big );    line top( w, hwlib::xy( 0, 0 ), hwlib::xy( 63, 0 ) );  line right( w, hwlib::xy( 63, 0 ), hwlib::xy( 63, 63 ) );  line bottom( w, hwlib::xy( 0, 63 ), hwlib::xy( 63, 63 ) );  line left( w, hwlib::xy( 0, 0 ), hwlib::xy( 0, 63 ) );  ball b( w, hwlib::xy( 50, 20 ), 9, hwlib::xy( 5, 2 ) );    std::array< drawable \*, 5 > objects = { &b, &top, &left, &right, &bottom };  for(;;){  w.clear();  for( auto & p : objects ){  p->draw();  }  w.flush();  hwlib::wait\_ms( 200 );  for( auto & p : objects ){  p->update();  }  for( auto & p : objects ){  for( auto & other : objects ){  p->interact( \*other );  }  }  }  } | |
| gebruik van de mirror decorator | 21-05 |

# Hardware interfacing via I2C



|  |
| --- |
| behandelde onderwerpen: |
| * I2C * Aansturen OLED schermpje * NVI pattern |

Bij SPI worden 3 aansluiting gedeeld door de slave chips die zijn aangesloten op de SPI bus, en daarnaast is er nog 1 individuele ‘chip select’ aansluiting per chip. De I2C (Inter Integrated Circuit) bus is een andere manier waarop chips met elkaar kunnen communiceren. I2C gebruikt maar 2 aansluiting voor de bus, en geen individuele select aansluiting. Het vergt dus minder ‘hardware’ (IO pinnen, draden), maar het is daardoor wel beduidend langzamer dan SPI, en het protocol is complexer.

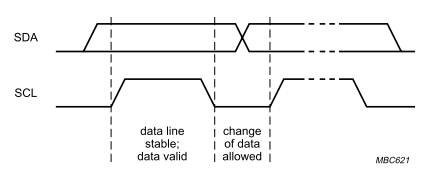
## I2C

I2C gebruikt twee lijnen: SDA (Serial DAta) en SCL (Serial CLock). Beide lijnen hebben een ‘pull-up’ weerstand. De chips kunnen deze een lijnen laag trekken (met de ground verbinden), maar niet hoog trekken (dat doen de weerstanden). Deze opzet wordt open-collector (of open-drain) genoemd.

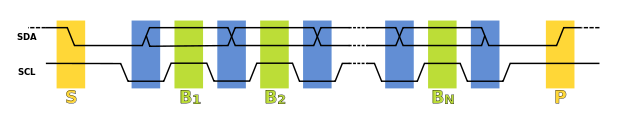
|  |  |
| --- | --- |
|  |  |

De I2C bus is oorspronkelijk ontwikkeld en gepatenteerd door Philips. Hierdoor moesten andere fabrikanten die het wilden toepasen een vergoeding betalen. Veel fabrikanten vermeden dit door een protocol te gebruiken dat helemaal overeen kwam met I2C, maar niet zo genoemd werd. Inmiddels is het patent van Philips verlopen, maar je ziet nog vaak in de beschrijving van een chip “deze chip implementeert een serieel twee-draads protoco (TWI)”. In veel gevallen betekent dit gewoon dat de chip I2C implementeert.

Op bit niveau gebruikt I2C een puls op de clock (SCL) lijn om 1 bit (op de SDA lijn) over te dragen. Het protocol eist dat de SDA lijn stabiel is (niet verandert) terwijl de clock lijn verandert of hoog is. De bits van een byte worden een voor een overgedragen, het hoogste (meest significante) bit eerst. De SCL lijn wordt altijd aangestuurd door de master (meestal een micro-controller), de SDA lijn wordt aangestuurd door het device dat de data verzendt. Dit kan dus zowel de master als een slaaf zijn.[[63]](#footnote-59).



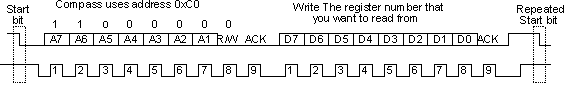
Veranderingen van de SDA lijn terwijl de SCL lijn hoog is worden gebruikt om speciale dingen over te brengen: de S (Start) conditie (SDA wordt laag terwijl SCL hoog is) en de P (stoP) conditie (SDA wordt hoog terwijl SCL hoog is). S geeft het begin van een I2C transactie aan (je kan het zien als een soort reset), P geeft het einde van een transactie aan.



Het eerste byte van een transactie is altijd een commando byte. Dit bevat een 7-bit adres, en 1 bit dat aangeeft of de transactie een write (de master zendt data naar de slaaf) of een read (de slaaf gaat data naar de master zenden) is. Het bitje is 1 voor een read, 0 voor een write. Na ieder byte wordt er een acknowledge bit verzonden door de partij die het voorafgaande byte heeft ontvangen. Voor het commando byte is dit dus de slaaf.

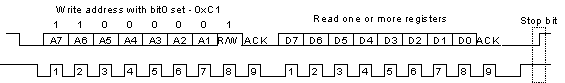
|  |  |
| --- | --- |
|  |  |

Het volgende diagram toont een 1-byte schrijf actie naar een slaaf chip met adres 1100000b. Het 8e bitje is laag, dus het is een write. Na de 8 bitjes die verzonden zijn door de master volgt 1 acknowledge bitje van de slaaf naar de master. Vervolgens zendt de master data 1 byte, en de slaaf zendt weer een acknowldge. Daarna had de master een P kunnen zenden en daarna een S om de volgende transactie te starten, maar in dit plaatje zend de master een ‘repeated start bit’, dit is een soort combinatgie van P en S.



Om iets uit een slaaf te lezen moet de master een read transactie gebruiken. Na het commando byte zal de slaaf dan data bytes naar de master sturen, net zo lang todat de master de transactie beeindigt.

Merk op dat er in een read transactie geen gelegenheid is voor de master om aan te geven welke data hij van de slaaf wil lezen. Bij een heel simpele chip (die maar 1 of een paar data bytes kan leveren) is dat geen probleem, maar anders is het erg inefficient om altijd alle data bytes van de slaaf te moeten lezen. De gebruikelijke oplossing hiervoor is dat een write transactie altijd begint met een aantal adres bytes, en dat dit adres in de slaaf onthouden wordt (ook als er 0 bytes naar dat adres geschreven zijn). Bij een volgende read transactie wordt dan van *dat* adres gelezen.



Net als SPI doet I2C geen uitspraak over wat de bytes die overgestuurd worden betekenen, dat is aan de chip zelf. Dit staat doorgaans beschreven in de datasheet van de chip.

I2C kent een aantal variaties die je tegen zou kunnen komen:

* 10-bit adressen: een techniek die meer adressen toestaat dan de 128 die mogelijk zijn bij 7-bits adressen;
* ‘multi-master’: een techniek waarbij er meerdere masters op de bus aangesloten zijn, die onderling uitmaken hoe ze het gebruik van de bus verdelen;
* ‘high-speed I2C’ (400 kHz and 3.4 MHz): alle I2C chips ondersteunen een klok tot 100 kHz, sommige chips ondersteunen ook hogere clock frequenties.

## OLED

Hwlib heeft een abstracte window klasse, die een minimale interface specificeert voor een grafische window. Je kan van zo’n window de afmetingen opvragen, de window wissen, en een pixel schrijven.

|  |
| --- |
| class window {  public:  const location size;    void write( location pos, color col );  void flush();    virtual void clear();  }; |
| Hwlib window interface klasse (vereenvoudigd) |

Een OLED is een schermp(je) dat voor iedere pixel een (organisch) LEDje bevat. Wij gebruiken een klein OLED display met 128 x 64 pixels dat wordt aangestuurd via I2C. Het Due bordje heeft twee pinnen (D21 en D20) die gemarkeerd zijn als SCL en SDA, dus het ligt voor de hand die pinnen te gebruiken. Verder heeft het displaytje nog ground en 3.3V nodig. De vier pinnen van het display moeten worden aangesloten volgens onderstaande tabel. Let op: er zijn verschillende versies van het OLED display, die verschillen in de volgorde van de pinnen. Let ook op dat je het OLEDje niet per ongeluk op 5V aansluit, daar kan hij niet tegen.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | --- | --- | | **Arduino** | **OLED** | | GND | GND | | 3.3V | VDD | | SDA (D20) | SDA | | SCL (D21) | SCK | |
| Het aansluiten van het OLED display | |

Net als voor SPI is er in hwlib een ge-‘bit-bangde’ I2C implementatie (dwz. een implementatie waarin alle acties op de GPIO pinnen in code zijn gerealiseerd). Deze hwlib::i2c\_bus\_bit\_banged\_scl\_sda klasse vereist twee open-collector pinnen. Hiervoor bevat hwlib klassen die een GPIO pin zich laat gedragen als open-collector pin.

Hwlib heeft een abstracte window klasse, die een minimale interface specificeert voor een grafische window. Je kan van zo’n window de afmetingen opvragen, de window wissen, en een pixel schrijven.

|  |
| --- |
| class window {  public:  const location size;    void write( location pos, color col );    virtual void clear();  }; |
| Hwlib window interface klasse (vereenvoudigd) |

Hwlib heeft een klasse voor het OLED dat de windows interface implementeert. Je kan deze klasse zien als een adapter van de windows interface naar het I2C protocol dat het OLED schermpje vereist.[[64]](#footnote-60) Hiermee kunnen we iets eenvoudigs doen met het display. Het volgende voorbeeld tekent een horizontale lijn.

|  |  |
| --- | --- |
| int main( void ){    // kill the watchdog  WDT->WDT\_MR = WDT\_MR\_WDDIS;    namespace target = hwlib::target;    auto scl = target::pin\_oc( target::pins::scl );  auto sda = target::pin\_oc( target::pins::sda );    auto i2c\_bus = hwlib::i2c\_bus\_bit\_banged\_scl\_sda( scl,sda );    auto display = hwlib::glcd\_oled( i2c\_bus, 0x3c );    display.clear();  for( int i = 10; i < display.size.x - 10; i++ ){  display.write( hwlib::location( i, 10 ), hwlib::black );  }  } | |
| OLED demo : teken een lijn | 18-02 |

In de volgende applicatie wordt de hwlib::graphics\_random\_circles demo gebruikt.

|  |  |
| --- | --- |
| #include "hwlib.hpp"  int main( void ){  . . .  auto display = hwlib::glcd\_oled{ i2c\_bus, 0x3c };  hwlib::graphics\_random\_circles( display );  } | |
| OLED demo | 18-03 |

We kunnen ook de windows\_mirror gebruiken in de demo, dan krijgen we een testje dat demo gespiegeld toont in de linker- en rechterhelft van het OLEDje.

|  |  |
| --- | --- |
| class window\_mirror : public hwlib::window {  private:  hwlib::window & w;  public:  window\_mirror( hwlib::window & w );  private:  void write\_implementation( hwlib::location pos, hwlib::color col ) override;  };  int main( void ){  . . .  auto display = hwlib::glcd\_oled( i2c\_bus, 0x3c );  auto mirrored = window\_mirror( display );  hwlib::graphics\_random\_circles( mirrored );  } | |
| Gespiegelde cirkels | 18-03 |

## NVI

De windows klasse van bmptk is een window met een bepaalde omvang. Die wordt bij het creeeren van de window aan de constructor meegegeven en opgeslagen in het window object.

|  |
| --- |
| class window {  . . .  public:  const location size;  window( location size )  : size{ size }  {}  . . .  }; |
| Window constructor |

Bij het tekenen van een pixel bestaat de mogelijkheid dat de aanroeper een coordinaat meegeeft dat buiten de window valt. In dat geval moet de pixel niet getekend worden. Deze check zou in iedere implementatie van de write functie gedaan kunnen worden, maar het is logischer om dit op 1 plaats te doen. Dit is in de window klasse gerealiseerd doordat de write() functie deze check doet, en vervolgens (als de pixel binnen de window valt) de write\_implementation() functie aanroept.

De write\_implementation hoeft dus niet meer te checken of de pixel wel binnen de window valt.

Dit is een toepassing van het NVI pattern: de interface naar de buitenwereld is een Niet Virtuele Interface, en handelt alvast een deel van de taak af, en laat dan de rest over aan een (private) functie die wel virtueel is.

|  |
| --- |
| class window {  private:  // write a pixel - implementation  virtual void write\_implementation( location pos, color col ) = 0;  public:  . . .  // write a pixel - interface  void write( location pos, color col ){  if( ( ! col.is\_transparent )  && ( pos.x >= 0 ) && ( pos.x < size.x )  && ( pos.y >= 0 ) && ( pos.y < size.y )  ){  write\_implementation( pos, col );  }  }  }; |
| NVI: Non Virtual Interface |

# Appendix: Weekschema

|  |  |  |
| --- | --- | --- |
| **Week** | **Behandele stof** | **Practicum** |
| 1 | Reader t/m hoofdstuk 4 (Klassen en compositie) | Software installeren  Rechthoek & samengesteld object |
| 2 | Reader t/m hoofdstuk 9 (Doxygen) | ADT, Doxygen, Catch |
| 3 | Reader t/m hoofdstuk 12 (Dingen en waarden) | Muren, stuiteren, verdwijnen |
| 4 | Reader t/m hoofdstuk 14 (GPIO) | Installeren, patroon, input |
| 5 | Reader t/m hoofdstuk 16 (Decorator en adapter) | HC595, decorator, knipperen |
| 6 | Reader t/m hoofdstuk 17 (Hardware interfacing via I2C) | OLED |
| Voorbeeldtentamen bespreken | |

# Appendix : Verschillen tussen de C++ standaarden

C++ standaarden zijn genoemd naar het jaar waarin de standaard is uitgekomen. Voor 1998 was er geen officiele standaard. De tabel geeft een beknopt overzicht. De werkelijk geimplementeerde features kunnen per compiler verschillen. Zie bv. voor meer details, specifiekl voor GCC, <https://gcc.gnu.org/projects/cxx-status.html>

|  |  |
| --- | --- |
| **Versie** | **(nieuwe) features** |
| C++ 98 | Eerste echte C++ standaard. |
| C++ 03 | Alleen kleine wijzigingen en correcties op C++ 98. |
| C++ 11 | Veel aanvullingen, waardoor het volgens veel mensen bijna een andere taal is geworden:   * Rvalue references, move smantics * Constexpr * Lambda’s * Auto, decltype * for( : ) syntax * override, final, explicit, using, default, deleted * nullptr * enum class * variadic templates * user-defined constants * long long int * std::array<> * user-defined literals |
| C++ 14 | Beperkte aanvullingen op de vorige standaard:   * Constexpr mogelijkheden verruimd * Lambda mogelijkheden verruimd * Binary literal, ‘ literal separator |
| C++ 17 | Er is wat meer nieuw dan in 14, maar minder dan in 11:   * Folding expressions * auto [ a, b ] = * if constexpr * template parameter deduction voor constructors * inline variables (variables in header files) * byte, std::optional<> |
| C++ 20 | De nieuwe standaard wordt verwacht in 2020, en naar verwachting zal deze nog meer gevolgend hebben dan C++ 11. Enkele verwachte zaken:   * concepts (eindelijk) * values of user-defined types as template parameters * reflectie * coroutines * misschien een bruikbare ‘freestanding’ versie van de libraries |

# Appendix : Index en endnotes

#define, 13, 23, 24, 42

#include, 13, 14, 15, 17, 18, 36, 37, 41, 42, 72, 77, 78, 79, 85, 98

74HC595, 82, 85

abstract, 52, 57, 58, 59, 60, 75, 91

Abstract Data Type. *See* ADT

adapter, 87, 91, 97

Adapter, 87, 91

ADT, 25, 32, 46, 64, 65, 101

Arduino, 4, 18, 66, 68, 74, 77, 87, 97

Arduino Due, 4, 18, 66, 74, 77, 87

Argument Dependent Lookup, 81

assert, 40, 41

associatie, 8, 14, 16, 17

attributen, 11, 12, 14, 17, 27, 29, 30, 32, 33, 41, 45, 46, 53

auto, 48, 50, 51, 61, 63, 64, 77, 78, 79, 80, 81, 83, 85, 86, 89, 90, 91, 93, 98, 99, 102

bmptk, 4, 66, 68, 77, 99

brief, 45, 46

Catch2, 4, 40

class, 8, 12, 13, 15, 17, 25, 26, 27, 28, 29, 30, 31, 32, 33, 46, 53, 54, 55, 56, 57, 59, 60, 62, 63, 76, 79, 82, 84, 88, 90, 92, 96, 97, 99, 100, 102

CLCK, 83

CodeLite, 4, 66, 68

CODR, 70, 72, 75

commando byte, 95, 96

compositie, 8, 14, 17

const, 19, 20, 21, 24, 28, 29, 32, 33, 41, 46, 48, 49, 50, 51, 54, 59, 62, 63, 78, 90, 92, 96, 97, 99

constructor, 4, 8, 10, 11, 12, 13, 14, 15, 16, 17, 28, 29, 30, 31, 46, 50, 53, 54, 62, 78, 79, 83, 84, 88, 89, 99

decorator, 87, 88, 89, 90, 91, 92, 93, 101

Decorator, 87, 101

default constructor, 29

default waarde, 31

details, 45, 46

diadic, 28, 29

Dingen, 64, 101

Doxygen, 4, 45, 46, 47, 77, 101

DRY, 33

Due, 4, 68, 69, 70, 73, 74, 75, 77, 78, 79, 80, 82, 83, 85, 86, 96

for(, 21, 22, 23, 24, 48, 49, 50, 51, 55, 61, 63, 71, 72, 75, 80, 81, 89, 90, 92, 93, 98, 102

getter, 26, 27

GPIO, 69, 70, 71, 72, 73, 75, 76, 77, 78, 85, 87, 88, 97

header file, 13, 70

hwlib, 68, 69, 76, 77, 78, 79, 80, 81, 83, 84, 85, 86, 88, 89, 90, 91, 97, 98, 99

I2C, 94, 95, 96, 97

IDE, 4, 68

implementatie file, 13

initialisatielijst, 15

inverter, 87, 88

Java, 4, 7, 27, 58

Kitt, 79, 80, 81, 82, 85, 86

macro, 19, 23, 24, 41, 42

master, 82, 95, 96

*member functie*, 10

*methode*, 10, 14, 28, 31, 33, 34, 44, 58

micro-controller, 66, 67, 68, 69, 72, 82, 95

mirror, 92, 93, 98, 99

MISO, 82, 83

monadic, 28, 29, 33

MOSI, 82, 83

namespace, 37, 69, 76, 77, 79, 81, 83, 85, 98

NVI, 94, 99, 100

OER, 70, 72, 75

OLED, 4, 94, 96, 97, 98, 101

OO, 6, 10, 12, 53, 64

**operator+**, 28, 29, 32, 33, 34, 35, 41

operatoren, 16, 25, 28, 29, 32, 33, 34, 35, 36, 37, 65

operatror+=, 33

overerving, 5, 52, 53

overloaded, 9, 29

parameter, 9, 10, 16, 17, 19, 20, 21, 22, 23, 25, 28, 29, 31, 33, 34, 42, 46, 47, 49, 51, 55, 59, 77, 78, 79, 80, 81, 83, 102

Patterns, 87

pointer, 16, 20, 32, 50, 56, 57, 59, 60, 61, 71

private, 8, 11, 12, 13, 14, 15, 17, 25, 26, 33, 53, 54, 59, 63, 76, 79, 88, 90, 92, 99, 100

protected, 52, 55, 59, 62

public, 8, 11, 12, 13, 14, 15, 17, 25, 26, 27, 28, 29, 30, 31, 32, 33, 46, 52, 53, 54, 55, 56, 57, 59, 60, 62, 63, 76, 79, 84, 88, 90, 92, 96, 97, 99, 100

publiek, 11, 27, 55

pull-up, 73, 94

PWM, 70

Python, 4, 7, 10, 27, 58

range-based-for, 49

Real-Time, 67

reference, 8, 16, 17, 19, 20, 21, 28, 32, 37, 39, 49, 50, 51, 56, 57, 59, 78

REQUIRE, 42, 43, 44

RTOS, 67

sam.h, 70, 72, 77

SCL, 94, 95, 96, 97

SDA, 94, 95, 96, 97

setter, 26, 27

short-circuit, 34, 35

SODR, 70, 72, 75

SPI, 82, 83, 84, 94, 96, 97

std::array<>, 21

std::vector<>, 21

stringstream, 44

struct, 8, 9, 10, 11, 12, 71

subklasse, 53, 55, 56, 57, 59, 84, 88

superklasse, 53, 54, 55, 56, 57, 58, 59, 61, 62, 75

supertype, 55

TEST\_CASE, 42, 43, 44

this pointer, 32

TWI, 94

UML, 4, 12, 14, 16, 17, 18, 26, 52, 53, 54, 58, 62, 76, 79, 84, 88

undefined behaviour, 30

unit test, 40, 42

Virtual, 57, 58, 100

waarden, 8, 9, 15, 18, 21, 22, 24, 25, 27, 28, 29, 30, 31, 32, 34, 36, 37, 38, 40, 41, 42, 43, 49, 51, 64, 91

watchdog, 71, 72, 75, 77, 78, 79, 85, 98

1. Een deel van deze materialen was al benodigd voor eerdere cursussen. [↑](#footnote-ref-0)
2. Een ‘de facto’ standaard is een standaard die niet door een standaardizeringsorganizatie is gedefinieerd, maar op een andere manier zo algemeen in gebruik is geraakt dat het in feite de standaard is. [↑](#footnote-ref-1)
3. Maar tegenstanders van C zeggen in plaats van stabiel liever *verouderd*. [↑](#footnote-ref-2)
4. Voor de lezer. De compiler kan het niets schelen wat voor naam je aan een functie geeft. [↑](#footnote-ref-3)
5. De . . . in een voorbeeld geeft aan dat er code is weggelaten. [↑](#footnote-ref-4)
6. De verwijzing rechtsonder geeft aan in welk voorbeeld deze code terug te vinden is. [↑](#footnote-ref-5)
7. Zo’n nieuw datatype is vrijwel gelijkwaardig aan de ingebouwde datatypen (int, float, etc.). [↑](#footnote-ref-6)
8. In C++ kunnen (anders dan in C) meedere functies zijn met dezelfde naam, mits ze verschillende parameters hebben. Dit heet overloading. Er zijn interessante discusies over het gebruik van overloading, sommigen (met name programmeurs die gewend zijn aan een taal zonder overloading) vinden het een probleem dat je aan de naam van een (overloaded) functie niet meteen kan zien welke functie wordt aangeroepen. [↑](#footnote-ref-7)
9. Die lijn parameter is er dus eigenlijk nog wel, maar je ziet hem niet in de declaratie, en hij wordt impliciet doorgegeven. Dit is een veel voorkomende keuze in OO talen, maar het kan ook anders: bv. in Python is het doorgeven impliciet, maar de parameter is expliciet (staat gewoon in de parameter lijst). [↑](#footnote-ref-8)
10. In C++ hoef je voor een lege parameterlijst niet zoals in C (void) te schrijven (dat mag nog wel): alleen () is voldoende. [↑](#footnote-ref-9)
11. Deze syntax wordt door de compiler afgedwongen: bij het creeeren van een variabele die een of meer constructors heeft moet een van die constructors worden aangeroepen. [↑](#footnote-ref-10)
12. In C zouden we om het scherm object zelf (dus niet een kopie) door te geven een pointer naar het scherm doorgeven (en in de tweede oplossing ook opslaan). In C++ kunnen we in zo’n geval een reference doorgeven (en desgewenst ook opslaan). [↑](#footnote-ref-11)
13. Een reference lijkt op een pointer, maar een reference kan je niet naar een ander object laten wijzen, je kan er niet mee rekenen, en je hoeft geen adres-van (&) en pointer-dereference (\*) operatoren te gebruiken. [↑](#footnote-ref-12)
14. Vaak kan je kiezen of je klasse variabelen hun initiele waarden geeft in de initializatie lijst, of in de body (met assignments). Over het algemeen heeft het de voorkeur dit te doen in de initializatielijst, omdat dat effectiever is (anders wordt de variabele eerst default-geconstrueert en daarna ge-assigned). Ook komt het voor dat een datatype wél een constructor heeft maar géén assignment, zodat de initializatielijst de enige optie is. (Andersom, dus wel assignment maar geen constructor die een specifieke waarde geeft, komt eigenlijk niet voor.) [↑](#footnote-ref-13)
15. Andere interfaces zijn ook mogelijk, zoals ‘maak deze pixel kleur k’, dan heb je geen aparte wis-functie nodig. [↑](#footnote-ref-14)
16. De term *variabele* is wat vreemd voor iets wat niet variabel (wijzigbaar is), maar zo is de terminologie. [↑](#footnote-ref-15)
17. Er wordt algemeen aangenomen dat code veel vaker gelezen wordt dan geschreven. Het is dus een goed principe om code te optimalizeren voor leesbaarheid, dus niet noodzakelijk voor ‘schrijfbaarheid’. [↑](#footnote-ref-16)
18. Maar ipv. die constante te gebruiken kan je (in dit geval) nog beter de size() van het array opvragen, dan kan je je ook niet vergissen in de naam van de constante. [↑](#footnote-ref-17)
19. C kent tegenwoordig ook const, maar het gebruik van #define voor dit doel is nooit helemaal verdwenen. [↑](#footnote-ref-18)
20. Helemaal sluitend is dit argument niet, want op een kleine processor en met een klein scherm is het wellicht beter om ruimte te besparen door 8-bit waarden op te slaan in plaats van integers die minimaal 16 bit zijn. En op grotere system is het soms juist handiger om doubles te gebruiken voor schermcoordinaten, omdat je dan met meer dan pixel-naukeurigheid kan rekenen met de plaats van een object. [↑](#footnote-ref-19)
21. Behalve voor ingebouwde operatoren typen (bv. int), daarvoor is alleen de a + b schrijfwijze toegestaan. [↑](#footnote-ref-20)
22. . In C is dat niet mogelijk, daar moet iedere functie een unieke naam hebben. [↑](#footnote-ref-21)
23. ‘Undefined behaviour’ betekent dat echt alles kan gebeuren. Volgens Murphy werkt alles dus prima als je het zelf test, maar het crasht zodra je er iets belangrijks mee wil doen, bv. demonstreren voor een klant of laten aftekenen bij je docent. [↑](#footnote-ref-22)
24. De meningen zin verdeeld over wat een goede deafult beginwaarde is. Eén stroming pleit voor een ‘logische, handige, neutrale’ waarde, bv. 0 voor getallen, en een lege string voor strings. Het idee is dat je de programmeur werk uit handen neemt door een variabele bij initializatie als de meest logische beginwaarde te geven. Hier tegenover staat de opvatting dat een programmeur altijd zelf de beginweaarde moet aangeven. Als hij dat vergeet is dat een fout, en de automatische beginwaarde moet die fout zo duidelijk mogelijk maken. Voor getallen kan je dan bv. voor -1 kiezen (voor floats NaN: de speciale not-a-number waarde), en voor strings ‘?’ op alle posities. De automatische beginwaarde dient dan als signaal-waarde bij het debuggen, en zou in de definitieve (productie) versie van de applicatie misschien zelfs weggelaten kunnen worden. [↑](#footnote-ref-23)
25. Beredeneer dit zelf: kan je een functie f( int x = 0, int y ) zodanig aanroepen dat de default waarde voor x gebruikt wordt? [↑](#footnote-ref-24)
26. Je krijgt dus geen foutmelding (maar, afhankelijk van de compiler en de gebruikte compiler opties, misschien wel een warning) bij het definieren van die twee constructors. [↑](#footnote-ref-25)
27. Pointers worden in een volgend vak besproken. Je kan een pointer zien als een soort reference waarbij je met een \* exliciet moet aangeven dat je het aangewezen object bedoelt, en niet de pointer zelf. Bij een reference gaat dat automatisch. [↑](#footnote-ref-26)
28. Het principe ‘schrijf alles maar één keer’ heeft zelfs een algemeen erkende afkorting: DRY (Don’t Repeat Yourself). [↑](#footnote-ref-27)
29. Je kunt ook een operator+() met 1 parameter maken (in het geval van een klasse functie dus géén), dat is dan de (minder nuttige) ‘monadic’ plus, die bv. wordt aangeroepen met ‘a = + b’. [↑](#footnote-ref-28)
30. Een extra (dummy) integer parameter wordt meegegeven door de compiler om de prefix operator te onderscheiden van de postfix versie. Vraag me niet wat de bedenker hiervan aan het roken was. [↑](#footnote-ref-29)
31. De << en >> operatoren worden eigenlijk altijd als losstaande functies geschreven, omdat het voor het algemeen aanvaarde gebruik van deze operatoren (geformatteerde tekst input en output) nodig is dat het linker argument van het type std::istream of std::ostream kan zijn, en aan die klassen kan een gebruiker niets toevoegen. [↑](#footnote-ref-30)
32. Deze operatoren kunnen alleen als methode worden gedefinieerd, niet als vrije functie. [↑](#footnote-ref-31)
33. Dit wordt in een volgend hoofdstuk besproken. [↑](#footnote-ref-32)
34. Sommige mensen vergelijken dit met een fabriek die veligheidsriemen en airbags plaats in testauto’s, maar niet in de productie modellen. [↑](#footnote-ref-33)
35. Maar dan moet je die operator== natuurlijk wel testen! [↑](#footnote-ref-34)
36. De voorganger van Catch2 had de naam Catch. [↑](#footnote-ref-35)
37. Helaas zie je, vooral in one-man open-source projecten, nogal eens ‘excuus-documentatie’ die is gegenereed uit alleen de declaraties, zonder toegevoegd kommentaar. Daar heeft dus niemand iets aan. [↑](#footnote-ref-36)
38. De header files kunnen ook ergens anders staan, maar dan moet je de Doxyfile aanpassen. [↑](#footnote-ref-37)
39. Maar die heb je natuurlijk niet, tenzij het constanten zijn. [↑](#footnote-ref-38)
40. De index begint op 1 (moet 0 zijn) en loopt tot <= array\_size, dat moet < zijn. [↑](#endnote-ref-0)
41. Als i == ( array\_size – 1 ) dan wordt het array element [ array\_size ] gelezen. Dit valt buiten het array. [↑](#endnote-ref-1)
42. De x die op 0 wordt gezet is een copie van het array element. Er veranderd dus niets in het array. [↑](#endnote-ref-2)
43. En, nog erger: dubbel leeswerk voor iemand die jouw code moet lezen, en de mogelijkheid dat een verbetering (bug fix, of een nieuwe feature) maar op 1 van de 2 plekken wordt doorgevoerd. Code duplicatie is een teken van ‘code smell’: de geur van slechte code. [↑](#footnote-ref-39)
44. Voor de richting van de pijl kan je als ezelsbruggetje gebruiken dat de wijdere kant van de pijl zit aan de kant van de klasse die meer eigenschappen heeft. [↑](#footnote-ref-40)
45. Niet echt alles: een constructors worden nooit geerfd. Dat zou ook raar zijn: een bal heeft meer attributen, en heeft dus ook een uitgebreidere constructor nodig. Er zijn nog andere dingen die niet geerft worden, die worden later besproken. [↑](#footnote-ref-41)
46. Het geldt zelfs voor assignment, maar daar wordt in deze cursus niet verder op in gegaan. [↑](#footnote-ref-42)
47. In talen die meer op gemak voor de programeur en minder op snelheid gefocust zijn wordt dit onderscheid doorgaans niet gemaakt en zijn alle klasse functies (methoden) virtual. Dit geldt bv. voor Python en Java. [↑](#footnote-ref-43)
48. Dit is in de taal C++ niet verplicht, maar het is wel sterk aan te raden (en in code die je in cursussen schrijf verplicht). De compiler kan dan namelijk checken of je wel echt een virtual functie uit de superklasse (her)definieert, zo niet dan krijg je een foutmelding. Daarnaast is het ook informatief voor iemand die je code leest. [↑](#footnote-ref-44)
49. Zoals eerder gezegd: een superklasse pointer of reference kan naar een subklasse object verwijzen. [↑](#footnote-ref-45)
50. Flash en EEPROM zijn twee technieken om ROM geheugen te maken dat electrisch herschreven kan worden. Het verschil met RAM is dat het schrijven van Flash en EEPROM veel langzamer is, en slijtage oplevert (het aantal wis/schrijf acties is meestal beperkt tot 1’000 .. 1’000’000 keer per sector). [↑](#footnote-ref-46)
51. Wij gebruiken daarvoor 1 kΩ weerstanden, herkenbaar aan de kleurencode bruin-zwart-rood. [↑](#footnote-ref-47)
52. SAM staat voor Smart Arm-based Micro-controllerroller, een afkorting van de fabrikant Atmel (nu overgenomen door Microchip) voor hun familie van ARM micro-controllerrollers. [↑](#footnote-ref-48)
53. Dit detail staat overigens niet vermeld in het datasheet hoofdstuk over GPIO. Soms moet je dus echt de hele datasheet doorlezen, of geluk hebben bij het googelen. [↑](#footnote-ref-49)
54. De zelfde truuk wordt door sam.h gebruikt om de dingen voor de juiste chip beschikbaar te stellen: er zijn veel meer SAM chips dan de ATSAM3X8E die in een Arduino Due wordt gebruikt. [↑](#footnote-ref-50)
55. Er zijn meer targets die (in meer of mindere mate) door bmptk en hwlib worden ondesrteund, bv. de Arduino Uno en de ESP8266. [↑](#footnote-ref-51)
56. Dat kan ook niet: de set() functie vereist dat het object niet const is. [↑](#footnote-ref-52)
57. Knight Industries Two Thousand [↑](#footnote-ref-53)
58. Dit heet Argument Dependent Lookup: de namespaces waarin de argument typen zijn gedeclareerd worden meegenomen bij het zoeken naar een passende functie die aangeroepen wordt. [↑](#footnote-ref-54)
59. bv. Omdat er dan minder draden hoeven te lopen naar van de micro-controllerroller naar de plek waar de IO pinnen nodig zijn, of omdat de pinnen van een extender chip vaak meer stroom kunnen leveren dan die van een micro-controllerroller. Of gewoon omdat de pinnen van de micro-controllerroller op zijn: goedkope micro-controllerrollers hebben vaak veel minder IO pinnen, een PIC10F200 bv. maar 6. [↑](#footnote-ref-55)
60. Een andere reden is dat de SPI hardware je dwingt bepaalde pinnen te gebruiken. De bit-banged implementatie in hwlib is wat dat betreft flexibeler: je kan zelf kiezen welke pinnen je gebruikt. [↑](#footnote-ref-56)
61. Wat get() teruggeeft is niet belangrijk. De compiler eist dat er een return statement in staat, anders hadden we die ook nog weg kunnenlaten. [↑](#footnote-ref-57)
62. Voor de meeste professionals zijn dit de ’oer’ patterns die beschreven zijn in het boek “Design Patterns: Elements of Reusable Object-Oriented Software”. [↑](#footnote-ref-58)
63. De I2C standaard staat ‘clock stretching’ toe: een slaaf verlengt een clock puls door de SCL lijn laag te houden totdat de slaaf het bitje klaar heeft. Deze mogelijkheid wordt in de praktijk niet vaak gebruikt. [↑](#footnote-ref-59)
64. Voor geinteresseerden: het OLED schermpje heeft een SSD13096 controller chip. In de datasheet van die chip vind je hoe je hem moet aansturen. Voor een aantal details (initiele settings) kan je de hwlib code raadplegen. [↑](#footnote-ref-60)